Join Processing in Relational Databases

PRITI MISHRA and MARGARET H. EICH

Computer Science & Engineering Department, Southern Methodist University, Dallas, Texas 75275

The join operation is one of the fundamental relational database query operations. It facilitates the retrieval of information from two different relations based on a Cartesian product of the two relations. The join is one of the most difficult operations to implement efficiently, as no predefined links between relations are required to exist (as they are with network and hierarchical systems). The join is the only relational algebra operation that allows the combining of related tuples from relations on different attribute schemes. Since it is executed frequently and is expensive, much research effort has been applied to the optimization of join processing. In this paper, the different kinds of joins and the various implementation techniques are surveyed. These different methods are classified based on how they partition tuples from different relations. Some require that all tuples from one be compared to all tuples from another; other algorithms only compare some tuples from each. In addition, some techniques perform an explicit partitioning, whereas others are implicit.

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

General Terms: Algorithms

Additional Key Words and Phrases: Database machines, distributed processing, join, parallel processing, relational algebra

INTRODUCTION

The database join operation is used to combine tuples from two different relations based on some common information. For example, a course-offering relation could contain information concerning all classes offered at a university and a student-registration relation could contain information for which courses a student has registered. A join would typically be used to produce a student schedule, which includes data about required textbooks, time, and location of courses, as well as general student identification information. The join operation is one of the operations defined in the relational data model [Codd 1970, 1972]. It is used to combine tuples from two or more relations. Tuples are combined when they satisfy a specified *join condition*. The result tuples have the combined attributes of the two input relations. In the above example, the join condition could

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1992 ACM 0360-0300/92/0300-0063 $01.50

CONTENTS

INTRODUCTION
1 JOIN OPERATION
1.1 Join versus Cartesian Product
1 2 Types of Joins
2. IMPLEMENTATION OF JOINS
2.1 Nested-Loops Join
2 2 Sort-Merge Join
2 3 Hash Join Methods
2 4 Summary
3. JOINS BASED ON SPECIAL DATA STRUCTURES
3 1 Join Indexes
3 2 Bc-Trees
3 3 T-Trees
3.4 Kd-Trees
3 5 Prejoins
3 6 Summary
4 JOIN CLASSIFICATION
4 1 Partitioning
4.2 Classification
5 JOIN PROCESSING IN A DISTRIBUTED ENVIRONMENT
5 1 Factors in Distributed Processing
5 2 Join Algorithms
5.3 Summary
6. HARDWARE SUPPORT FOR JOINS
6.1 Hardware Approaches
6.2 Nested-Loops Join
6 3 Sort-Merge Join
6.4 Hash-Based Joins
6.5 Summary
7 OTHER JOIN ISSUES
7.1 Selectivity Factor
7 2 Optimal Nesting of Joins
7 3 Hash Joins
7.4 Indexing and Clustering
7 5 Partitioning of Relations
7.6 Join-Type Processing in Nonrelational Databases
8. CONCLUSIONS
APPENDIX
ACKNOWLEDGMENTS
REFERENCES
BIBLIOGRAPHY

be to combine tuples when they have the same course number and offering number values. The resulting tuples would contain all the offering data, as well as all the student data.

Consider, for example, the relations R and S shown below:

Relation R

employee	payscale
james	1
jones	2
johns	1
smith	2

Relation S

payscale	pay
1	10000
2	20000
3	30000

A join of R and S with the join condition R(payscale) = S(payscale) results in the following relation:

Relation Q

employee	payscale	payscale	pay
james	1	1	10000
jones	2	2	20000
johns	1	1	10000
smith	2	2	20000

Normalization of relations in relational databases results in related information being stored in separate relations [Dutka and Hanson 1989; Fagin 1979; Kent 1983]. Hence, queries that require data from several relations are very common. The join operation is used to satisfy such queries.

The join operation has been studied and discussed extensively in the literature because it is one of the most time-consuming and data-intensive operations in relational query processing. It is also important that joins be performed efficiently because they are executed frequently. Many optimizations of the join operation automatically include the optimization of the other common relational operations, namely, the select and project

operations, because they are implicit in the join operation.

Joins may be implemented in many ways, and it has been found that certain techniques are more efficient than others in some computing environments. This paper collates the information available in the literature in order to study the unique and common features of join algorithms. With this information, it is possible to classify the algorithms into categories. The categorization scheme included in Section 4 provides a simple pictorial technique to visualize the different approaches to implement join processing.

A join can be two-way or multiway. It is said to be two-way when it is performed on two relations and multiway when more than two relations are joined. A multiway join produces the same result as a series of two-way joins. A join between n relations is usually executed as a sequence of $(n - 1)$ two-way joins [Mackert and Lohman 1986].

Join processing has been studied from many points of view:

Query optimization. Issues such as selecting best available access paths, determining optimal nesting of joins in multiway joins, and devising strategies for distributed join processing are discussed in a number of articles; for example, Christodulakis [1985], Kim et al. [1985], Perrizo et al. [1989]; Segev [1986], Swami and Gupta [1988], Yoo and Lafortune [1989], and Yu et al. [1985, 1987].

Optimizing I/O. Reducing the total number of page accesses by means of efficient indexing [Cheiney et al. 1986; Desai 1989; Goyal et al. 1988; Lehman and Carey 1986] and clustering has been the subject of much discussion [Blasgen and Eswaran 1977; Chang and Fu 1980; Omiecinski 1989; Ozkarahan and Bozsahin 1988].

Optimizing buffer usage. Buffer usage can be optimized by reducing the number of times a page is accessed [Omiecinski 1989; Pramanik and Fotouhi 1985]. Some algorithms have been modified to reduce the amount of buffer space needed, whereas others have been tuned to make the best use of the available space [Fotouhi and Pramanik 1989; Goyal et al. 1988; Hagmann 1986; Sacco and Schkolnick 1986].

Reducing computation. The number of comparisons between tuples can be reduced, such as by using partitioning or by using the simple sort-merge algorithm. This aspect of the join operation will be discussed at length later.

Hardware support. Processing has been speeded up by means of direct hardware support such as special join processors and indirect support in the form of hardware hashing units and sorters, data filters, and associative memories. All database machines incorporate one or more of these features. The number of database machine designs is too numerous to list individually here, but a comprehensive survey can be found in Su [1988].

Parallel processing. Many join algorithms have a high degree of inherent parallelism [Bitton et al. 1983; Schneider and DeWitt 1989]. Parallelization is seen in two forms: in the shape of specialized database machines and of general-purpose multiprocessors [Baru and Frieder 1989; Baru et al. 1987; Menezes et al. 1987; Valduriez and Gardarin 1982, 1984]. The latter appear to have an edge over database machines for a number of reasons such as cost, scalability, and availability.

Physical database design. Relations in a database must be such that any pair can be joined without loss of data and that the result relation contains only valid data [Aho et al. 1979; Dutka and Hanson 1989]. The literature on join dependencies [Beeri and Vardi 1981; Gyssens 1986], lossless join decomposition, and theory of separability [Whang et al. 1985] that affect physical database design is voluminous. An indication of this is the extensive bibliography found in Dutka and Hanson [1989].

Although all of the above topics are not covered in detail in this paper, an

extensive bibliography on all topics is provided. For example, the relationship between the join operation and query processing is not discussed in detail. The topic is mentioned briefly where appropriate, and several references are provided that treat the issue at length.

The remainder of the paper is organized as follows. Section 1 contains definitions and descriptions of the basic join operation and its various derivatives. Section 2 describes some of the numerous implementations. Join algorithms based on specific data structures are described in Section 3. A classification scheme for join algorithms is described in Section 4. Distributed join processing and hardware support for join processing are discussed in Sections 5 and 6, respectively. Miscellaneous issues related to join performance, such as the effect of data distribution on parallel algorithms and joinlike operations in nonrelational database systems, are contained in Section 7. Section 8 contains a brief summary of the paper. Notations and abbreviations used in the paper are summarized in the appendix.

1. JOIN OPERATION

In this section, first the relationship between the join operation and the Cartesian product is explored. Next, the many derivatives of the basic join operation are reviewed.

1.1 Join versus Cartesian Product

The join operation is closely related to the Cartesian product. The *Cartesian product* of two relations concatenates each tuple of the first relation with every tuple of the second relation. The result of this operation on relations R and S, with n and m number of tuples, respectively, consists of a relation with $(n \times m)$ tuples and the combined attributes of the input relations. The Cartesian product of the two example relations R and S is the relation Q:

Relation $Q = R \times S$

employee	payscale	payscale	pay
james	1	1	10000
james	1	2	20000
james	1	3	30000
jones	2	1	10000
jones	2	2	20000
jones	2	3	30000
johns	1	1	10000
johns	1	2	20000
johns	1	3	30000
smith	2	1	10000
smith	2	2	20000
smith	2	3	30000

The join operation is used to combine related tuples from two relations into single tuples that are stored in the result relation. The desired relationship between tuples or some attributes in the tuples is specified in terms of the join condition. In its simplest form, the join of R and S is written as

$$R \bowtie_{r(a)\theta s(b)} S,$$

where $r(a)\theta s(b)$ defines the join condition. The θ operator defines the condition that must hold true between the attributes $r(a)$ and $s(b)$ of R and S, respectively. This general join is called a *theta-join*. The theta operator can be one of the following: $=, \neq, <, >, \leq, \geq$. The attributes used to define the join condition must be comparable using the theta operator.

In its most general form, the join condition consists of multiple simple conditions of the form described above, connected with the logical connective AND [Desai 1990; El-Masri and Navathe 1989; Maier 1983]:

$$condition \wedge condition \wedge \ldots \wedge condition.$$

The presence of the join condition distinguishes the join operation from the Cartesian product. In effect, the join operation may be said to be equivalent to a Cartesian product followed by a select operation [El-Masri and Navathe 1989], where the select operation is implicit in

the join condition. Or,

$$R \bowtie_{r(a)\theta s(b)} S \equiv \sigma_{r(a)\theta s(b)}[R \times S].$$

As such, the result of joining two relations is a subset, proper or otherwise, of the Cartesian product of the two relations. The tuples of either relation that do not participate in the join are called *dangling tuples* [Ullman 1988].

The result of joining relations R and S with n and m attributes, respectively, is a relation Q with $(n + m)$ attributes. The relation Q has one tuple for each pair of tuples from R and S that satisfies the join condition. The result relation Q may then be defined as

$$Q = \{t \mid t = rs \wedge r \in R \wedge s \in S \wedge t(a)\theta t(b)\}$$

For example, joining R and S with the join condition R(payscale) < S(payscale) results in a relation Q is as follows:

Relation $Q = R \bowtie_{r(\text{payscale}) < s(\text{payscale})} S$

employee	payscale	payscale	pay
james	1	2	20000
james	1	3	30000
jones	2	3	30000
johns	1	2	20000
johns	1	3	30000
smith	2	3	30000

This relation is a proper subset of the cross product of R and S.

Both Cartesian product and join are time-consuming and data-intensive operations. In the most naive implementation, each tuple in one relation must be compared to every tuple in the other relation. Therefore, the complexity of the operation for relations with n tuples each is $O(n^2)$. Further, the staging of tuples for the comparison steps requires an enormous number of I/O operations. The large number of algorithms that has been devised to execute joins is a result of efforts to reduce the number of comparisons, reduce the amount of I/O, or both.

1.2 Types of Joins

Several other types of joins have been defined. Some are direct derivatives of the theta-join; others are combinations of the theta-join and other relational operations such as projection. Some have been implemented as primitive operations in database management systems, whereas others are currently found only in the database literature. Variations of the join operation seen in database management systems and in the literature are discussed below.

1.2.1 Equijoin

The most commonly used theta operator is the equality operator; in these cases, the join is called an *equijoin*. For all other theta operators the join is called a *nonequijoin*. The result relation Q is defined as follows:

$$Q = \{t \mid t = rs \wedge r \in R \wedge s \in S \wedge t(a) = t(b)\}.$$

In other words, the result relation contains tuples t made up of two parts, r and s, where r must be a tuple in relation R and s must be a tuple in relation S. In each tuple t, the values of the join attributes $t(a)$, belonging to r, are identical in all respects to the values of the join attributes $t(b)$, belonging to s.

The result of joining R and S such that R(payscale) = S(payscale) is the relation Q is as follows:

Relation $Q = R \bowtie_{r(\text{payscale}) = s(\text{payscale})} S$

employee	payscale	payscale	pay
james	1	1	10000
jones	2	2	20000
johns	1	1	10000
smith	2	2	20000

1.2.2 Natural Join

In the theta-join, the tuples from the input relation are simply concatenated. Therefore, the result tuple, in the case of equijoins, contains two sets of the join attributes that are identical in all respects. Thus, one set of join attributes in the result relation can be removed without any loss of information. Certainly,

the removal of one column from the result relation should reduce the amount of storage space required. In most cases, it is required that one of these sets be projected out to get the final result. This gives rise to a derivative of the equijoin —the *natural join*. The natural join can, therefore, be defined as an equijoin on attributes with the same name in both relations, followed by projection to remove one set of the join attributes.

The natural join operation can be written as

$$R * S = \pi_{(a_i b_j - s(a))}[R \bowtie_{r(a) = s(a)} S],$$

where a_i and b_i are the attributes in relations R and S, respectively. The result relation Q is given by

$$Q = \{t \mid t = (rs - s(a)) \wedge r \in R \wedge s \in S \wedge r(a) = s(a)\}.$$

The result of $R * S$ on the payscale attribute is the following relation:

Relation $Q = R * S$

employee	payscale	pay
james	1	10000
jones	2	20000
johns	1	10000
smith	2	20000

1.2.3 Semijoin

In the conventional execution of the join operation, the resulting relation has all the attributes of both input relations. Sometimes it is required that only the attributes of one of the relations be present in the output relation. The *semijoin* operation is designed to perform such a join [Bernstein and Chiu 1981; Bernstein and Goodman 1979a]. It has also been defined as an operation that selects a set of tuples from one relation that relate to one or more tuples in another relation. The relationship is defined by the join condition. (Inequality semijoins are discussed in Bernstein and Goodman [1979b, 1980].) It is equivalent to the join of the two relations followed by a project operation that results in the attributes of the second relation being dropped from the output relation. The initial join itself may be performed by any of the join techniques.

The semijoin operation is written as

$$R \ltimes_{r(a) \theta s(b)} S = \pi_{a_i}[R \bowtie_{r(a) \theta s(b)} S],$$

and the result relation Q is given by

$$Q = \{t \mid t = r \wedge t(a) \theta s(b) \wedge r \in R \wedge s \in S\}.$$

A list of payscales such that there is at least one employee who gets paid according to that scale is a semijoin between S and R. That is,

$$Q = S \ltimes_{s(\text{payscale}) = r(\text{payscale}))} R.$$

The result relation Q is the following relation:

payscale	pay
1	10000
2	20000

Unlike most other join operations, the semijoin operation is not commutative, that is,

$$(R \ltimes S) \neq (S \ltimes R).$$

An alternative expression for the semijoin operation is

$$R \ltimes S = \pi_{a_i}[R \bowtie (\pi_{s(b)} S)].$$

No join condition has been specified in this expression because it represents the general case. Although the effect is the same in both cases, this version of the semijoin reduces the size of the second relation participating in the join operation. The initial projection on S results in a smaller relation while maintaining all data needed to get the result of the semijoin. This feature is especially useful when R and S are at different sites and S must be transferred to R's site.

Semijoins can be used to reduce the processing load of regular joins and to

avoid the creation of large intermediate relations [Kambayashi 1985; Perrizo et al. 1989; Yoo and Lafortune 1989]. Semijoins have been found to be a useful tool in the processing of certain kinds of queries, particularly tree queries [Bernstein and Chiu 1981]. Efficient procedures for handling cyclic queries using semijoins are discussed in Kambayashi [1985] and Yoshikawa and Kambayashi [1984].

1.2.4 Outerjoin

Strictly speaking, the outerjoin is an extension of the join operation [Date 1983; El-Masri and Navathe 1989; Rosenthal and Reiner 1984]. It is also called an *external join* [Gardarin and Valduriez 1989]. The *outerjoin* operation was defined to overcome the problem of dangling tuples. In conventional joins, tuples that do not participate in the join operation are absent from the result relation. In certain situations it is desired that the result relation contain all the tuples from one or both relations even if they do not participate in the join (they must be padded with null values as needed).

Depending on whether the result relation must contain the nonparticipant tuples from the first, second, or both relations, three kinds of outer joins are defined: the *left-outerjoin*, the *right-outerjoin*, and the *full-outerjoin*, respectively [El-Masri and Navathe 1989]. The corresponding join symbols are $=\bowtie$, $\bowtie=$, and $=\bowtie=$. The left and right outerjoins are collectively referred to as one-sided or directional joins. It must be noted that only the full-outerjoin is commutative. Although the left and right outer joins are not commutative, they are related as follows:

$$(R =\bowtie S) = (S \bowtie= R)$$

The result relation for a full-outerjoin will contain three types of tuples: tuple pairs that satisfy the join condition, dangling tuples from R with padded S-attributes, and dangling tuples from S with padded R-attributes. The result relation Q can be written as

$$Q = \{t \mid t = rs \wedge r \in R \wedge s \in S \wedge r(a) \theta s(b)\} + \{t \mid t = rn \wedge r \in R\} + \{t \mid t = ns \wedge s \in S\},$$

where, n represents the padding of the tuples with null values.

An example of a right outer join using the relations R and S is

$$Q = R \bowtie=_{(r(\text{payscale}) = s(\text{payscale}))} S.$$

The result relation Q is as follows:

Relation Q

employee	payscale	payscale	pay
james	1	1	10000
jones	2	2	20000
johns	1	1	10000
smith	2	2	20000
$\perp$	$\perp$	3	30000

Here the symbol $\perp$ represents the null value.

This operation is particularly useful in dealing with situations where there are null values in the join attributes of either relation. At this point, not many DBMSs support outerjoins directly. The processing of queries containing outerjoins is discussed in Rosenthal and Galindo-Legaria [1990] and Rosenthal and Reiner [1984].

A scheme for query optimization in heterogeneous, distributed database systems that uses outerjoins to help in database integration is discussed in Dayal [1985]. One-sided outer joins may be used in aggregate function evaluation in statistical databases [Ozsoyoglu et al. 1989]. Semiouterjoins, which combine the qualities of outerjoins and semijoins, have been used in query processing in multidatabase systems [Dayal 1985].

1.2.5 Self-Join

The self-join may be considered a special case of the theta-join [Hursch 1989]. The only difference is that the input relation

is joined with itself. The output relation Q is given by

$$Q = \{t \mid r \in R \wedge s \in S \wedge R = S \wedge t = rs \wedge r(a) \theta s(b)\}.$$

The operation of the self-join can be illustrated using example relation R. To get a list of pairs of employees such that each pair of employees gets the same pay, the relation R can be selfjoined with itself. That is,

$$Q = R \bowtie_{r(\text{payscale}) = r(\text{payscale})} R.$$

Then, the result relation Q is as follows:

Relation $Q = R \bowtie_{r(\text{payscale}) = r(\text{payscale})} S$

employee	payscale	payscale	employee
james	1	1	james
james	1	1	johns
jones	2	2	jones
jones	2	2	smith
johns	1	1	johns
johns	1	1	james
smith	2	2	smith
smith	2	2	jones

1.2.6 Composition

Composition was first described as one of the operations on relations in Codd [1970]. Two relations are said to be *composable* if they are joinable. The composition of two relations is equivalent to joining the two relations, then projecting out both sets of join attributes. Formally, this is written as

$$R \otimes S = \pi_{(a_i b_j - r(a) - s(b))}[R \bowtie S],$$

where $\otimes$ represents the composition operator and $\bowtie$ without the join condition represents any join between the relations R and S.

The *natural composition* of R and S is defined as the composition based on the natural join between R and S [Codd 1970]. It is written as

$$R \cdot S = \pi_{(a_i b_j - r(a) - s(a))}[R * S].$$

The result relation Q, of the natural composition of R and S, is given by

$$Q = \{t \mid t = (rs - r(a) - s(a)) \wedge r \in R \wedge s \in S \wedge r(a) = s(a)\}.$$

The natural composition of the example relations R and S on the attribute payscale is the relation Q:

Relation $Q = R \cdot S$

employee	pay
james	10000
jones	20000
johns	10000
smith	20000

This operation has been discussed further in Agrawal et al. [1989] and Ozkarahan [1986].

1.2.7 Division

Strictly speaking, the division operation is not a member of the set of relational algebra operations. However, it is included in discussions on the relational algebra because it is commonly used in database applications [Desai 1990; El-Masri and Navathe 1989; Maier 1983]. The divide operator allows a tuple to be retrieved from one relation if it is related to all tuples in another based on some predefined condition. For example, we might find each employee of one company who has a salary larger than that of all employees of another company.

Let relations R and S have attributes a_i and b_j such that $b_j \subseteq a_i$. Let $c_k = a_i - b_j$. That is, c_k is the set of attributes in relation R that is not in relation S. Then, R divided by S is written as

$$R \div S = \pi_{c_k}(R) - \pi_{c_k}((\pi_{c_k}(R) \times S) - R).$$

The result relation Q is the quotient of R divided by S and is given by

$$Q = \{t \mid \forall s \in S \exists r \in R \text{ such that } t \| s = r\}.$$

In other words, the result relation Q is the maximal subset of the relation $\pi_{c_k}(R)$ such that the Cartesian product of relations Q and S is contained in R.

To illustrate the division operation, let us redefine the relations.

Relation R

A	B
A1	B1
A2	B1
A3	B1
A1	B2
A2	B2
A3	B2
A1	B3
A2	B3

Relation S

A
A1
A2
A3

Then, the result relation Q is as follows:

Relation $S = R \div S$

B
B1
B2

Several hash-based algorithms for performing division are presented in Graefe [1989]. An implementation of the division operation on a shuffle-exchange network is described in Baba et al. [1987].

2. IMPLEMENTATIONS OF JOINS

The techniques and methods used to implement joins are discussed in the following sections. Unless otherwise noted, the algorithms are used to implement the theta-join. The description of a method includes the basic algorithm, general discussion of the method, special data structures (if any), and applicability and performance of the technique. General problems that apply to a whole class of join techniques, such as the effect of clustering or the effect of collisions on the hash join techniques, are discussed separately in Section 7.

2.1 Nested-Loops Join

Nested-loops join is the simplest join method. It follows from the definition of the join operation. One of the relations being joined is designated as the *inner relation*, and the other is designated as the *outer relation*. For each tuple of the outer relation, all tuples of the inner relation are read and compared with the tuple from the outer relation. Whenever the join condition is satisfied, the two tuples are concatenated and placed in the output buffer.

Algorithm

The algorithm for performing

$$R \bowtie_{r(a)\theta s(b)} S$$

is as follows:

```
for each tuple s do
    { for each tuple r do
        { if r(a)θs(b) then
          concatenate r and s
          place in relation Q }}
```

Note that for efficiency, the relation with higher cardinality (R in this case) is chosen to be the inner relation.

Discussion

In practice, a nested-loops join is implemented as a nested-block join; that is, tuples are retrieved in units of blocks rather than individually [El-Masri and Navathe 1989]. This implementation can be briefly described as follows. The inner relation is read one block at a time. The number of main memory blocks available determines the number of blocks read from the outer relation. Then all tuples in the inner relation's block are joined with all the tuples in the outer relation's blocks. This process is repeated with all blocks of the inner relation before the next set of outer relation blocks is read

in. The amount of reduction in I/O activity (compared to a simple tuple-oriented implementation) depends on the size of the available main memory.

A further step toward efficiency consists of "rocking" the inner relation [Kim 1980]. In other words, the inner relation is read from top to bottom for one tuple of the outer relation and bottom to top for the next. This saves on some I/O overhead since the last page of the inner relation which is retrieved in one loop is also used in the next loop.

Performance

In the above algorithm, it is seen that each tuple of the inner relation is compared with every tuple of the outer relation. Therefore, the simplest implementation of this algorithm requires $O(n \times m)$ time for execution of joins.

The block-oriented implementation of the nested-loops join optimizes on I/O overhead in the following way. Since the inner relation is read once for each tuple in the outer relation, the operation is most efficient when the relation with the lower cardinality is chosen as the outer relation. This reduces the number of times the inner loop is executed and, consequently, the amount of I/O associated with reading the inner relation. An analysis of buffer management for the nested-loops method with rocking shows that buffering an equal number of pages for both relations is the best strategy [Hagmann 1986].

If the join attributes can be accessed via an index, the algorithm can be made much more efficient. Such an implementation has been described in Blasgen and Eswaran [1977].

Applicability

The exhaustive matching performed in this method makes it unsuitable for joining large relations unless the *join selectivity factor*, the ratio of the number of tuples in the result of the join to the total number of tuples in the Cartesian product, is high. If the selectivity factor is low, the effort of comparing every tuple in one relation with every tuple in the other is further unjustified.

The simplicity of this algorithm has made it a popular choice for hardware implementation in database machines [Su 1988]. It has been found that this algorithm can be parallelized with great advantage. The parallel version of this algorithm is found to be more efficient than most other methods. Thus, we see that for the nested-loops join, a parallel implementation of an inefficient serial algorithm looks good. More details concerning the parallel approach can be found in Section 6.

This algorithm is also chosen in a proposed model for main memory databases called the DBGraph storage model [Pucheral et al. 1990]. The entire database is represented in terms of a graph-based data structure called the DBGraph. A set of primitive operations is defined to traverse the graph, and all database operations can be performed using these primitive operations. Advantages of this model are efficient processing of all database operations and complex queries, compact storage, and uniform treatment of permanent and transient data.

2.2 Sort-Merge Join

The *sort-merge* join is executed in two stages. First, both relations are sorted on the join attributes. Then, both relations are scanned in the order of the join attributes, and tuples satisfying the join condition are merged to form a single relation. Whenever a tuple from the first relation matches a tuple from the second relation, the tuples are concatenated and placed in the output relation.

Algorithm

The exact algorithm for performing a sort-merge join depends on whether or not the join attributes are nonkey attributes and on the theta operator. In all cases, however, it is necessary that the two relations be physically ordered on their respective join attributes.

The algorithm for performing equijoins is as follows:

```
Stage 1: Sort process
sort R on r(a);
sort S on s(b);

Stage 2: Merge process
read first tuple from R;
read first tuple from S;
for each tuple r do
    { while s(b) < r(a)
        read next tuple from S;
        if r(a) = s(b) then
            join r and s
            place in output relation Q };
```

Discussion

The merge process varies slightly depending on whether the join attributes are primary key attributes, secondary key attributes, or nonkey attributes. If the join attributes are not the primary key attributes, several tuples with the same attribute values may exist. This necessitates several passes over the same set of tuples of the inner relation. The process is described below.

Let there be two tuples, $r1$ and $r2$, in R that have a given value x of the join attribute $r(a)$ and three tuples, $s1$, $s2$, and $s3$, in S that have the same value x of the join attribute $s(b)$. If the above join algorithm is used then when $r2$ is the current tuple in R, the current tuple in S would be the tuple following $s3$. Now the result relation must also include the join of $r2$ with $s1$, $s2$, and $s3$. To achieve this, the above algorithm must be modified to remember the last $r(a)$ value and the point in S where it started the last inner loop. Whenever it encounters a duplicate $r(a)$ value, it backtracks to the previous starting point in S. This backtracking can be especially expensive in terms of the I/O if the set of tuples does not fit into the available main memory and the tuples have to be retrieved from secondary storage for each pass.

Performance

If the relations are presorted, this algorithm has a major advantage over the brute force approach of the nested-loops method. The advantage is that each relation is scanned only once. Further, if the join selectivities are low, the number of tuples compared is considerably lower than in the case of the nested-loops join. It has been shown that this algorithm is most efficient for processing on a uniprocessor system [Blasgen and Eswaran 1977].

The processing time depends on the sorting and merging algorithms used. If the files are already sorted on the join attributes, the cost is simply the cost of merging the two relations. In general, the overall execution time is more dependent on the sorting time, which is usually $O(n \log n)$ for each relation, where n is the cardinality of the relation.

Execution is further simplified if the join attributes are indexed in both relations. The Simple TID algorithm starts by scanning the join attribute indices and making a list of tuple-id pairs corresponding to the tuple pairs that participate in the join [Blasgen and Eswaran 1977]. In the next stage, the tuples themselves are fetched and physically joined. This approach reduces the number of tuples read into main memory and, as a result, the amount of I/O needed. If the index is not the primary index, however, retrieval of the records may be rather inefficient [El-Masri and Navathe 1989].

Applicability

If no indexes exist on the join attributes, if not much is known about the selectivities, and if there is no basis for choosing a particular join algorithm, then this algorithm is often found to be the best choice [Blasgen and Eswaran 1977; Su 1988].

With the help of hardware sorters, this algorithm makes a good candidate for hardware implementation. Several database machines, such as VERSO [Bancilhon et al. 1983] use this as the primary join method.

The sort-merge join algorithm can also be used to implement the full-outerjoin. The algorithm for performing the

full-outerjoin

$$R = \bowtie =_{r(a) = s(b)} S$$

using the sort-merge method is as follows:

```
sort R on r(a);
sort S on s(b);
read first tuple from R;
read first tuple from S;
for each tuple r do
    { while s(b) < r(a)
    write s into Q
    pad R-attributes with null values
    read next tuple from S;
    if r(a) = s(b) then
        join r and s
        place in output relation Q };
```

2.3 Hash Join Methods

The success of the sort-merge join lies in the fact that it reduces the number of comparisons between tuples. A tuple from the first relation is not compared with those tuples in the second relation with which it cannot possibly join. *Hash join* methods achieve the same effect in another way [Bratbergsengen 1984; Goodman 1981]. They attempt to isolate the tuples from the first relation that may join with a given tuple from the second relation. Then, tuples from the second relation are compared with a limited set of tuples from the first relation.

A large number of join methods using hashing has been proposed and/or implemented. Some are discussed in the following sections.

2.3.1 Simple Hash Join Method

With the *simple hash join*, the join attribute(s) values in the first relation are hashed using a hash function. These hashed values point to a hash table in which each entry may contain entire tuples or only tuple-ids [DeWitt et al. 1984]. In the latter case, it may be useful to store the key values as well. Depending on the efficiency of the hash function, one or more entries may hash to the same bucket. Then for each tuple of the other relation participating in the join, the join attribute values are hashed using the same hashing function as before. If the values hash to a nonempty bucket of the previous hash table, the tuple(s) in the hash table bucket are compared with the tuple of the second relation. The tuples are joined if the join condition is satisfied.

Algorithm

The algorithm for performing

$$R \bowtie_{r(a) \theta s(b)} S$$

is as follows:

```
for each tuple s in S do
    { hash on join attributes s(b)
    place tuples in hash table based on hash
    values};
for each tuple r do
    { hash on join attributes r(a)
    if r hashes to a nonempty bucket of hash
    table for S
    then { if r θ-matches any s in bucket
        concatenate r and s
        place in relation Q }};
```

In the above algorithm, the same hashing function must be used for both relations.

Discussion

The hash table should ideally be created for the relation with the fewest distinct values of the join attributes. This would require maintaining detailed statistics on each attribute of the relation or determining the number of distinct values on the fly. In order to avoid this overhead, the hash table is usually created for the smaller of the two input relations [Bratbergsengen 1984]. This optimizes on the amount of space needed to store the hash table. This property is particularly useful in the case of main memory processing, where the entire hash table can be placed in memory.

Nonequijoins are difficult to implement since they require that the hashing function used maintain the correct ordering of the tuples. Hash functions with this property are not uncommon; how-

ever, they have the problem that they do not easily lead to uniform distribution.

Performance

As a class, hash-based joins have been found to be some of the most efficient join techniques [Gerber 1986]. The complexity of this method is $O(n + m)$ because both relations are scanned once. The performance of this method depends on the performance of the hash function. If the hash function were perfect, it would be possible to join tuples whenever a tuple from the second relation hashed to a nonempty entry of the hash table. Hash collisions decrease the efficiency greatly because each tuple that hashes to a nonempty entry must be checked to see if it satisfies the join condition. A further problem with hash-based join methods is that elimination of duplicates might be harder because of collisions [Agrawal et al. 1989].

Applicability

Hardware hashing units have made hardware implementations of this method feasible. This is discussed in detail in a later section.

The simple hash join method can also be used to implement one-sided joins, that is, left- and right-outerjoins. The algorithm for performing the left-outerjoin,

$$R = \bowtie_{r(a)\theta s(b)} S,$$

using the simple hash join technique is as follows:

```
for each tuple s in S do
    { hash on join attributes s(b)
    place tuples in hash table based on hash
    values; }
for each tuple r do
    { hash on join attributes r(a)
    if r hashes to a nonempty entry of hash
    table for S
    and if suitable s found
    then { concatenate r and s
        place in relation Q }
    else write r into Q
    pad S-attributes with null values};
```

With a little modification, a similar algorithm can be used to perform the right-outerjoin.

2.3.2 Hash-Partitioned Joins

Hash-partitioned joins attempt to optimize join execution by partitioning the problem into parts. They use hashing to decompose the join problem into several subproblems that are much easier to solve. The divide-and-conquer approach has been found to have a number of advantages; not only is the overall efficiency improved, but partitioning makes the parallelization of the algorithm easier by allowing independent pairs to be processed in parallel. This results in further performance improvement. The general process works as follows.

A hash function, referred to as the *split function*, is used to partition the tuples in each relation into a fixed number of disjoint sets. The sets are such that a tuple hashes into a given set if the hashed value of its join attributes falls in the range of values for that set. Then the tuples in the first set of one relation can match only with the tuples in the first set of the second relation. Thus, the processing of different pairs of corresponding partitions from the two sets are independent of other sets. As a result, these partition pairs can be processed in parallel. Figure 1 shows the load reduction. The horizontal side represents tuples in relation S, and the vertical is for relation R. In both Figures 1a and 1b the shaded area represents the number of tuples that must be actually compared in the join process. It can be seen that partitioning reduces the join load considerably. Tuples in partition pairs from both relations may be joined by a simple hash join or any other join method.

Several implementations of hash-partitioned joins have been described in the literature; for example, GRACE hash join, hybrid hash join, and the hashed loops join [DeWitt and Gerber 1985]. Some of these are discussed in the following sections.

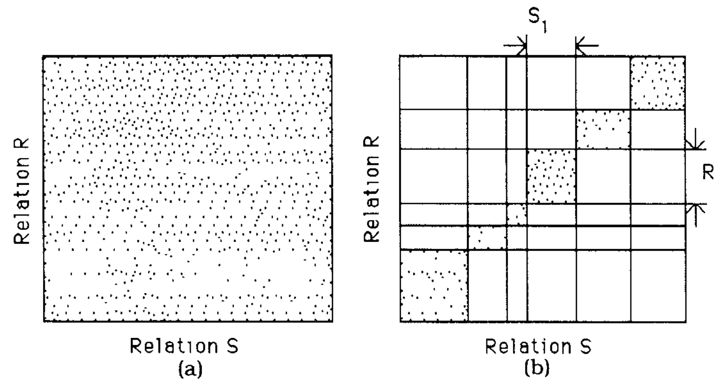

Figure 1. Reduction of join load. (a) No partitioning; (b) with partitioning.

Simple Hash-Partitioned Join. In the *simple hash-partitioned join* [Gerber 1986], not all the partitions are created at the beginning of the operation; instead, each partition pair is created and used up before the next is generated.

As in the description above, the number of partitions and the range of attribute values in each partition are determined. Before the relations are scanned, the range of hash values for the current partition is chosen. Ideally, the ranges should be such that all partitions are of equal size. Since this is hard to accomplish, the ranges are determined by dividing the total range of attribute values into equal-sized subranges. The number of subranges is equal to the number of partitions desired. This, in turn, depends on factors like the amount of main memory available and the number of processors. Next, the outer relation is scanned, and the attribute values are hashed. If the hash value falls in the current range of hash values, the tuple is inserted into the hash table for the current partition; otherwise, it is written into a temporary file. The hash table for the partition is complete when the whole relation has been scanned.

Next, the inner relation is scanned. The join attributes are hashed using the same hash function. If the hashed value falls in the current range, the join process is initiated; otherwise, the tuple is written into a temporary file. The process is repeated using the temporary files as input until all the tuples have been used up in at least one of the relations.

Algorithm

The algorithm for performing

$$R \bowtie_{r(a)\theta s(b)} S$$

is as follows:

```
repeat
    choose hash range for current partition;
    for every s do
        { hash attribute values;
        if value falls in current range then
            make entry in partition hash
            table
            else write tuple in temp-S };
    for every r do
        { hash on join attributes;
        if value falls in current range then
            { if match found in hash table
            then
                { concatenate tuples r and s
                place in output relation Q}
                else discard tuple}
            else write tuple in temp-R };
    S := temp-S;
    R := temp-R;
until (|S| = null);
```

Temp-R and temp-S are temporary relations used to avoid processing of all tuples at each step in the algorithm. The smaller relation S is partitioned first to reduce the execution time.

Discussion

This is a naive implementation of a partition-based join. An obvious source of inefficiency lies in the fact that a tuple may be read and written into the temporary relation several times before it actually participates in a join.

A variation of this algorithm, called the *simple toss hash join* [Gerber 1986], is found to be similar in performance. This algorithm is exactly the same as the foregoing simple hash-partitioned algorithm except that the temporary relations are not generated. Instead, the two input relations are scanned in their entirety each time a new partition is processed. This method has the disadvantage that all tuples will be read as many times as the number of partitions being generated. It has been found that the overhead associated with multiple scans of the relation is comparable to the overhead of creating temporary relations in each iteration of the algorithm.

Performance

According to the results of a performance evaluation study reported in Gerber [1986], the amount of memory available has a significant effect on the performance of this algorithm. Unless the memory is about 1.2 times the size of the smaller relation, the algorithm is found to perform worse than other join algorithms.

Applicability

The I/O overhead associated with this process is so high as to preclude its use for any but the smallest relations. However, partitioning may not be necessary for small relations, and the simple hash join method may be found to be more useful [Nakayama et al. 1988]. Simple hash-partitioned joins may be good to support pipelined processing.

GRACE Hash Join Method. The *GRACE hash join* method relies on the fact that the dynamic clustering feature of hashing allows joins to be performed efficiently [Kitsuregawa et al. 1983]. It can be used on both single processor and multiprocessor machines and is divided into a partitioning and a matching phase. During the partitioning phase, the two relations are split into an equal number of sets. The sets are disjoint within a relation. For each set of tuples from one relation there is a corresponding set of tuples in the other relation. These sets reside on different machines or different disk files. The matching operation is performed separately for each partition. A tuple in one partition need not be compared with a tuple in any other partition. This reduces the join load considerably. Results from all matching steps are merged together to form the output relation.

Algorithm

In the algorithm presented here, the tuples in related buckets are joined by a simple hash join. The algorithm for performing $R \bowtie_{r(a) \theta s(b)} S$ is as follows:

```
for each tuple r do
    { hash the join attributes r(a)
    place tuple in appropriate output buffer
    Ri};
flush all output buffers to disk;

for each tuple s do
    { hash the join attributes s(b)
    place tuple in appropriate output buffer
    Si};
flush all output buffers to disk;

for i = 1,2,..., N do
    { for Ri do
        { build a hash table for tuples
        read hash table for Ri into memory};
    for Si do
        { for each tuple in Si do
            hash the join attributes s(b)
            if match found in Ri then
            concatenate the two tuples
            place in output relation Q}};
```

Discussion

In this algorithm the partitioning and joining phases are completely disjoint. There are N partitions created for each relation. Flushing of buffers to disk is shown to be performed only at the end of

the partitioning phase. As a buffer for a partition is filled, however, it is actually written to the corresponding disk file for that partition. At the end of the partitioning phase, if any additional memory is available, some of the partitions may be split into smaller partitions. The advantage of this is discussed in the section on handling partition overflow.

Performance

The GRACE hash join method takes $O((n + m)/K)$ time to perform joins where the relations R and S are stored in K memory banks and $(2 \times K)$ processors are used [Kitsuregawa 1983]. A multiprocessor system is not essential, even though it is desirable, for this algorithm to be applicable. The performance of the GRACE hash join method is not affected much by the amount of memory available [Gerber 1986]. Also, memory use is optimal during the join phase as compared to the partitioning phase.

The overall gain in performance depends on the collision factor of the hash function, on how effectively the hash function randomizes the attribute values, and on the size of individual buckets with respect to the available main memory.

Applicability

The load reduction feature (tuples in one bucket do not have to be compared to tuples in other buckets) makes it useful in the case of large databases. Disjoint partitioning makes it ideal for implementation on multiprocessors.

Hybrid Hash Join Method. The *hybrid hash join* has been described in DeWitt and Gerber [1984] and Shapiro [1986]. During the partitioning phase, the hybrid hash algorithm creates a number of partitions such that the hash table for any partition would fit into the available main memory. Instead of writing out each partition as it is created, however, the hybrid hash join creates and keeps one partition in memory while writing out all others. Thus, at the end of the partitioning phase, the hash table for one of the partitions is in main memory and the remaining partitions reside on the disk.

Algorithm

The algorithm for performing

$$R \bowtie_{r(a)\theta s(b)} S$$

is as follows:

```
for each tuple r do
    { hash the join attributes r(a);
    if hash value lies range for partition R1
    then
          insert entry in hash table
          else place tuple in appropriate buffer
          Ri };
flush all buffers except R1 to disk;

for each tuple s do
    { hash the join attributes s(b);
    if hash value lies in range for partition S1
    then
          initiate join process with tuples in
          R1
    else place tuple in appropriate buffer Si};
flush all buffers to disk;

for i = 2, . . . , n do
    { read tuples in Ri into memory
    build a hash table for Ri};
    for each Si do
          { for each tuple in Si do
                hash the join attributes s(b);
                if match found in Ri then
                      concatenate the two tuples
                      place in output relation Q};
```

Discussion

This algorithm is similar to the GRACE hash join in that the partitioning and joining phases are, for the most part, disjoint. The difference lies in the fact that the hybrid hash algorithm does not write out all partitions to the disk. It starts the join process on the first pair of partitions while the second relation is being partitioned. This minimizes the I/O activity to the extent of not having to write the first partition to the disk and then read it back into main memory once the partitioning phase is complete. This is particularly advantageous in the case

of systems with large main memories where partitions may be quite large.

Performance

This algorithm has been found to perform at least as well as the GRACE hash join method that it resembles [Gerber 1986]. It optimizes the use of main memory even during the partitioning phases by initiating the join process if additional memory is available. Thus, if additional memory is available, it outperforms the GRACE algorithm. In situations where the amount of main memory available changes dynamically, an adaptive hash-join algorithm is more efficient [Zeller and Gray 1990]. This has been implemented as a prototype in Tandem's NonStop SQL. It is limited to the case of equijoins since nonequijoins require the correct ordering of tuples.

Applicability

The performance characteristics indicate that this algorithm may be chosen over other hash-partitioned algorithms if significant amounts of main memory are available.

Hashed Loops Join. The *hashed loop join* is a variation of the nested-loops join technique [Gerber 1986]. In the first phase, the outer relation is divided into a fixed number of parts. Next, each partition is read into memory one at a time. A hash table is constructed for each partition when it is read into main memory. Next, the tuples in the inner relation are read. The join attributes are hashed, and the hash values are used to probe the hash table for the current partition for matching tuples. Thus, the inner relation is read once for each partition, and the hash table is used to speed up the process of finding a match.

Algorithm

The algorithm for performing

$$R \bowtie_{r(a)\theta s(b)} S$$

is as follows:

```
partition R using a hash split function;
for each partition Ri do
    { read partition into memory;
    create hash table for partition;
for each tuple s do
    { hash attribute value;
        if s hashes to nonempty entry of hash
        table for Ri then
        initiate join of s and r;
    place result in Q}};
```

Discussion

A simpler version of the above algorithm avoids the overhead of partitioning by staging the outer relation R in phases [Nakayama et al. 1988]. In each phase, as much of R is staged as the size of the available main memory allows.

Performance

Even though this algorithm is based on the slow nested-loops join method, it is found to perform better than other algorithms for a large range of available memory [Gerber 1986]. The reason for this is that the presence of a hash table makes probing for matches fast. Since this method does not involve writing out and reading in of partitions, it incurs less I/O overhead than even the hybrid hash join method [Gerber 1986]. Furthermore, the inner relation is read once for each partition of the outer relation rather than once for each tuple or each block of the outer relation as is the case in the tuple- and block-oriented implementations of the nested-loops join method.

Applicability

This method is found to be especially useful when partition overflow is a problem [Gerber 1986].

2.4 Summary

Various join techniques have been described in this section. Although there may be other methods, this is a representative selection. Most of these methods have been studied in detail, and the

relative performance in various kinds of computing environments has been compared [Bitton et al. 1983; Gerber 1986; Schneider and DeWitt 1989].

3. JOINS BASED ON SPECIAL DATA STRUCTURES

It has been found that some inefficiencies in the join operation can be overcome by means of specially tailored data structures. Some of these have been in the form of efficient index structures; others have been designed specifically to support the join operation. The former speed up joins by facilitating fast access to relations; for example, T-trees [Lehman and Carey 1986], data space partitioning [Ozkarahan and Bozsahin 1988], kd-trees [Kitsuregawa et al. 1989b], Bc-trees [Goyal et al. 1988]. The latter reduce the response time of queries involving joins by maintaining lists of matching tuples for commonly joined relations. This information may be stored explicitly as in the case of join indexes [Valduriez 1987] or implicitly as in the case of the data-partitioning scheme in Ozkarahan and Bozsahin [1988]. Some such data structures are discussed below.

3.1 Join Indexes

Data structure

A *join index* is a relation with arity two [Valduriez 1987]. It identifies each result tuple of a join by pointing to the participating tuples from each input relation. Each pointer may be in the form of a key value, physical address, or any other logical value, *surrogate*, which uniquely identifies a tuple in a relation. It is a prejoined relation created by joining the relations R and S and projecting the result on the surrogates of the joined tuples. Thus, each tuple in the relations being joined must have a unique surrogate. Then a join index for two relations is a binary relation that contains pairs of surrogates. An example of a join index for the two example relations R and S is shown in Figure 2. Here we have added tuple-ids as the surrogates for each relation. The relation may be clustered on the surrogates for tuples of either or both relations for fast access. The join algorithm uses the join index to locate matching tuples from the two input relations.

Algorithm

Executing a join using the join index consists of the following steps [Valduriez 1987]:

Scan join index to find matching tuples x;
Retrieve matching tuples x;
Concatenate tuples and place them in result relation Q x;

Discussion

Since a join index is itself a relation, it will incur all the overheads associated with the creation and maintenance of a relation. In Valduriez [1987] it is shown that the maintenance cost of join indexes is marginal in the case of foreign key joins (most joins) because it is included in referential integrity checking included in the process. Furthermore, the join index must be consistent with any changes made in the participant relations. If join indexes are maintained for several relation pairs in the database, the amount of storage and housekeeping overhead can be considerable [Desai 1989]. If the selectivity factor is high, the number of entries in the join index could approach the size of the product of the cardinality of the input relations. This expense must be justified by frequent joins of the relations involved.

Performance

The major cost in this method is the cost of retrieving matching tuples. (The other cost component, that of comparing tuples to find matching pairs, is incurred only at the time the join index is created.) Therefore, the overall performance of the algorithm is affected by the clustering properties of the two relations. The best results are obtained when both relations

Relation R

R-id	employee	payscale
1	james	1
2	jones	2
3	johns	1
4	smith	2

Relation S

S-id	payscale	pay
1	1	10000
2	2	10000
3	3	10000

Join index

R-id	S-id
1	1
2	2
3	1
4	2

Figure 2. Join index.

are clustered on the join attributes. Join indexes also give good results in the context of complex queries involving many joins and selects.

Applicability

Since surrogates are stored, selects and projects may also be performed using the join indexes [Valduriez 1987]. Join indexes also provide an easy means of implementing semijoins. With the help of a join index, the semijoin operation $R \ltimes S$ simply consists of retrieving the tuples of R listed in the join index. Query-processing techniques based on join indexes are presented in Valduriez [1986]. Processing of recursive queries using join indexes is discussed in Valduriez and Boral [1986]. A natural extension of join indexes is for complex object support through hierarchical join indexes [Valduriez et al. 1986].

3.2 Bc-Trees

Data Structure

The *Bc-tree* (composite B-tree) is a special index structure which facilitates select and join processing [Desai 1989; Goyal 1988]. The Bc-tree is similar to a B^+-tree except that it indexes a given attribute in more than one relation. That is, it points to tuples with a given attribute value in several relations. Figure 3 shows the basic format of a Bc-tree node assuming m relations are indexed. It contains the key value and a list of surrogates identifying the tuples from each of the m relations with the associated key value. Here relation 1 is shown to have n such tuples, and relation m is shown to have k. This node list contains all the information needed to perform a natural join. An example of a Bc-tree for the example relations R and S is shown in Figure 4. The index is created on the payscale attribute, and the tuple-ids shown in Figure 2 are used as the surrogates. A λ is used to indicate a null pointer or surrogate value.

Algorithm

The algorithm for performing the equijoin $R \bowtie_{r(a)=s(b)} S$ is as follows:

```
for every tuple s in S do
    { find node in Bc-tree for key value s(b);
    if a tuple r in R is found in the node list
        then { concatenate r and s
            place result tuple in Q }};
```

Discussion

The join process is expedited because the tuple-ids of the tuples participating in the join are isolated in the Bc-tree nodes. The page numbers of the pages participating in the join are determined from

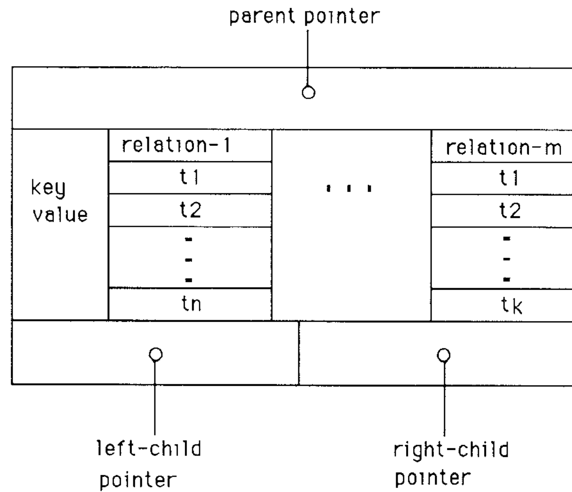

Figure 3. Bc-tree node.

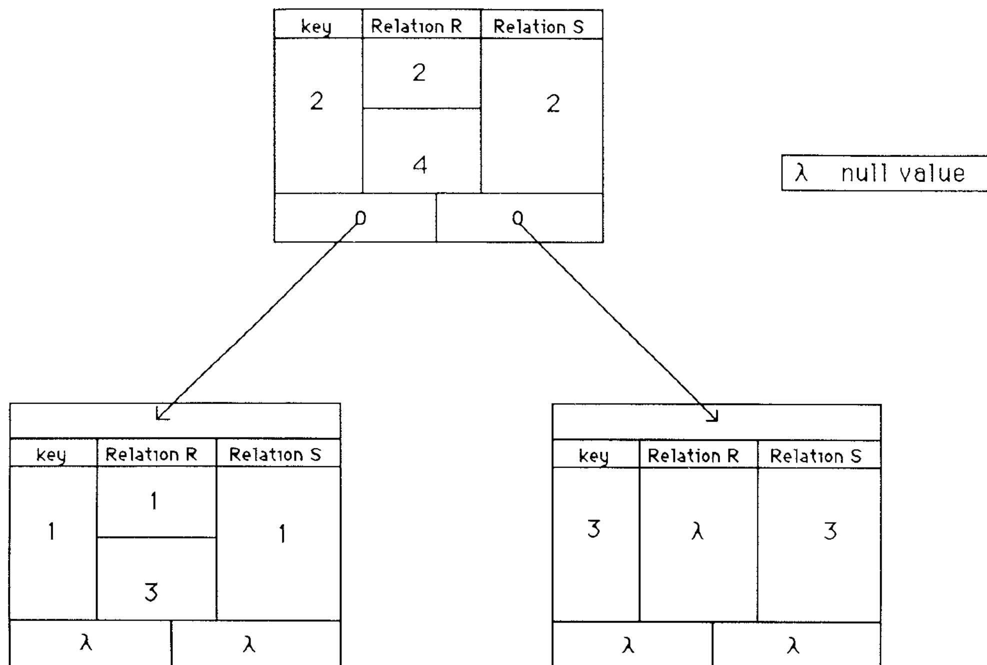

Figure 4. Bc-tree on payscale for relation R and S.

the tuple-id. Information on page pairs participating in the join is stored in a matrix, which is used to decide the order in which the pages are retrieved from the disk. Once a pair of pages has been retrieved, the tuple-id lists are used to read the tuples and join them. The major advantages that have been projected are that the number of I/Os can be minimized and that buffer use can be optimized by knowing which page pairs participate in the join in advance.

Performance

The major cost in this algorithm is that of searching the Bc-tree. The number of searches is least when the index structure is searched for the values of the join attributes in the smaller relation.

Applicability

This method can only be used when an index on the join attributes exists in the form of a Bc-tree. Although the equijoin is the simplest to implement using this index structure, the basic algorithm can be extended to cover nonequijoins. This can be done by means of appropriate Bc-tree traversal and search of all nodes along the path for matching tuples.

3.3 T-Trees

Data Structure

The T-trees index structure was proposed in Lehman and Carey [1986]. It has been found to have special use in the context of main memory processing. A *T-tree* is a binary tree with several elements per node. It can be described as a combination of the AVL-tree and the B-tree data structures. Its search method is similar to that of AVL-trees, and its nodes are similar in structure to B-tree nodes. Thus, it has the fast search characteristic of AVL-trees and the good storage properties of the B-tree. T-trees can have three types of nodes: internal nodes, half-leaf nodes, and leaf nodes. The structure of an internal node is shown in Figure 5. This node contains n key values and would also have surrogates indicating the tuples from the relation with each of those values. Unlike the Bc-tree, the T-tree is constructed for only one relation. Each interior node of the tree, called the T-node, can have at most two children. The left subtree is searched for key values less than key_1, and the right subtree is searched for key values greater than key_n.

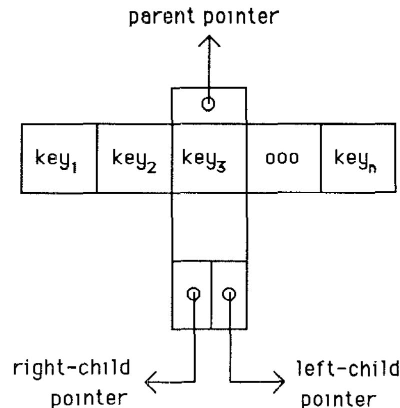

Figure 5. T-tree internal node.

Algorithm

The underlying algorithm for performing joins using T-tree index is the nested-loops join method [Lehman and Carey 1986]. The relation with the T-tree index is chosen to be the inner relation. For each attribute value in the outer relation, the T-tree is searched to find the corresponding values in the tree. The surrogates are then used to find the appropriate tuples from the inner relation. The gain in efficiency is due to the fact that the T-tree index permits fast location of matching tuples from the inner relation.

Discussion

A T-tree can be made to accommodate duplicates easily [Lehman and Carey 1986]. This property makes it useful for processing nonkey and secondary key attributes.

Applicability

This algorithm is useful only if a T-tree index exists on one of the input relations. If an index were to be constructed at the time of the join, the construction time would mitigate any gains resulting from the fast search of the tree.

3.4 Kd-Trees

Data Structure

The kd-trees data structure was first proposed and discussed in Bentley [1975]. Application of this data structure to the case of searching in databases is discussed in Bentley [1979] and Bentley and Kung [1979]. Many derivatives have since appeared in the literature; for example, extended kd-tree [Chang and Fu 1980], kdb-tree [Robinson 1981], and gkd-tree [Fushimi et al. 1985].

A *kd-tree* is a multidimensional binary search tree, where k is the dimensionality of the search space. It can be used to index a relation that has k distinct keys $a1, a2, \ldots, ak$. Each node in the tree consists of a record, a left pointer, and a right pointer. The index tree is created as follows. The relation is first divided into two equal-sized parts based on the first key $a1$. Both parts have the same number of tuples. The values of attribute $a1$ in the tuples in the first part are less than values in the tuples in the second part. Leaf nodes in the left subtree of the root point to the first part, whereas those in the right point to the second part. At the root level, the attribute $a1$ is known as the discriminator. At the next level, the discriminator attribute is $a2$. Each of the two parts above is divided into two parts, each based on the values of $a2$. This process is performed recursively, with the appropriate discriminator at each step, until the partitions are small enough to fit on one page. Leaf nodes point to pages that satisfy all of the discriminator values above it.

All nodes at the same level in the tree have the same discriminator. For a k-dimensional index, the first k levels have the discriminators $a1, a2, \ldots, ak$, respectively. At lower levels in the tree, the cycle repeats with discriminator at the $(k + 1)$th level being $a1$.

This data structure has been found to be an effective way of achieving multidimensional clustering. It is particularly useful in systems that allow queries to specify search conditions in terms of several keys. In this respect, a kd-tree may be considered as a replacement for features like inverted files and superimposed coding systems. The advantages that kd-trees claim over previous attempts to address the same problem are logarithmic insertion, search and deletion times, and the ability to handle many different types of queries, such as range, partial match, nearest neighbor, and intersection queries.

An algorithm based on the use of the kd-tree is a partition-based algorithm, where the range of attribute values in and the size of partitions is not fixed or predefined. The partitioning of relations is based on the number of tuples, not on the ranges of attribute values. It is highly unlikely that both relations would have the same overall range of attribute values and also the same distribution of attribute for each discriminator. As a result, partitions of two relations do not have a unique mapping. In most cases, partition pairs will have overlapping ranges of attribute values. In order to distinguish these partitions from the fixed-range partitions discussed before, the partitions in this case are called waves.

A *wave* is a set of pages from each relation participating in any given stage of the join operation. Each join step is characterized by a join range. The tuples in the pages making up a wave must include attribute values in this range. Each page in a wave is characterized by the upper and lower limits of the join attribute values contained in a page. Since the range of join attribute values contained in a page depends on the distribution of the attribute values, upper and lower limits of the join attribute values of all pages in a wave are usually not the same. This leads to the fact that a wave does not have a well-defined range of attribute values. The attribute range in each page in a wave will fall within the join range or overlap it partially. For example, the wave shown in Figure 6a consists of three pages, each with a different range of join attribute values. It must be remembered that each page con-

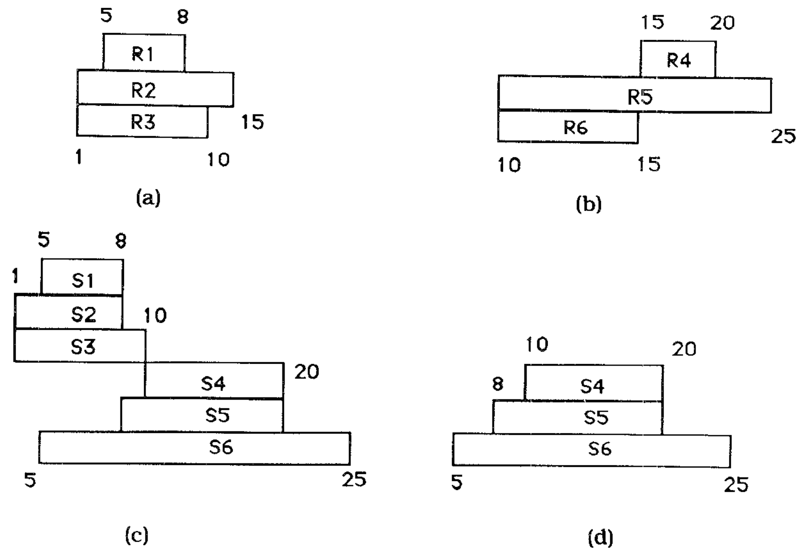

Figure 6. Kd-tree join.

tains the same number of tuples, and the range of attribute values is different as a result of the nonuniform distribution of attribute values in relations.

Algorithm

Any algorithm based on the concept of waves must define how waves are determined. Consider the relations R and S, with six pages each, shown in Figure 6. The basic steps in one wave-based algorithm are

```
repeat
    load wave from R;
    determine join range Jr;
    load corresponding wave from S;
    join the two waves;
    until R has been exhausted;
```

Then, the relations in Figure 6 could be joined in the following steps:

```
load R-wave = [R1, R2, R3];
let join range = 1 – 15;
load S-wave = [S1, S2, S3, S4, S5, S6];
join R-wave and S-wave;
load R-wave = [R4, R5, R6];
let join range = 10 – 25;
load S-wave = [S4, S5, S6];
join R-wave and S-wave;
```

In terms of Figure 6, the first wave from relation R (shown in Figure 6a) joins with the first wave in relation S (shown in Figure 6c). The second wave in relation R (shown in Figure 6b) joins with the second wave in relation S (shown in Figure 6d). The join range is determined according to the Jrout definition (detailed below). This example is meant only to illustrate the general working of the Kd-tree join method; it is not the most optimal way of joining R and S.

Discussion

Factors that determine the exact form of the algorithm are as follows:

(1) *Wave size* can vary from a minimum of one page to a maximum of $(M - 1)$ pages, where M is the size of main memory available [Kitsuregawa et al. 1989b].

(2) The *join range* is defined in terms of the maximum and minimum values of the join attributes contained in the pages of a wave. Two possible definitions are as follows [Kitsuregawa et al. 1989b]:

(a) **Jrin**. The lower limit of the range is the largest of the lower limits of all the pages in the wave. The upper

limit of the join range is the smallest of the upper limits of the pages in the wave. A consequence of this definition is that many tuples in the wave have join attribute values that fall outside the join range.

(b) **Jrout**. The lower limit of the join range is the smallest of the lower limits of the individual pages. The upper limit the join range is the largest of the upper limits of the pages in the wave. As a result, many tuples in the relation that would fall in the join range are not part of the wave.

In the former case, waves of consecutive join steps may have some pages in common. In the latter case, there is no overlap in the pages of waves of consecutive join steps. A number of algorithms based on different combinations of the above factors are described in Kitsuregawa et al. [1989b].

Performance

An algorithm between kd-tree indexed relations that does not take advantage of the wave concept will have all the overhead of the nested-loops method [Kitsuregawa et al. 1989b]. The wave concept offers the benefits of partitioning to a limited extent. Instead of comparing every tuple in one relation to every tuple in the other relation, tuples in wave pairs are compared. This is not as efficient as comparing tuples in disjoint partition-pairs, such as in the GRACE hash join method, but it is still better than the exhaustive comparisons of the nested-loops method.

Factors that affect the performance of the wave-based join operation on kd-tree indexed relations are as follows:

(1) Wave size.
(2) Join range, which may be determined from the point of view of either one or both relations.
(3) Wave propagation speed, defined as the inverse of the number of join steps. The number of join steps is determined by the wave size and how the boundaries of a wave are determined.

Other join algorithms based on this data structure are presented and evaluated in Harada et al. [1990].

Applicability

This method provides an efficient way of joining two relations if both have a kd-tree index on the join attributes. Its real significance, however, lies in the fact that it is general enough to be applied to any pair of relations that have clustered indexes on the join attributes.

3.5 Prejoins

Data Structure

The data structure on which this join method is based is the *predicate tree* [Valduriez and Viemont 1984], which is a multidimensional access method. The use of predicate trees for performing joins and selects is discussed in Cheiney et al. [1986]. A predicate tree is a balanced tree index structure associated with a relation. The tuples in the relation are divided into classes based on the predicates defined for the tree. At each level in the tree, the predicate pertaining to one specific attribute is used; for example, refer to Figure 7. This figure shows a predicate tree with two levels. The first level has predicates based on name; the second has predicates based on payscale. The number of levels in the tree is equal to the number of attributes on which the predicates are defined. The tuples in any subtree satisfy the predicates defined at each node on the path from the root of the predicate tree to the root of the subtree. The leaf nodes of a predicate tree contain pointers to the actual database where the tuples with the corresponding predicate values exist. In Figure 7, the database is, in effect, partitioned into four parts based on the predicate values indicated by the four leaf nodes.

Algorithm

The join space for each relation is divided into disjoint partitions based on the pred-

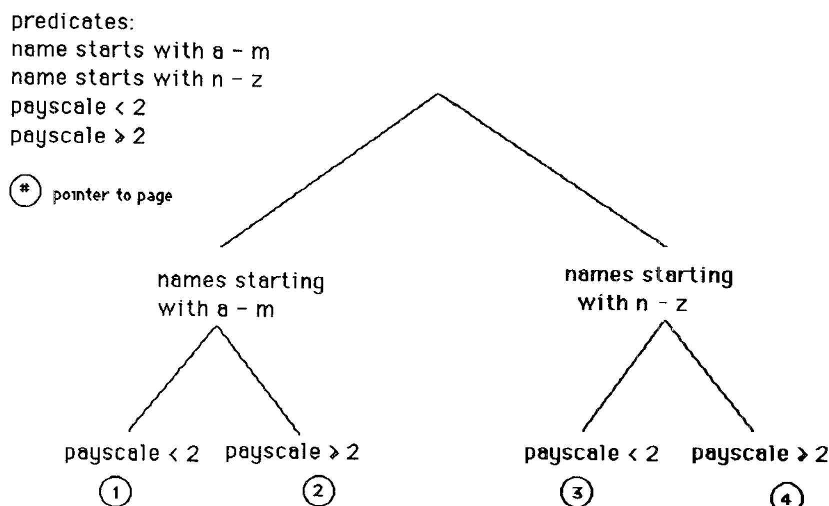

Figure 7. Predicate tree. Predicates: Name starts with a-m, name starts with n-z, payscale < 2, payscale ≥ 2. (#), pointer to page.

icates specified for the relation. As a result, the total join consists of the union of the joins of the individual partition pairs. The above holds true if it is assumed that both input relations have the same predicates. If not, the join may have to be performed in a manner similar to the kd-tree join.

Discussion

This algorithm takes advantage of multiattribute clustering. Queries containing the predicates on which the tree is based execute faster and generate fewer I/O requests. A query-processing method that considers multiattribute clustering is discussed in Harada et al. [1990].

Performance

The performance of joins on relations indexed by predicate trees depends on two factors: (1) the join method used to join partition pairs and (2) whether the same predicates are used to create the predicate trees for both relations. If the same predicates are used, the efficiency is as high as any other partition-based join method. If not, the efficiency is the same as the kd-tree join method described earlier.

Applicability

This method is limited to the cases where the join attributes are used to define the predicates associated with a relation's predicate tree. Although equijoins are the simplest to perform, with appropriate tree traversal, the predicate tree can also be used for nonequijoins.

3.6 Summary

Join methods in this section have been described in the context of specific data structures. Some of them facilitate join execution by providing fast access to relations. Others, such as the Kd-tree join method, are general enough to apply to cases where overlapping partitioning of relations exists. The join index method and the Bc-tree method are examples of precomputed joins that are most useful in situations where the relations are joined often and rarely updated.

4. JOIN CLASSIFICATION

Each join algorithm performs the following three major functions:

(1) Partition
(2) Match
(3) Merge

For discussion purposes, suppose we wish to perform $R \bowtie S$, where R and S have n and m number of tuples, respectively. As we have seen, in algorithms with no partitioning, such as, nested loops, all $(n \times m)$ tuple pairs must be examined to calculate the resulting set. This is the worst-case scenario. The purpose of the partition step is to reduce the number of pairs of tuples to be examined; for example, the hash-partitioning approaches use a hashing step, and the sort-merge method uses sorting. Partitioning may, however, increase the overhead associated with the execution of a join. Partitioning is usually done on each input relation separately. The match step is where tuples in each of the partitions are matched. Finally, in the merge phase, the matched tuples are combined together to create the result relation. The merge step may simply consist of concatenation of tuples as in the sort-merge method. Or it may involve staging of tuples, for example, in hash joins where the tuples themselves are not stored in the hash tables.

In this section we concentrate on the partitioning phase and describe a classification scheme based on it. The type of partitioning is used to differentiate between the various join methods. In the following sections, we further investigate the partitioning phase of join processing and then introduce the final categorization scheme.

4.1 Partitioning

The various kinds of partitioning seen in join algorithms are as follows:

(1) None. No partitioning is performed, and the input relations must be exhaustively compared in order to find the tuple pairs that participate in the join.
(2) Pre. Partitioning is not performed as part of the actual join algorithm. These techniques assume that some partitioning exists.
(3) Implicit. Although the join algorithm does not have a step aimed specifically at performing the partitioning, it does do some dividing or ordering of the data so fewer tuples need to be compared in the match step.
(4) Explicit. The join algorithm contains an explicit partitioning phase as part of its execution.

In all cases of explicit partitioning, each tuple is uniquely assigned to only one partition. That is, partitions within a relation are disjoint. In addition to the kind of partitioning phase, another important factor is the mapping between the partitions of the two input relations. The mapping can be best described in terms of the join attribute range of partitions, which is the range of join attribute values that can be assumed by the tuples falling in the partition.

Suppose that R and S have been partitioned in such a way that tuples in a given partition R_i can join tuples in exactly one partition S_i. In other words, R_i and S_i have the same join attribute range. In this case, there is a one-to-one mapping between partitions of R and S. Such a situation is usually a result of explicit partitioning. However, it is more usual to have relations partitioned such that the join ranges do not coincide; instead, they overlap to a greater or lesser extent. This is called the *degree of overlap* between partitions R_i and S_i.

Join algorithms can also be differentiated on the basis of the degree of overlap between the partitions of the two input relations. Some possible degrees of overlap are described below and shown pictorially in Figure 8.

Complete. If no partitioning is performed, there is no question of join ranges. There is a complete overlap, and all tuples must be compared. These cases have the highest join load in that all tuple pairs must be compared. In Figure 8a all tuples of R are shown to be compared to all tuples from S. For example, in performing a nested-loops join, the james employee tuple in relation R would be compared to each of the three tuples in relation S. Likewise, each of the other

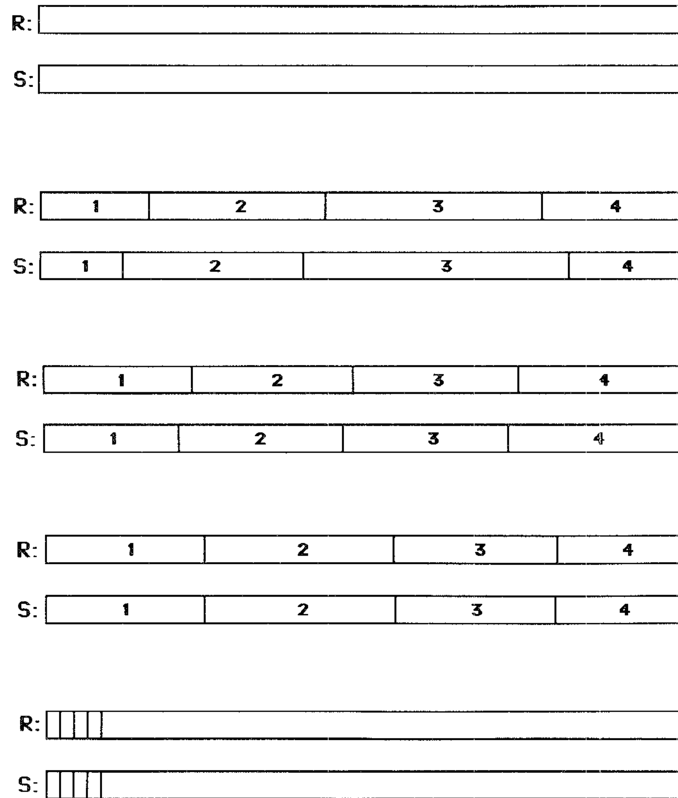

Figure 8. Degrees of overlap. (a) None: only one partition in each relation; (b) variable overlap; (c) minimum overlap; (d) disjoint; (e) complete.

tuples in R would be compared to all three tuples in S.

Variable overlap. There may be a considerable amount of overlap between different partitions. However, the amount of overlap varies between different partitions. In Figure 8b tuples in the first partition of R are compared to tuples in the first and second partitions of S. Tuples from the second partition of R are compared to tuples in the second and third partitions of S. Tuples in the third partition of R are compared only to tuples in the third partition of S. Tuples in the fourth partition of R are compared to tuples in the third and fourth partitions of S. The amount of overlap between join ranges may be anywhere from no overlap to complete overlap. Since the amount of overlap cannot be predicted, it is not possible to predict the reduction in join load accurately. The join process involves fixing the join range at each step and finding the partitions from both relations that include all tuples that fall in the join range. Due to the variable nature of waves when using Kd-tree indexes, the wave-based algorithms have variable overlapping partitions.

Minimum overlap. With these techniques the join ranges may overlap. The degree of overlap is, however, minimum. There may be at most one attribute value in common between two partitions (see Figure 8c). Such partitioning results in a high degree of reduction in the join load. The sort-merge algorithm has a minimum overlap.

Table 1. Partitioning Description

Algorithm	Partitioning Type	Degree of Overlap
Sort-merge	Implicit	Minimum overlap
Nested loops	None	Complete
GRACE hash join	Explicit	Disjoint
Hybrid hash join	Explicit	Disjoint
Simple hash partitioning	Explicit	Disjoint
Hash loops	None	Complete
Kd-tree index	Pre	Variable overlap
Join index	Pre	None
Simple Hash	Implicit	Variable overlap
Bc-tree Index	Pre	None
Prejoin	Pre	Variable overlap

Disjoint. Both relations have the same join ranges for all partition pairs. These join ranges are fixed before partitioning the relations explicitly. Thus, tuples in a given partition R_i can join only with tuples in the corresponding S_i (see Figure 8d). That is, each partition of R maps to one and only one partition of S. Partition pair joins can be carried out in isolation and be independent of each other. The result of joining the two relations is the union of the results of joining the partition pairs. Disjoint partitioning therefore provides a high and predictable amount of reduction of the join load. Hash-partitioning algorithms are disjoint since the partitions are disjoint and tuples from only one partition of each relation are compared.

None. Each tuple of one relation in the partition joins with each tuple of the other relation in the partition. In other words, each join range consists of exactly one attribute value (Figure 8e). There is no overlap between partitions of the two relations. Partitions that do not overlap are seen in precomputed joins using the join index and Bc-tree index. Join methods relying on no overlap have the smallest join load. Note that this saving is achieved at the expense of creating the index beforehand and storing the precomputed join.

In the above classification, it may appear that some classes are merely special cases of others. For instance, the minimum overlap class appears to be a special case of the variable overlap class. It is, however, necessary to make the distinction between the two classes because there are situations when the partitioning is consistently of the minimum overlap variety. On the other hand, the variable overlap class encompasses all algorithms where it is never possible to predict the degree of overlap.

In Table 1 the basic join algorithms are described using these partitioning descriptions. The nested-loop algorithms perform no partitioning but have complete overlap. Thus all $(n \times m)$ tuple pairs must be examined for any such join. Techniques that rely upon special index data structures have no partitioning within the algorithms. These algorithms simply do the matching by appropriate examination of the partitions created by the index. The relations must be indexed on the join attribute, and the index must be clustered. Most algorithms, however, perform partitioning within the join algorithm itself. Thus, whenever a join is performed, the overhead of partitioning is incurred. We can differentiate between these algorithms based on whether the partitioning is implicit or explicit. Implicit algorithms, although reducing the number of tuple pairs to be examined, do not explicitly perform a partition. For example, the sort-merge algorithm reduces the number of tuples to be examined but does not put tuples into partitions. The partition-

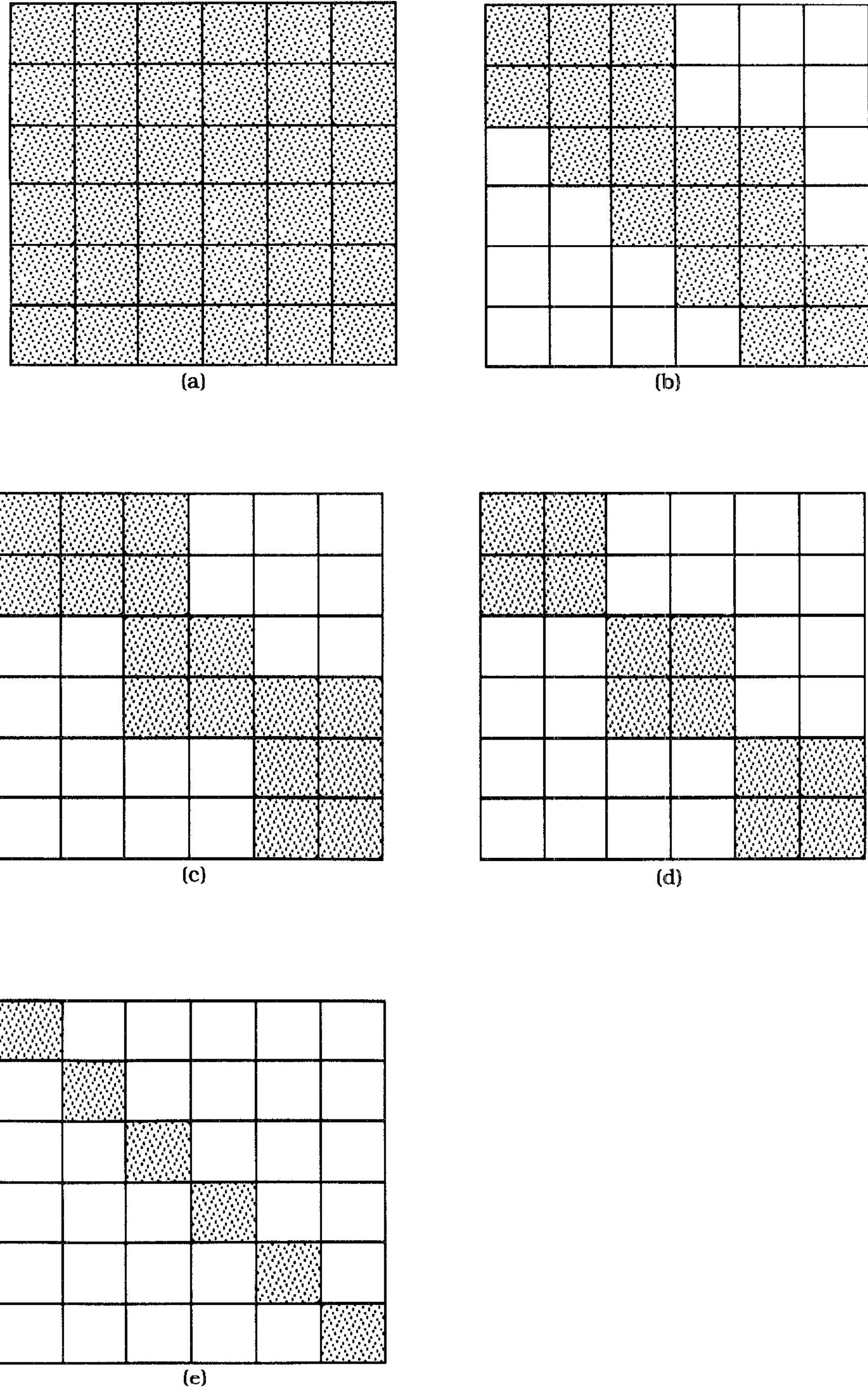

Figure 9. Join load. (a) No partitioning; (b) variable overlap; (c) minimum overlap; (d) disjoint partitioning; (e) complete partitioning.

ing is implicit based on the relative order of tuples after the relation has been sorted.

In Figure 9,[1] we pictorially show the five different types of join loads or overlap. As in Figure 1, the horizontal and vertical axes represent the tuples in the two relations. Each small box, then, represents a tuple pair to be examined. (In this figure we assume that both R and S have six tuples each.) The shaded boxes in each figure indicate tuples to be compared; the unshaded portions show tuple pairs that do not need to be examined. Figure 9a shows unpartitioned relations where a complete overlap exists. Thus, the entire 6×6 square is shaded. In this case there is no partitioning and no savings. The algorithms with complete overlap are the loop types (nested and hash). The other partitioning extreme is shown in Figure 9e. Here a complete partitioning is performed with no overlap. The maximum cost saving results because the matching of tuples has been done beforehand, and there is no need for a match step in the join algorithm. As seen in Table 1, the algorithms falling into this class are the predefined join index and the Bc-tree index. Figure 9d shows a disjoint partitioning. Here each tuple of each relation is placed into only one partition (as with the three hashing algorithms). Figures 9b and 9c show the two versions of overlap partitioning, namely, the variable overlap and the minimum overlap. In Figure 9b, the type of overlap provided by using a Kd-tree index is shown. Notice that partitions are generated with no regular structure. The structure shown in Figure 9c is much more precise. Here the overlap between two successive partitions is at most one tuple from one relation. This is thus a minimum overlap. The sort-merge algorithm displays this kind of partitioning. It is possible with the sort-merge algorithm that two successive partitions do not overlap at all. Although these figures are not precise in that each algorithm of that type always partitions precisely as shown, they do accurately reflect the overlapping between the partitions and the savings the partitioning provides.

[1] This pictorial representation was inspired by similar figures used to explain GRACE hash joins [Kitsuregawa et al. 1983].

4.2 Classification

Figure 10 shows our classification of join algorithms based on this partitioning description. The first level in the tree indicates the type of partitioning; the second shows the degree of overlap. We have indicated the class in which each of the algorithms surveyed earlier fall. The classification is fairly general, and it is expected that any and every join algorithm will have a definite place in the classification.

5. JOIN PROCESSING IN A DISTRIBUTED ENVIRONMENT

5.1 Factors in Distributed Processing

Most of the algorithms discussed so far have been for centralized computing environments in which the database resides at one site and is accessed by users at that site alone. With simple modifications, many of these algorithms can be extended to the distributed case. In evaluating a join method for application to the distributed environment extra factors must be considered, such as the following:

Data transmission cost versus local transmission cost. The cost of transmitting data between sites depends on the kind of network [Epstein 1982; Mackert and Lohman 1986; Perrizo et al. 1989; Yu et al. 1987]. In wide area networks (WAN), the cost of data transmission overshadows the local processing costs. Query optimizers in databases distributed across such networks try to optimize transmission costs even if it is done at the expense of increased local processing. In fast local area networks, transmission costs are low, and processing costs at the sites can no longer be ignored [Wang and Luk 1988].

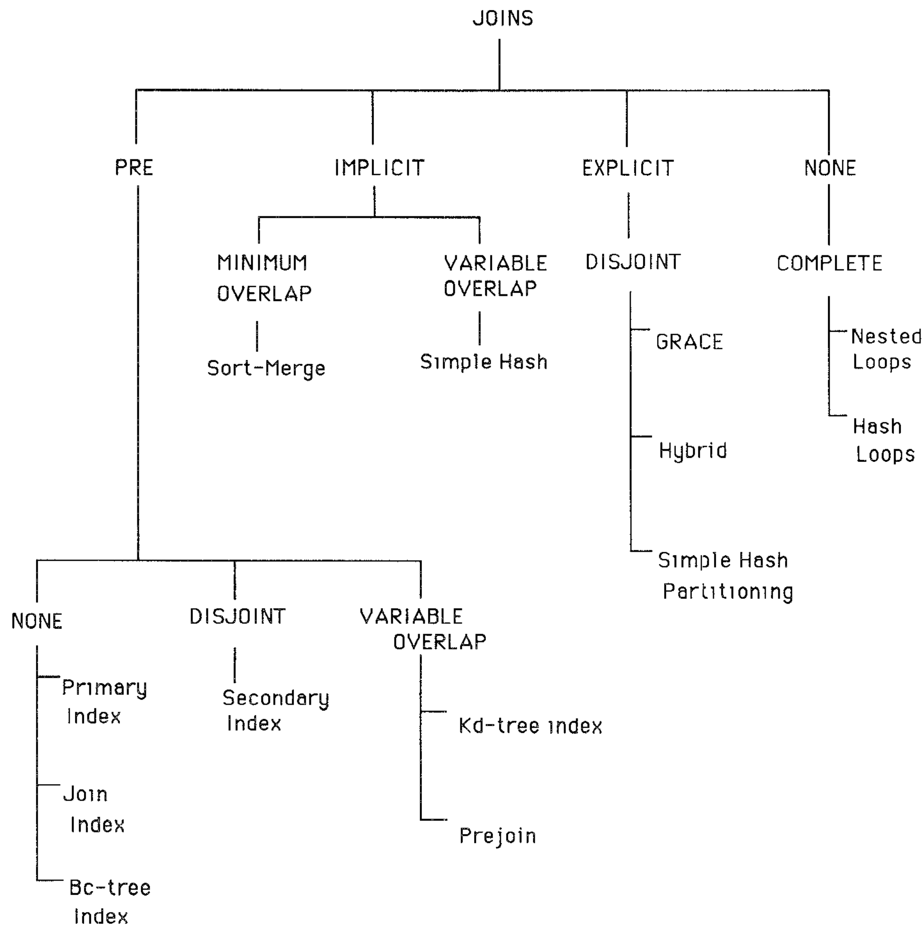

Figure 10. Join classification.

Partitioning. Large relations are often partitioned and distributed across sites. This is usually done to accommodate cases where parts of a relation are accessed most frequently by a single site [Ceri et al. 1986; Chen and Lu 1989; Yu et al. 1985]. The type of partitioning, whether horizontal, vertical, or both, depends on the usage patterns. Horizontal partitioning has a direct effect on the join operation and must be taken into account.

Join site. Choosing sites at which the join is to be executed depends on many factors [Bandyopadhyay and Sengupta 1988; Cornell and Yu 1988]; for example, the site at which the result relation is desired, site of the largest relation or relation fragment resides, and replication. The choice of site is also affected by the data transmission costs. Depending on the partitioning of relations, it may be feasible to perform joins of fragments at different sites. The partial results may then be transmitted to a single site where they can be concatenated to get the final result.

Type of query processing. Operations in query processing may be performed in strict sequence or be pipelined [Lu and Carey 1985; Mikkilineni and Su 1988; Richardson et al. 1987].

Replication. Whether relations or fragments of relations are replicated is

important in that the query optimizer must be aware of the existence of all the replications and whether they are consistent with each other. It must then choose between the copies based on the choice of join site.

In the following section, the techniques used for distributed join execution are discussed in the context of the above-mentioned criteria. The main steps in the algorithms are also given.

5.2 Join Algorithms

In distributed databases, most join algorithms used are built using the algorithms devised for centralized databases. For instance, since a semijoin can be used to identify and transmit tuples from one relation participating in a join to another network node where the other join relation resides, it is often a common starting point for join algorithms in distributed databases.

5.2.1 Semijoin

This implementation, consisting of one semijoin operation followed by a join operation, is particularly suited to the distributed environment since it helps reduce the quantity of data transferred between sites.

Algorithm

If the two relations R and S reside at different sites and the result of a θ-join between them must be made available at the site of relation R, then they may be joined in the following steps:

$R1 := \pi_{r(a)}[R]$;
transfer $R1$ to site of S;
$S1 := R1 \bowtie_{r(a)\theta s(b)} S$;
Transfer $S1$ to site of R;
$Q := S1 \bowtie_{r(a)\theta s(b)} R$;

Discussion

The first three steps in the above algorithm constitute the semijoin $S \ltimes R$. The result relation Q in the last step is the same as the result of $R \ltimes S$. However, to perform $R \ltimes S$ in a single step, one of the two relations would have to be transferred to the other site in its entirety. In the above method, the set of distinct values of the join attributes of R and matching tuples from S are transmitted between sites in the form of the intermediate relation $R1$. This volume of data is not likely to exceed the data in either one of the two relations. This is especially true if the join attributes are not the primary key.

Applicability

Semijoins have been used to great advantage in distributed systems despite the fact that they necessitate multiple scans of relations [Kang and Roussopoulos 1987a, 1987b; Valduriez 1982]. This is especially true in wide area networks where the cost of data transmission is usually higher than local processing costs.

In high-speed, local area networks, however, the data transfer rate between sites approaches the speed of data transfer between main memory and disk [Lu and Carey 1985]. In such cases, local processing costs may overshadow data transfer costs [Mackert and Lohman 1986], and the number of semijoins executed may be more important than the reduction of transmitted data [Masuyama et al. 1987].

In queries that are solved using a series of semijoins, the sequence in which the semijoins are executed can be important [Gouda and Dayal 1981; Masuyama et al. 1987; Yoo and Lafortune 1989].

A semijoin-based technique called *composite semijoining* has been developed for use in cases where multiple join attributes are involved [Perrizo et al. 1989]. If the majority of the attributes in a relation make up the join condition, however, the overhead of transmitting the entire relation may actually be less than that of projecting out the join attributes before transmission. Such a *relation-transmission* technique has been described in Sacco [1984].

5.2.2 Two-Way Semijoins

The *two-way semijoin* operator is an extended version of the semijoin operation [Kang and Roussopoulos 1987a]. Like the semijoin operation, its usefulness lies in its ability to reduce relations. The result of a two-way semijoin between two relations is a set of two relations. Formally, it can be described as

$$R \leftarrow [r(a)\theta s(b)] \rightarrow S = \{R1, S1\},$$

where

$$R1 = R \rtimes_{r(a)\theta s(b)} S,$$

$$S1 = S \rtimes_{r(a)\theta s(b)} R.$$

Algorithm

Let relations R and S reside at sites 1 and 2, respectively. Then the steps in the computation of

$$R \leftarrow [r(a)\theta s(b)] \rightarrow S$$

are as follows:

$R1 = \pi_{r(a)}[R]$;
send $R1$ to site 2;
$S1 = S \rtimes R1$;
$R1_m = R1 \rtimes S$;
$R1_{nm} = R1 - R1_m$;
if $R1_m < R1_{nm}$ then
  send $R1_m$ to site 1;
  $R11 = R \rtimes R1_m$;
  else send $R1_{nm}$ to site 1;
    $R11 = R \rtimes R1_{nm}$;
  send $S1$ to site 1;
  $R \bowtie S = R11 \bowtie S1$;

Note the following:

(1) The intermediate relations $R1_m$ and $R1_{nm}$ are created along with $S1$; the tuples of $R1$ that participate in the semijoin are placed in $R1_m$, whereas those with no match in S are placed in $R1_{nm}$.
(2) The semijoin conditions in the creation of $R11$ using $R1_{nm}$ is the opposite of the condition on all other semijoin and join operations in the algorithm. The join condition on all other semijoins and joins is $r(a)\theta s(b)$.
(3) The relation $R11$ is created to reduce the size of $R1$ to the minimum needed to perform the join.

Performance

The performance of a two-way semijoin is evaluated on the basis of the amount of reduction in the relations on which it operates. Simulation results indicate that it is more powerful than ordinary semijoins [Kang and Roussopoulos 1987a]. The cost of executing two-way semijoins is comparable to executing two back-to-back semijoins. It will, however, always be less since the relation fragment sent back to the first site is the smallest possible.

Applicability

Most query-processing algorithms based on semijoins can be modified to use two-way semijoins instead. Simulation experiments indicate that the response time improves significantly [Kang and Roussopoulos 1987a].

5.2.3 Bloomjoin

A *bloomjoin* uses bloom filters to filter out tuples that do not participate in a join [Mackert and Lohman 1986]. We consider it a special implementation approach for a semijoin rather than a new join algorithm. A *bloom filter* is a large vector of bits that are initially set to 0 [Bloom 1970]. For a given relation, a bloom filter is created in the following steps:

(1) Apply a hash function to the join attributes. The resulting hash value points to a bit in the bit array.
(2) Set the bit pointed to by the hash value to 1.

Once the bloom filter has been created for a relation, it can be used to determine whether a given attribute value is present in the relation. In the case of the join operation, a bloom filter is created for one of the relations. Next, the join attributes from the second relation are hashed. If the hash value points to a bit set to 1 in the bloom filter, the corresponding tuple is likely to have a match in the first relation.

Algorithm

Assume that relations R and S reside at sites 1 and 2, respectively. The major steps in the process of joining R and S are

```
generate bloom filter for S;
send filter to site 1;
for each r do
    hash join attributes;
    if hash value addresses a bit set to 1
        then put r in R1
        else discard s;
send R1 to site 2;
join R1 and S;
```

Note that the same hash function must be used on both relations.

Discussion

Bloomjoins have also been called hashed semijoins [Mackert and Lohman 1986]. Although the principle behind bloomjoins and semijoins is similar, bloomjoins have certain advantages over regular semijoins. Bloomjoins consist of one join operation and one bloom filter creation, whereas regular semijoins involve one join step and one semijoin. Local processing costs of bloomjoins are less because the creation of a bloom filter is cheaper than that of a semijoin. Less data are transmitted between sites because bloom filters are smaller than the set of join attributes transmitted in semijoin operations.

An attendant disadvantage is that some tuples survive filtration due to the collision feature of hashing. Therefore, the cardinality of a relation reduced by filtration is likely to be higher than that reduced by a semijoin.

Performance

The bloom filter can be created on the relation with the fewest distinct values of the join attributes or on the smaller relation. The former strategy results in a more selective filter, but the latter is more practical since the former requires that the DBMS maintain a detailed and accurate statistical database profile.

The final join load can be reduced even further by reducing R with a bloom filter on S and joining the two reduced relations. This double reduction is similar to the operating principle of two-way semijoins described earlier.

Applicability

Bloomjoins can be used in distributed systems with a high-speed underlying network, where local processing costs approach data distribution costs. In such situations, semijoins are more expensive since they attempt to reduce the amount of data transmitted at the expense of increased local processing. A bloom filter-based semijoin algorithm with very low communication costs has been proposed in Mullin [1990]. Its only limitation is that, at present, because of its hash technique it is only used to support natural joins.

5.3 Summary

In the above discussion it is seen that reducing communication data volume achieved by reducing the join load is the primary objective. The algorithms are largely based on semijoin-like principles. This is true of most distributed query-processing strategies. The exact algorithms depend on the factors introduced in Section 5.1. In many cases, generation of optimal solutions is prohibitively expensive. As a result, heuristic procedures are fairly widespread.

6. HARDWARE SUPPORT FOR JOINS

The growing popularity of database systems has highlighted the need for fast query response. The large sizes of database files has made the task difficult. Processing large data files is a time-consuming activity that involves large amounts of I/O. This problem is magnified in the case of joins because at least two relations must be processed. In many cases, much of the I/O effort does not serve a purpose and should be avoided. The aim is to reduce I/O whether it is done by reducing the number of file scans, by partitioning, or by other means.

The idea that implementation of functions in hardware results in substantial gains in speed leads to the design and development of database machines. Numerous database machines using varied hardware technologies, ranging from general multiprocessors, to specially designed processors, to achieve response time reduction, have been proposed, and some have been prototyped [Ozkarahan 1986; Su 1988]. Some commercially available machines are Britton Lee's IDM [Britton Lee, Inc. 1981], Intel's[2] iDBP [Intel Corporation 1982], and Teradata's DBC [Ehrensberger 1984], which runs as a backend on some IBM[3] mainframes. Special-purpose database machines are expensive to design and build. Therefore, recent research has been aimed at the implementation of database operations on general-purpose multiprocessors [Dale et al. 1989; Su 1988; Walton 1989]. Database machine architectures have been surveyed extensively in Ozkarahan [1986] and Su [1988].

6.1 Hardware Approaches

Several approaches that have been found to be helpful in solving the problems associated with joining large data files are discussed in this section.

6.1.1. Reduction of Data

A common feature of all database activities is the vast amount of data that must be accessed. The size of the search space can be reduced by means of the cellular logic approach, of the data filter approach [Pramanik and Fotouhi 1985], or by using a large staging memory [Raschid et al. 1986].

In the *data filter approach*, a database filter can be used to ease the I/O bottleneck by reducing the quantity of data transferred from the disk to the main memory [Pramanik 1986; Su 1988]. It is located between the secondary storage devices and the main memory. It performs all the conventional secondary device control functions, as well as two data management functions, namely, data reduction and data transformation. Data reduction refers to the operations like select, which can be done by scanning a file once. Data transformation functions that some proposed filters can perform are sorting of an input file and merging, joining, and taking the difference of two input files. Transformation is more difficult than reduction; it may also involve several scans of the input data.

6.1.2 Fast Search

Searching through large data files has been speeded up by using associative memories, such as STARAN [Rudolph 1972] and ASLM [Hurson 1981].

Associative memories have certain properties that make them suitable for database management systems [Su 1988]. The most important feature is that they allow content-addressing and context-addressing capabilities. Each memory element has an associated processing element. This increases the parallel processing capability of the system. For instance, the select operation becomes very fast because a large number of tuples can be searched in parallel. This feature has been used to speed up the execution of the nested-loops join method.

6.1.3 Fast Processing

Fast processing has been achieved by means of architectural features such as general multiprocessing [Baru and Frieder 1986], pipelining [Richardson et al. 1987; Tong and Yao 1982], and systolic arrays [Kung and Lehman 1980]. The efficiency of join algorithms is increased by parallel execution and, in many cases, by parallel I/O as well. The HyperKYKLOS architecture, for example, consists of a multiple-tree topology network built on top of a hypercube. It allows the processing of joins using the nested-loops join, sort-merge join, a

[2] Intel is a registered trademark of Intel Corporation.
[3] IBM is a registered trademark of International Business Machines Corporation.

hash-based join method, or a semijoin-based algorithm [Menezes et al. 1987].

6.1.4 Others

(1) Hardware sorters have speeded up the sort-merge algorithm considerably, such as in DELTA [Sakai et al. 1984].
(2) Hardware implementations of hashing units support joins indirectly. Some architectures that use hashing units are CAFS [Babb 1979], DBC [Hsiao 1980], and GRACE [Kitsuregawa et al. 1983].
(3) Hardware for multiple search conditions as in CAFS [Babb 1979].

6.2 Nested-Loops Join

In this section, the impact of hardware support on the nested-loops join is discussed.

6.2.1 Associative Memory

STARAN implements the nested-loops algorithm using associative memories [Rudolph 1972]. The join operation is carried out as a sequence of select operations as described below. One tuple from the smaller relation is joined with all appropriate tuples from the larger relation. These tuples are identified by performing a select operation on the larger relation. The join attributes and their values for the tuple in the smaller relation are used as the search predicates. The join attribute values are loaded into the comparand register, while the larger relation is loaded into the associative arrays. In each step, the matching tuples from the longer relation are concatenated with the tuple with which they are currently being compared. This is followed by a project operation on the result relation to remove duplicate attributes. Any duplicate tuples are automatically deleted from the result as part of the project operation. This method of performing the join operation is expensive (in terms of hardware costs) for joins with low selectivities. This method, however, has the main advantage of speed over any software implementation.

Another system that executes an associative version of the nested-loops method is the *associative parallel join module* [Hurson 1989]. The steps in this algorithm are as follows:

```
load R and S into associative modules;
presearch R and S to mark off nonrelevant tuples;
for each r do
    { route r to concatenation unit;
    compare r with all s in S module;
    route matching s to concatenation unit;
    concatenate tuples;
    route concatenated tuples to result module;
    send acknowledge signal to R module};
```

If relation S is too large to fit in its module, it must be split into several subrelations, each small enough to fit in the module. The above steps are then performed for each subrelation.

6.2.2 Special-Purpose Join Processor

The DBM (database machine) designed at the University of Maryland uses a specially designed join processor called the *join filter* (JF) [Yao et al. 1981]. The join filter is used to execute join operations. Its operation is controlled by a timing and control logic unit. It is a two-dimensional array of comparators that receives data from two sources, namely, the two relations being joined. It contains a number of buffers for storing the join values and a buffer for storing the result. The R and S buffers are one-dimensional buffers used to store the join values of the two relations. Their contents are simultaneously broadcast to the corresponding rows and columns of comparators C. A bit matrix B is used to record if values in a pair of R and S buffers match. The B values are used to generate the join result. The join result is stored in an output buffer F in the form of addresses of matched tuples. The *database filter* (DF) consisting of a two-dimensional array of parallel comparators is central to the processing of joins. This component is used to perform the select and project operations.

The join operation is performed in the join processor array in the following

manner:

```
partition R into x subrelations;
project out join attributes in DF;
store join attributes of each subrelation in
R(x);
partition S into y subrelations;
project out join attributes in DF;
store join attributes of each subrelation in
S(y);
for each set of x-values do
    { broadcast x-values along rows in com-
    parator array;
    broadcast addresses of tuples into F
    buffers;
    for each set of y-values do
        { broadcast y-values along columns in
        comparator array;
        broadcast addresses of tuples into F
        buffers;
        perform comparisons in all compara-
        tors;
        store results in B matrix;
        for each bit in B matrix which is set
        do
            output tuple addresses from F
            buffers}};
```

The efficiency of this system is tied to the ratio between the size of the relations being joined and the size of the join processor array. It exploits the inherent parallelism of the nested-loops join. A potential drawback of the system lies in the fact that duplicate tuples in the result relation are not removed as part of the join process.

6.3 Sort-Merge Join

Examples of hardware implementations of the sort-merge are given in this section.

6.3.1 Hardware Sorters

The database machine DELTA sorts the input relations in parallel and uses a single merger to perform the merge phase [Sakai et al. 1984].

6.3.2 Filtering

The sort-merge algorithm is used in the database machine VERSO [Bancilhon et al. 1983]. The relations are sorted on the join attributes and loaded into 233the filter buffers $SB1$ and $SB2$. Tuple pairs being tested for the join condition are loaded into the registers $Q1$ and $Q2$. The movement of the relations within the filter is controlled by an automaton program, which is the compiled code of a relational query. The filter control decides when a tuple needs to be read in from the filter buffers into the registers. Tuples are read from the buffer into the registers $Q1$ and $Q2$ only if they will actually be present in the result of the join. The comparison steps in the algorithm described below decide which tuple to read in next and from which relation. Tuples that do not have a match, or would result in a duplicate in the result relation, are filtered out. The equijoin operation is performed as follows:

```
repeat
    load a block from R into SB1;
    load a block from S into SB2;
    repeat
        load r into Q1;
        load s into Q2;
        if r(a) = s(b) then join r and s;
        r(a) < s(b) then load next r from
        Q1;
        r(a) > s(b) then load next s from
        Q2;
    until end-of-block is reached;
until either R or S is exhausted.
```

Note that the relations are sorted before being fed into the filter.

6.4 Hash-Based Joins

Some hardware implementations of hash-based joins are described briefly in this section.

6.4.1 Data Filtering

Data filtering is a form of implicit partitioning. It consists of elimination of nonmatching tuples using Babb arrays as in the database machine CAFS [Babb 1979]. A Babb array is a single bit-wide random access memory (RAM). There is a one-to-one mapping between all distinct join attribute values in a relation and the addresses of the bits in the array. In the simplest case, the attribute value itself can be used as an address pointing to a bit. For relations with a wide range of

attribute values, however, a very large store would be needed. Furthermore, large parts of this array may not be used at all if the number of distinct attribute values in the relation is small. The mapping is usually obtained by hashing attribute values and using the hash value as the address. This method is used in the algorithm described below.

The equijoin procedure using hashed bit arrays is as follows [Su 1988]:

```
for each tuple r do
    {hash join attribute
    store attribute value in list P1
    set corresponding bit in Babb array};
for each tuple s do
    {hash join attribute
    if corresponding bit is set in Babb array
    then
        store attribute value in list P2};
for each attribute value in list P2 do
    {if attribute value not in list P1 then
        delete from P2
        else concatenate corresponding r and
        s
        place result in Q};
```

The above algorithm is based on the concept of filtering. The lists P1 and P2 must be matched in the merge step because of possibility of collisions.

6.4.2 Multiprocessing

Most multiprocessor hash algorithms are based on explicit partitioning, for example, GRACE hash join. The hashing units are used to partition the relations and also to join the tuples in each partition pair using the simple hash join. Most of these algorithms have already been described in Section 2.

Teradata's DBC/1012 is a commercially available backend processor with special access module processors (AMP) for the execution of database operations. The join strategy depends on whether either of the relations has a primary index on the join attributes. If there are no indexes, the larger relation is distributed over the AMPs. The smaller relation is broadcast to all sites, where it is joined with the fragment of the larger relation at that site using the simple nested-loop or sort-merge join method. If one of the relations has a primary index on the join attribute, it is distributed across the AMPs based on the hash values of the index. The same hash function is then applied to the other relation. The tuples of this relation are then broadcast only to the AMP, where they are likely to find a match. Each AMP then performs its part of the join operation. Other data placement and processing details are discussed at length in Su [1988].

6.4.3 Pipelining

The hash loops join with pipelining and multiprocessing has been implemented in DBC [Hsiao 1980]. It uses multiple processors by partitioning one of the input relations across the processors. Each tuple of the other input relation is successively sent to all the processors. Thus, at any given time, all processors are processing different tuples of the second relation.

6.5 Summary

It is seen that most join algorithms implemented in hardware perform simple actions repetitively. Software implementations of the same algorithms would generally be considered inefficient. Prerequisites that an algorithm must have before it becomes a viable candidate for hardware implementation are communication simplicity, space efficiency, scope for parallelism, and regularity [Hurson 1986]. The speed of hardware compensates for the brute force approach.

Simple implementations of the nested-loops join, the parallel nested-loops join, the sort-merge join, and the hash join are found to be the underlying paradigm in most cases. Algorithms based on semijoins have been used with great success.

7. OTHER JOIN ISSUES

The previous sections of this paper have examined types of join operations and techniques for implementation. In this section we discuss various topics related to the efficiency of join operations. These

issues are related to multiple types of algorithms and have merited research on their own. Thus, we include a separate discussion of them.

7.1 Selectivity Factor

The *selectivity factor* [Piatetsky-Shapiro and Connell 1984] or the *join selection factor* [El-Masri and Navathe 1989] is defined as the ratio of the number of tuples participating in the join to the total number of tuples in the Cartesian product of the relations. It is an important factor in the cost of performing joins. A high selectivity factor requires a larger number of tuple comparisons, produces a larger result relation, and requires more I/O than does a low selectivity factor. Thus, a high selectivity factor implies a more expensive join. In this section, join algorithms are evaluated based on the effect of selectivity on the performance.

Based on the actual execution cost, some algorithms may be preferred over others for different levels of selectivity. The nested-loops method is considered the most inefficient method to use in the case of low join selectivities. This is because most of the comparisons do not result in a match, and the effort is wasted. The hash join methods are significantly better when the selectivity is low. The advantage that hash joins have over the nested-loops method diminishes as the selectivity factor increases. In this case, exhaustive comparison is useful because of the large number of tuples participating in the join. Furthermore, the nested-loops method does not have the overhead of doing hashing. In partition-based joins, the selectivity of the join between tuples in a partition should be higher than that of the join between the two relations when treated as a whole.

The problem of estimation of selectivities to an acceptable degree of accuracy remains open even though it has been studied in some detail in the past. The general approach has been to study the distribution of attribute values and to find a useful way of storing such information. The uniform assumption method [Selinger 1979], the worst-case assumption method [Epstein and Stonebraker 1980], and the perfect knowledge method [Christodulakis 1985] are some approaches that have been advocated. The first method assumes that the values of attributes are uniformly distributed within the domain of the attribute. Such an assumption simplifies processing but is considered somewhat unrealistic. The worst-case assumption method assumes a selectivity of one, that is, a Cartesian product. This, too, is unrealistic. The perfect knowledge method does not make any assumptions about the distributions. Instead, it relies on the calculation of exact relation sizes at the time of processing. This approach gives good results. The cost of calculating the storage overhead and the cost of maintaining such information are, however, considered prohibitive. The piecewise uniform method [Bell et al. 1989] represents a compromise between the above methods. Another class of methods, called *sampling methods*, does not require storing and maintaining large amounts of statistical data about relations [Lipton et al. 1990]. An adaptive, random sampling algorithm for estimating selectivities of queries involving joins and selects is discussed in Lipton et al. [1990]. It also considers skewed data and very low selectivities.

7.2 Optimal Nesting of Joins

When performing a join over several relations, the order in which the joins is performed may make a difference to the overall efficiency. This problem of optimal nesting of joins is usually considered in the context of the overall problem of query processing. The usual objective is to reduce the size of intermediate relations the most. The optimal order is the one that produces the fewest total number of intermediate tuples; thus the effect of the selectivity factor is extremely significant in the case of multiway joins [Bell et al. 1989; Kumar and Stonebraker 1987; Piatetsky-Shapiro and Connell 1984]. An optimal ordering for

such joins is achieved by processing the relations in the order of increasing selectivities. The joins with the lowest selectivities should be performed first. This logic also applies to the case of join processing in distributed environments where semijoins are used to reduce the size of the relations participating in the join [Masuyama et al. 1987; Segev 1986].

The accuracy with which the size of intermediate relations can be estimated depends on the accuracy of the selectivity factor. It has been found that inaccurate estimates of the selectivity factor are more likely to lead to the generation of suboptimal query execution plans in the case of distributed database systems [Epstein and Stonebraker 1980] than in the case of centralized databases [Kumar and Stonebraker 1987].

In this context, it must be mentioned that the problem of large intermediate result relations can be avoided by pipelining the process of joining several relations [Bitton 1987]. If the results generated by one join are directly fed to the next join stage, there is no need to store intermediate relations.

7.3 Hash Joins

In most situations, hash joins have generally been found to be the most efficient method. The efficiency of hash joins is reduced by partition overflow in the case of hash-partitioned joins. Hash table collisions are a problem in all hash joins. The problems are discussed in the following sections.

7.3.1 Partition Overflow

Partition overflow is said to occur when a partition becomes too large to fit into the available main memory. The number of partitions, and therefore the size of the partitions, into which a relation must be split is determined by the amount of main memory available and the size of the input relation. There is, however, no easy way of guaranteeing that each of the partitions will be small enough to fit into the staging memory. This is because the number of tuples that will hash to a given range of values cannot be accurately estimated in advance. In most cases, the number of partitions is statically determined before the partitioning process even begins.

It has been suggested [Gerber 1986] that most cases of partition overflow can be avoided by creating more partitions than appears to be necessary. In other words, it may be possible to avoid partition overflow by creating a large number of small partitions. A small partition is more likely to fit into the available memory in its entirety. If the exact amount of available memory is known, the partition size should be created so the largest partition possible without overflow is used.

Another suggested solution proposes that a family of hash split functions be maintained [Gerber 1986]. The efficiency with which a given hash function can randomize tuples across partitions for specific attributes can be precomputed and stored. This information can be used at the time of partitioning. It must be remembered that this information is subject to change depending on the modifications to the database. These strategies are examples of *static partitioning*. If all attempts at preventing partition overflow fail, the only solution may be to detect overflow and be prepared to resplit the relation using another hash function. Based on this principle, redistribution of tuples is sometimes done at runtime to balance the processing load for each partition. This is known as *dynamic partitioning* [Su 1988]. The subject of partitioning is detailed in Section 7.5.

The GRACE hash join avoids partition overflow by means of a strategy called partition or bucket tuning. It is based on the fact that, inspite of the best attempts, not all partitions are of a size such that they will fit exactly into the amount of main memory available. The main idea behind this approach is that partitions that are too large can be divided into smaller divisions as and when needed. Recall that the GRACE hash join first divides each of the relations into partitions. Then tuples within each of the

partitions are joined. These two separate parts of the algorithm are referred to as *stages*. When the second stage is performed, the partitions chosen for staging are based on the best-fit criterion. This method of staging partitions avoids the problem of partition overflow. It has been called the dynamic destaging strategy [Nakayama et al. 1988]. The effect of tuple distribution within buckets and bucket size tuning on this algorithm has been reported in Kitsuregawa [1989c].

7.3.2 Hash Table Collisions

In hash-based join methods, collisions must be kept at a minimum to reduce the cost of the probing operation during the join stage [Gerber 1986]. It is well known that there is no way of avoiding collisions altogether. As a result, standard methods of dealing with hash collisions have been developed. Most methods involve the overhead of maintaining lists of values that hash to the same entry in the hash table.

A unique method of handling collision has been implemented in the CAFS architecture [Babb 1979]. It has been found to be very useful inspite of the fact that there is a small possibility of error. Each attribute value is hashed by three different hash functions. The hash value is used to mark the appropriate bit in the corresponding bit array store. The output of the 3-bit array is logically ANDed. The bit array is considered to have been marked for a particular attribute value only if the output of the AND operation is 1.

7.3.3 Nonequijoins

Nonequijoins can be performed using hash join techniques only if the hash function maintains ordering of tuples. Since such hash functions are rare, hash joins are usually not associated with equijoins. This, however, does not really detract from the use of hash joins since nonequijoins are uncommon; hash join techniques remain among the most efficient methods available.

7.4 Indexing and Clustering

The performance of some join methods is affected by the presence of indexes on the join attributes and whether or not the index is clustered. For instance, it has been shown that the performance of even the nested-loops join method can be improved substantially if the inner relation is indexed on the join attributes [Blasgen and Eswaran 1977]. In some cases, it may even be advantageous to create indexes dynamically [Omiecinski 1989].

Join methods using specialized data structures, such as Bc-trees or T-trees, must necessarily have an index using the data structure in question to be viable.

If both relations are clustered on the join attributes, then the sort-merge method is the best and requires the least amount of I/O activity [Omiecinski 1989]. If the indexes are unclustered, heuristics can be used to devise efficient procedures for implementing joins [Omiecinski 1989; Omiecinski and Shonkwiler 1990].

7.5 Partitioning of Relations

In Section 2, it was observed that algorithms based on partitioning of relations are suitable for use on parallel systems. Significant gains in performance, however, can be achieved only when the partitions are of approximately equal size. Most performance results reported assume equal-sized partitions. In practice it is very difficult to create equal-sized partitions. The reasons for this and the resulting effect on the performance of joins are discussed below.

7.5.1 Sources of Skewness

In most multiprocessor systems, relations are horizontally partitioned and distributed across all storage units. In response to queries, these partitioned data are processed in parallel by a number of processors. Query response time is determined by the time taken by individual processors. If all processors handle equal amounts of data, the maximum performance improvement is achieved; otherwise, the time is determined by the processor that handles the largest

volume of data. In practice the volume of data handled by individual processors varies. There are several reasons for this [Lakshmi and Yu 1988, 1989; Walton 1989]:

Skewed distribution of attribute values. Horizontal partitioning, and therefore distribution of data, is usually done based on the values of some attribute in the relation. For instance, the attribute domain is divided into fixed-size ranges, and tuples falling in a given range are sent to a particular storage location. If the values of the attribute in question are present with uniform frequency, the subrelations will be of equal size. Such a uniform distribution is, however, rarely encountered in practice. As a result, any distribution method based on attribute values is bound to result in unequal subrelations.

Result of select. Database queries consist of various operations being performed on a set of relations. A general rule of query optimization is that the join operation is performed after all select operations. The selectivity of these operations may be different for different partitions. Thus, even if the partitions at all locations were equal to begin with, the subrelations that are input to the join operation are likely to vary substantially in size.

Result of hash function. Hash functions are often used to distribute tuples of a relation over several storage locations. This is an alternative that may be chosen over the fixed-range distribution discussed earlier. In the ideal case, the hash function produces hash values that are uniformly distributed. In such a situation, the relation being distributed is divided into equal parts. This, however, is usually not the case. Hash functions usually contribute to the uneven distribution of tuples.

7.5.2 Effect of Skew on Join Performance

It has been found that each of the factors that can result in unequal-sized partitions has an effect on join performance and that different factors have different effects [Walton 1989].

Speed up. The *speed up* of an algorithm as result of parallelization is the ratio of the execution times of the sequential and parallel versions of the algorithm. Maximum speed up can be achieved when the join processing load is evenly distributed among all processing sites. Otherwise, the site with the largest load becomes the bottleneck. Partitioning has been the primary method of distributing the join load. In most cases, it is assumed that the attribute values are uniformly distributed in a relation. This is usually not true. As a result, some partitions are significantly larger than others. This nonuniformity in the sizes of partitions limits the speed up [Lakshmi and Yu 1988, 1989, 1990].

Scalability. An algorithm is said to be *scalable* if its performance increases approximately in proportion to the number of processors; that is, the speed up of the parallel version of an algorithm is approximately proportional to the number of processors [Dale et al. 1989; Walton 1989]. In Lakshmi and Yu [1988], experiments suggest that scalability is adversely affected if the data are skewed. Increasing the number of processors did not result in a corresponding increase in speed up. Initial results [Walton 1989] indicate that although all types of skew increase the execution time of joins, the amount of increase depends on the source of skew. Further, it is possible to anticipate this problem and take steps to resolve it; for example, by dynamic redistribution of tuples before joining.

7.5.3 Cure for Skew

Ideally, each processor participating in a join would have subrelations approximately equal to the subrelations at other processors. Many database systems have recognized the fact that this cannot be taken for granted and have implemented schemes that attempt to create equal-sized partitions at all processors. A large number of static partitioning options is used for database partitioning [Ozkara-

han 1986], for example, sorting, hash-based data clustering, and dynamic order-preserving partitioning [Ozkarahan and Bozsahin 1988].

Most systems have static schemes that distribute relations evenly across all storage locations. An improvement can be made on this by balancing the load at the time of join execution. Balancing the size of partitions at the time of execution is also known as dynamic data distribution [Su 1988]. Experiments show that arbitrary distribution of data followed by dynamic redistribution at the time of processing is a viable alternative to static partitioning based on hashing [Wang and Luk 1988] as long as the communication network does not become a bottleneck. Some static load distribution strategies are as follows:

Round robin. Tuples are distributed in a round-robin fashion based on the value of some attribute.

Hashed. Tuples are stored at a particular location based on the result of applying a hash function to a given attribute. Obviously, the success of this scheme depends on the efficiency of the hash function.

Range partitioned. The domain of a given attribute is split into ranges specified by the user. Each range maps to a particular storage location. Tuples whose attribute values fall into a given range are stored at the storage location determined by the mapping.

Uniform distribution. A relation is divided into equal-sized parts based on some attribute value. The relation segments are then distributed to the available disk drives.

Dynamic data redistribution during execution is advantageous but can be time consuming unless the degree of connectivity between processors is high. Thus far, only cube-connected systems appear to have the degree of connectivity needed to support dynamic redistribution [Su 1988]. A scheme for tuple balancing at execution time in a cube-connected database multicomputer is described in Baru and Frieder [1989]. A parallel hash join algorithm, called bucket-spreading hash join, has been implemented in a high-performance database server called super database computer (SDC) [Kitsuregawa and Ogawa 1990]. Buckets are dynamically allocated to processors based on their size. Again, SDC relies on a highly functional network to perform redistribution of data buckets without burdening the processors. Other dynamic load distribution schemes are discussed in Hua and Lee [1990] and Ghandeharizadeh and DeWitt [1990]. In Hua and Lee [1990], the partitioning scheme is based on the grid file structure. The hybrid-range scheme of Ghandeharizadeh and DeWitt [1990] is a compromise between simple range partitioning and hashed partitioning.

The idea of dynamic data partitioning has been implemented by Tandem in its NonStop SQL systems [Tandem Database Group 1987]. Here, relations are horizontally partitioned among processors with a split table defining partitioning predicates. Special *separator* and *merge* operators are then used to parallelize processing and scanning of the relation partitions based on the join predicates dynamically [DeWitt and Gray 1990]. The result is a parallel scan and join occurring at each site.

7.6 Join-Type Processing in Nonrelational Databases

Although the join operation is unique to relational algebra and query languages on relational databases, equivalent operations exist in other data models. In fact, the use of join-type operations predates what is currently thought of as database systems. With the introduction of direct access files [Harbron 1988], the capability of linking two different files together was provided. By using pointers between such files, users were provided the ability to "join" the two files together in predefined ways. Early bill-of-material processors facilitated the easy processing of these joined files by providing "chain-chasing" operations.

With the development of the DBTG (Database Task Group) database standard in 1971 [CODASYL], the DML (Data Manipulation Language) operations required by network database systems were defined. Many of these operations allow the chasing of chains between records. As with the early file systems, however, only predefined chains (sets) with predefined joins were allowed. Operations such as FIND OWNER and FIND NEXT are used to chase chains in network databases. IBM's IMS DBMS [1978] provides similar chain-chasing capabilities. Since IMS is only hierarchical in nature, however, the types of joining operations that are allowed is somewhat reduced from that of the DBTG network systems. Operations such as GET NEXT and GET NEXT WITHIN PARENT are provided. Via the use of logical relationships and logical databases, IMS provides the ability to combine segments from one physical database with segments from another dynamically. To the user, it appears as if the two segments are joined together into one. IMS also facilitates path calls that allow the joining of segments from multiple levels in the same database into one segment. The concatenated key for IMS is the key of all the joined segments.

Object-oriented databases also provide operations similar to a join. The idea of a class hierarchy that indicates inheritance relationships (ISA) between object classes must often be traversed [Atkinson et al. 1989]. Its traversal requires a join-type operation to travel between the levels in the structure. Since other relationships are also allowed between objects and object classes, other types of join operations are often provided. The ORION query model provides the ability to follow the ISA relationship as well as complex attribute relationships [Kim 1989]. As stated by Kim [1989], this type of operation "is an implicit join of the classes on a class-composition hierarchy rooted at the target class of the query." Unlike relational databases where foreign keys are often used to facilitate the join, object-oriented database systems may use the unique object identifier. Special indexes may be built to provide a fast retrieval of the joined objects [Bertino and Kim 1989]. Complex attributes also require a type of join operation to find the actual primitive attribute values [Banerjee et al. 1988].

Recently, many researchers have investigated the extension of relational database systems to provide features beyond those of the original relational proposal and its associated relational algebra. Starburst extensible DBMS provides a pointer-based access structure [Carey et al. 1990; Shekita and Carey 1990]. The purpose of these pointers is to provide an efficient implementation technique for joins. The result is a pointer structure similar to that provided by the earlier network and hierarchical models. An extension to the relational model to handle temporal data and join processing on it has been proposed [El-Masri 1990]. The idea of a join has also been extended to include non-1NF data. Postgres has extended relations to include procedures as data types [Stonebraker and Rowe 1986]. A MATCH operator that performs join-type operations for pattern matching has also been proposed [Held and Carlis 1987]. An experimental database system called RAD, which allows comparisons (and thus joins) between arbitrary abstract data types, has been proposed [Osborn and Heaven 1986]. These comparisons are provided by user-defined procedures. Join algorithms have been parallelized for nested relations [Deshpande 1990].

8. CONCLUSIONS

A large number of algorithms for performing the join operation is in use. It has been found that many algorithms are variants of others, often tailored to suit a particular computing environment. For instance, the hash-loops join is a variant of the nested-loops join that takes advantage of the presence of a hashed index on the join attributes of the inner relation. An algorithm and its derivatives may be considered as a class of algorithms.

Some algorithms require preprocessing, for example, sorting of relations in

the sort-merge method. The efficiency of the split-based algorithms is on account of the join load reduction. Others require special index structures to be viable methods, for example, T-trees. The large number of algorithms of each type indicates that different computing environments demand different join methods. This feature could have an effect on the process of query optimization. In fact, the proportion of query-processing literature devoted to the optimization of the join algorithm is larger than for any other aspect of query processing. The purpose of this survey has been to study join algorithms, to provide a means of classifying them, and to point to query-processing techniques based on particular join methods.

Join processing remains an area of active research. New algorithms are being devised for specific computing environments, parallel architectures [Baru et al. 1987; Lakshmi and Yu 1990; Nakayama 1984; Su 1988], main memory databases [DeWitt et al. 1984; Pucheral et al. 1990; Shapiro 1986], and distributed databases [Bandyopadhyay and Sengupta 1988; Mullin 1990; Pramanik and Vineyard 1988]. They are usually variants or extensions of the basic methods reviewed here. Optimization of queries involving joins is also being studied in detail. For example, queries involving joins using large data files are examined in Kitsuregawa et al. [1989a] and Swami and Gupta [1988]. The issue of processing join queries accessing a large number of relations is addressed in Ioannidis and Kang [1990]. Queries involving joins and outerjoins are discussed in detail in [Rosenthal and Galindo-Legaria 1990]. A join-processing method, called *intelligent join*, consisting of determining which relations need to be joined in order to respond to a query and the order in which they could be joined has been proposed in Cammarata et al. [1989]. The problem of optimizing processor use and reducing I/O bandwidth requirements for join processing on multiprocessors has been studied in Murphy and Rotem [1989a, 1989b].

APPENDIX: NOTATION

Symbol	Explanation
R	Input relation
S	Input relation
Q	Result relation
n	Cardinality of R
m	Cardinality of S [Note: It is assumed that $n > m$.]
r	Tuple in relation R
s	Tuple in relation S
a_i	All attributes of r
b_j	All attributes of s
$r(a)$	Join attributes in relation R
$s(b)$	Join attributes in relation S
$s(a)$	Join attributes in relation S This is necessary for an equijoin between R and S
$t(a)$	Join attribute columns in result relation Q
$t(b)$	Join attribute columns in result relation Q

ACKNOWLEDGMENTS

The authors would like to thank the referees for their valuable comments and recommendations for improvements to early drafts of this paper. Special thanks go to Salvatore March for his thorough review of the paper and to Masaru Kitsuregawa for inspiring our partitioning classification and Figure 9.

This material is based in part upon work supported by the Texas Advanced Research Program (Advanced Technology Program) under Grant No. 2265.

REFERENCES

AGRAWAL, R., DAR, S., AND JAGADISH, H. V. 1989. Composition of database relations. In *Proceedings of Conference on Data Engineering*, pp 102–108.

AHO, A. V., BEERI, C., AND ULLMAN, J. D. 1979. The theory of joins in relational databases. *ACM Trans. Database Syst. 4*, 3 (Sept.).

ATKINSON, M., BANCILHON, F., DEWITT, D., DITTRICH, K., MAIER, D., AND ZDONIK, S. 1989 The object-oriented database system manifesto. In *Proceedings of the Deductive and Object Oriented Databases Conference.*

BABA, T., SAITO, H., AND YAO, S. B. 1987. A network algorithm for relational database operations. In *International Workshop on Database Machines*, pp. 257–270.

BABB, E. 1979. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst..4*, 1, 1–29.

BANCILHON, F., RICHARD, P., AND SCHOLL, M 1983. VERSO: The relational database machine. In *Advanced Database Machine Architecture*, D. Hsiao, Ed., Prentice-Hall, Englewood Cliffs, N.J.

BANDYOPADHYAY, S., AND SENGUPTA, A. 1988. A robust protocol for parallel join operation in distributed databases. In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, pp. 97–106.

BANERJEE, J., KIM, W., AND KIM, K.-C. 1988. Queries in object-oriented databases. In *Proceedings of the 4th International Conference on Data Engineering* (Feb.), pp. 31–38.

BARU, C. K., AND FRIEDER, O. 1989. Database operations in a cube-connected multicomputer system. *IEEE Trans. Comput. C-38*, 6 (June), 920–927.

BARU, C. K., FRIEDER, O., DANDLUR, D., AND SEGAL, M. 1987. Join on a cube: Analysis, simulation and implementation. In *Proceedings of International Workshop on Database Machines* (Dec.), pp. 74–87.

BEERI, C., AND VARDI, M. Y. 1981. On the properties of join dependencies. In *Advances in Database Theory*, vol. 1. Plenum Publishing, New York, pp. 25–72.

BELL, D. A., LING, D. H. O., AND MCCLEAN, S. 1989. Pragmatic estimation of join sizes and attribute correlations. In *Proceedings of Conference on Data Engineering*, pp. 76–84.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM, 18*, 9 (Sept.), 509–516.

BENTLEY, J. L. 1979. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng. SE-5*, 4 (July).

BENTLEY, J. L., AND KUNG, H. T. 1979. A tree machine for searching problems. *IEEE Conference on Parallel Processing*, pp. 257–266.

BERNSTEIN, P. A., AND CHIU, D.-M. W. 1981. Using semi-joins to solve relational queries. *J. ACM 28*, 1 (Jan.) 25–40.

BERNSTEIN, P. A., AND GOODMAN, N. 1979a The theory of semijoins. Computer Corporation of America Rep. 79-27, Cambridge, Mass.

BERNSTEIN, P. A., AND GOODMAN, N. 1979b. Inequality semijoins. Computer Corporation of America Rep. 79-28, Cambridge, Mass.

BERNSTEIN, P. A., AND GOODMAN, N. 1980. The power of inequality semijoins Aiken Computation Lab Rep. 12-80, Harvard Univ., Cambridge, Mass.

BERTINO, E., AND KIM, W. 1989. Indexing techniques for queries on nested objects. *IEEE Trans. Knowl. Data Eng. 1*, 2 (June), 196–214

BITTON, D., HANRAHAN, M. B., AND TURBYFILL, C. 1987. Performance of complex queries in main memory database systems. In *Proceedings of Conference on Data Engineering*, pp. 72–81.

BITTON, D., BORAL, M., DEWITT, D. J., AND WILKINSON, W. K. 1983. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst. 8*, 3 (Sept.), 324–353.

BLASGEN, M. W., AND ESWARAN, K. P. 1977. Storage and access in relational databases. *IBM Syst. J. 16*, 4, 363–377

BLOOM, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July), 422–426.

BRATBERGSENGEN, K. 1984 Hashing methods and relational algebra operations. In *Proceedings of Conference on Very Large Data Bases*, pp. 323–333.

BRITTON LEE, INC. 1981. *IDM 500: Intelligent Database Machine Product Description.*

CAMMARATA, S., RAMACHANDRA, P., AND SHANE, D. 1989. Extending a relational database with deferred referential integrity checking and intelligent joins. In *Proceedings of SIGMOD*, pp. 88–97.

CAREY, M., SHEKITA, E., LAPIS, G., LINDSAY, B., AND MCPHERSON, J. 1990. An incremental join attachment for starburst. In *Proceedings of the 16th VLDB Conference* (Brisbane, Australia), pp 662–673

CERI, S., GOTTLOB, G., AND PELAGATTI, G. 1986 Taxonomy and formal properties of distributed joins *Inf. Syst. 11*, 1, 25–40.

CHANG, J. M., AND FU, K. S. 1980. A dynamic clustering technique for physical database design. In *Proceedings of SIGMOD*, pp. 188–199.

CHEINEY, J.-P., FAUDEMAY, P., AND MICHEL, R. 1986. An extension of access paths to improve joins and selections. In *Proceedings of Conference on Data Engineering*.

CHEN, J. S. J., AND LI, V. O. K. 1989. Optimizing joins in fragmented database systems on a broadcast local network. *IEEE Trans. Softw. Eng. 15*, 1 (Jan.), 26–38.

CHRISTODULAKIS, S. 1985 Estimating block transfer and join sizes In *Proceedings of SIGMOD*
CODASYL, 1971. Data Base Task Group Report.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June), 377–387.

CODD, E. F. 1972. Relational completeness of data base sublanguages. In *Data Base Systems*. Prentice-Hall, Englewood Cliffs, N.J., pp. 65–98.

CORNELL, D. W., AND YU, P. S. 1988. Site assignment for relations and join operations in the distributed transaction processing environment. In *Proceedings of Conference on Data Engineering*, pp. 100–108.

DALE, A. G., MADDIX, F. F., JENEVEIN, R. M., AND WALTON, C. B. 1989. Scalability of parallel joins on high performance multicomputers. Tech. Rep. TR-89-17, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, Tx.

DATE, C. J. 1983. The outer join. In *Proceedings of International Conference on Databases* (Cambridge, England), pp. 76–106.

DAYAL, U. 1985. Query processing in a multidatabase system. In *Query Processing in*

Database Systems, W. Kim, D. S. Reiner, and D. S. Batory, Eds. Springer-Verlag, New York, pp. 81–108.

DESAI, B. P. 1989. Performance of a composite attribute and join index. *IEEE Trans. Softw. Eng. SE-15*, 2 (Feb.), 143–152.

DESAI, B. C. 1990. *An Introduction to Database Systems*. West Publishing Co, St. Paul, Minn.

DESHPANDE, V., LARSON, P.-A., AND MARTIN, T. P. 1990. Parallel join algorithms for nested relations on shared-memory multiprocessors. *IEEE Symposium on Parallel and Distributed Processing*, pp. 344–351.

DEWITT, D., AND GERBER, R. J. 1985. Multiprocessor hash-based join algorithms. In *Proceedings of Conference on Very Large Data Bases*, pp. 151–164.

DEWITT, D. J., AND GRAY, J. 1990. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Rec. 19*, 4 (Dec.), 104–112.

DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. 1984. Implementation techniques for main memory datab ase systems. *Proceedings of SIGMOD*, pp. 1–8.

DUTKA, A. F., AND HANSON, H. H. 1989. *Fundamentals of Data Normalization*. Addison-Wesley, Reading, Mass.

EHRENSBERGER, M. J. 1984. The DBC/1012 database computer's systems-architecture, components, and performance. In *Minnowbrook Workshop on Database Machines*.

EL-MASRI, R., AND NAVATHE, S. B. 1989. *Fundamentals of Database Systems*. Benjamin/Cummings, Menlo Park, Calif.

EL-MASRI, R., WUU, G. T. J., AND KIM, Y.-J. 1990. The time index: An access structure for temporal data. In *Proceedings of the 16th VLDB Conference* (Aug., Brisbane, Australia), pp. 1–12.

EPSTEIN, R., AND STONEBRAKER, M. 1980. Analysis of distributed database processing strategies. In *Proceedings of Conference on Very Large Data Bases*, pp. 92–101.

EPSTEIN, R. 1982. *Query Processing Techniques for Distributed, Relational Database Systems*. University Microfilms International, Ann Arbor, Mich.

FAGIN, R. 1979. Normal forms and relational database operators. In *Proceedings of SIGMOD*.

FOTOUHI, F., AND PRAMANIK, S. 1989. Optimal secondary storage access sequence for performing relational join. *IEEE Trans. Know. Data Eng. 1*, 3 (Sept.), 318–328.

FUSHIMI, S., KITSUREGAWA, M., NAKAYAMA, M., TANAKA, H., AND MOTO-OKA, T. 1985. Algorithm and performance evaluation of adaptive multidimensional technique. In *Proceedings of SIGMOD*, pp. 308–318.

GARDARIN, G., AND VALDURIEZ, P. 1989. *Relational Databases and Knowledge Bases*. Addison-Wesley, Reading, Mass.

GERBER, R. J. 1986. Dataflow query processing using multiprocessor hash-partitioned algorithms. Computer Sciences Tech. Rep. No. 672, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wisc.

GHANDEHARIZADEH, S., AND DEWITT, D J. 1990. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of Conference on Very Large Data Bases*, pp. 481–492.

GOODMAN, J. R. 1981. An investigation of multiprocessor structures and algorithms for database management. Tech. Rep. UCB/ERL, M81/33, Univ. of California, Berkeley.

GOUDA, M. G., AND DAYAL, U. 1981. Optimal semijoin schedules for query processing in local distributed database systems. In *Proceedings of SIGMOD*, pp. 164–173.

GOYAL, P., LI, H. F., REGENER, E., AND SADRI, F. 1988. Scheduling of page fetches in join operations using Bc-trees. In *Proceedings of Conference on Data Engineering*, pp. 304–310.

GRAEFE, G. 1989. Relational division: Four algorithms and their performance. In *Proceedings of Conference on Data Engineering*, pp. 94–101

GYSSENS, M. 1986. On the complexity of join dependencies. *ACM Trans. Database Syst. 11*, 1 (Mar.), 81–108

HAGMANN, R. B. 1986. An observation on database buffering performance metrics. In *Proceedings of Conference on Very Large Data Bases*, pp. 289–293.

HARADA, L., NAKANO, M., KITSUREGAWA, M., AND TAKAGI, M. 1990. Query processing method for multi-attribute clustered relations. In *Proceedings of Conference on Very Large Data Bases*, pp. 59–70.

HARBRON, T. R. 1988. *File Systems Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, N.J.

HELD, J. P., AND CARLIS, J. V. 1987. MATCH: A new high-level relational operator for pattern matching *Commun. ACM 30*, 1 (Jan.), 62–75.

HSIAO, D. K. (ED.) 1980. *Collected Readings on a Database Computer (DBC)*. The Ohio State University, Columbus, Ohio

HUA, K. A., AND LEE, C. 1990. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of Conference on Very Large Databases*, pp. 493–506.

HURSCH, J. L. 1989. Relational joins: More than meets the eye. *Database Program. Design 2*, 12 (Dec.), 64–70.

HURSON, A. R. 1981. An associative backend for data base management. *IEEE Workshop on Computer Architecture for Pattern Analysis and Image Data Base Management*, pp. 225–230.

HURSON, A. R. 1986. VLSI time/space complexity

of an associative parallel join module. In *International Conference on Parallel Processing*, pp. 379–386.

HURSON, A. R. ET AL. 1989. Performance evaluation of an associate parallel join module. *Comput. Syst. Sci. Eng. 4*, 3 (July), 131–146.

IBM 1978. IMS/VS General Information Manual GH20-1260. White Plains, New York

INTEL CORPORATION. *iDBP DBMS Reference Manual*, Order No. 222100.

IOANNIDIS, Y. E., AND KANG, Y 1990. Randomized algorithms for optimizing large join queries. In *Proceedings of SIGMOD*, pp. 312–321

KAMBAYASHI, Y 1985 Processing cyclic queries. In *Query Processing in Database Systems*, W. Kim, D. S. Reiner, and D. S. Batory, Eds. Springer-Verlag, New York, pp. 62–78.

KANG, H., AND ROUSSOPOULOS, N. 1987a. Using 2-way semijoins in distributed query processing In *Proceedings of Conference on Data Engineering*, pp. 644–651.

KANG, H., AND ROUSSOPULOS, N. 1987b. On the cost-effectiveness of a semijoin in query processing. In *COMPSAC*, pp. 531–537.

KENT, W 1983. A simple guide to five normal forms in relational database theory. Commun. *ACM 26*, 2 (Feb.).

KIM, W. 1980. A new way to compute the product and join of relation. In *Proceedings of SIGMOD*, pp. 179–187.

KIM, W. 1989. A model of queries for object-oriented databases In *Proceedings of the 15th International Conference on Very Large Databases* (Amsterdam), pp. 423–432.

KIM, W., REINER, D. S., AND BATORY, D. S. 1985. *Query Processing in Database Systems*. Springer-Verlag, New York.

KITSUREGAWA, M., TANAKA, H., AND MOTO-OKA, T. 1983. Application of hash to database machine and its architecture. *New Generation Comput. 1*, 1.

KITSUREGAWA, M., NAKANO, M., AND TAKAGI, M. 1989a. Query execution for large relations on functional disk system. In *Proceedings of Conference on Data Engineering*, pp. 159–167.

KITSUREGAWA, M., HARADA, L , AND TAKAGI, M 1989b. Join strategies on Kd-tree indexed relations. In *Proceedings of Conference on Data Engineering*, pp. 85–93.

KITSUREGAWA, M., NAKAYAMA, M., AND TAKAGI, M. 1989c The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proceedings of Conference on Very Large Data Bases*, pp. 257–266.

KITSUREGAWA, M., AND OGAWA, Y. 1990. Bucket spreading parallel hash: A new, robust, parallel hash join method. In *Proceedings of Conference on Very Large Data Bases*, pp. 210–221.

KUMAR, A., AND STONEBRAKER, M. 1987. The effect of join selectivities on optimal nesting order. *SIGMOD Rec. 16*, 1 (Mar.), 28–41.

KUNG, H. T., AND LEHMAN, P. L. 1980. Systolic (VLSI) arrays for relational database operations. In *Proceedings of SIGMOD*, pp. 105–116.

LAKSHMI, M. S., AND YU, P. S. 1988 Effect of skew on join performance in parallel architectures In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, pp. 107–117.

LAKSHMI, M. S , AND YU, P. S. 1989 Limiting factors of join performance on parallel processors. In *Proceedings of Conference on Data Engineering*, pp. 488–496.

LAKSHMI, M. S., AND YU, P. S. 1990. Effectiveness of parallel joins. *IEEE Trans. Know. Data Eng. 2*, 4 (Dec.), 410–424.

LEHMAN, T., AND CAREY, M. 1986. Query processing in main memory database systems. In *Proceedings of SIGMOD*, pp 239–250.

LIPTON, R. J., NAUGHTON, J. F., AND SCHNEIDER, A. D. 1990. Practical selectivity estimation through adaptive sampling In *Proceedings of SIGMOD*, pp. 1–11.

LU, H , AND CAREY, M. 1985. Some experimental results on distributed join algorithms in a local network. In *Proceedings of Conference on Very Large Databases*, pp 292–304.

MACKERT, L. F., AND LOHMAN, G. M. 1986. R* Optimizer: Validation and performance evaluation for distributed queries. In *Proceedings of Conference on Very Large Data Bases*, pp. 149–159.

MAIER, D. 1983. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md

MASUYAMA, S., IBARAKI, T., NISHIO, S , AND HASEGAWA, T. 1987 Shortest semijoin schedule for a local area distributed database system. *IEEE Trans. Softw. Eng. SE-13*, 5 (May), 602–606.

MENEZES, B. L., THADANI, K., DALE, A. G., AND JENEVEIN, R 1987. Design of a HyperKYKLOS-based multiprocessor architecture for high-performance join operations. In *International Workshop on Database Machines*, pp. 74–87.

MIKKILINENI, K. P., AND SU, S. Y. W. 1988 An evaluation of relational join algorithms in a pipelined query processing environment. *IEEE Trans. Softw. Eng. 14*, 6 (June), 838–848.

MULLIN, J. K. 1990. Optimal semijoins for distributed database systems. *IEEE Trans. Softw. Eng. 16*, 5 (May), 558–560.

MURPHY, M. C., AND ROTEM, D. 1989a. Effective resource utilization for multiprocessor join execution. In *Proceedings of Conference on Very Large Data Bases*, pp. 67–76.

MURPHY, M. C., AND ROTEM, D. 1989b. Processor scheduling for multiprocessor joins. In *Proceedings of Conference on Data Engineering*, pp. 140–148.

NAKAYAMA, T., HIRAKAWA, M., AND ICHIKAWA, T.

1984. Architecture and algorithm for parallel execution of a join operation. In *Proceedings of Conference on Data Engineering*, pp 160–166.

NAKAYAMA, M., KITSUREGAWA, M., AND TAKAGI, M. 1988. Hash-partitioned join method using dynamic destaging strategy. *Proceedings of Conference on Very Large Databases*, pp. 468–478.

OMIECINSKI, E R. 1989. Heuristics for join processing using nonclustered indexes. In *IEEE Trans. Softw. Eng. 15*, 1 (Jan.), 18–25.

OMIECINSKI, E., AND SHONKWILER, R. 1990. Parallel join processing using nonclustered indexes for a shared memory environment. In *IEEE Symposium on Parallel and Distributed Processing*, pp. 144–159.

OSBORN, S. L., AND HEAVEN, T. E. 1986. The design of a relational database system. *ACM Trans. Database Syst. 11*, 3 (Sept.), 357–373.

OZKARAHAN, E. A. 1986. *Database Machines and Database Management*. Prentice-Hall, Englewood Cliffs, N.J.

OZKARAHAN, E. A., AND BOZSAHIN, H. 1988. Join strategies using data space partitioning. *New Generation Comput. 1* 6, 19–39.

OZSOYOGLU, G., MATOS, V., AND OZSOYOGLU, Z. M. 1989. Query processing techniques in the summary-table-by-example database query. *ACM Trans. Database Syst. 14*, 4 (Dec.), 526–573.

PERRIZO, W., LIN, J. Y. Y., AND HOFFMAN, W. 1989. Algorithms for distributed query processing in broadcast local area networks. *IEEE Trans. Know. Data Eng. 1*, 2 (June), 215–225.

PIATETSKY-SHAPIRO, G., AND CONNELL, C. 1984. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of SIGMOD*, pp. 256–276.

PRAMANIK, S. 1986. Performance analysis of a database filter search hardware. *IEEE Trans. Comput. C-35*, 12 (Dec.), 1077–1082.

PRAMANIK, S., AND FOTOUHI, F. 1985. An index database machine: An efficient m-way join processor. In *Proceedings of Hawaii International Conference on System Sciences*, vol. 1, pp. 330–338.

PRAMANIK, S., AND ITTNER, D. 1985. Use of graph-theoretic models for optimal relational database accesses to perform joins. *ACM Trans. Database Syst. 10*, 1 (Mar.), 57–74.

PRAMANIK, S., AND VINEYARD, D. 1988. Optimizing join queries in distributed databases. *IEEE Trans. Softw. Eng. 14*, 9 (Sept.), 1319–1326.

PUCHERAL, P., THEVENIN, J. M., AND VALDURIEZ, P. 1990. Efficient main-memory data management using the DBGraph model. In *Proceedings of Conference on Very Large Data Bases*.

RASCHID, L. ET AL. 1986. A special-function unit for sorting and sort-based database operations. *IEEE Trans. Comput. C-35*, 12 (Dec.), 1071–1077.

RICHARDSON, J. P., LU, H., AND MIKKILINENI, K. 1987. Design and evaluation of parallel pipelined join algorithms. In *Proceedings of SIGMOD*, pp. 399–409.

ROBINSON, J. T. 1981. The KDB tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of SIGMOD*, pp 10–18.

ROSENTHAL, A., AND GALINDO-LEGARIA, C. 1990. Query graphs, implementing trees, and freely reorderable outerjoins. In *Proceedings of SIGMOD*, pp. 291–299.

ROSENTHAL, A. AND REINER, D. 1984. Extending the algebraic framework of query processing to handle outerjoins. In *Proceedings of Conference on Very Large Data Bases*, pp. 334–343

RUDOLPH, J. A. 1972a. A production implementation of an associative array processor. In *Proceedings of Fall Joint Computer Conference*, pp. 229–241.

SACCO, G. M. 1984. Distributed query evaluation in local area networks. In *Proceedings of Conference on Data Engineering*, pp. 510–516.

SACCO, G. M., AND SCHKOLNICK, M. 1986. Buffer management in relational database systems. *ACM Trans. Database Syst. 11*, 4 (Dec.), 473–498.

SAKAI, H., IWATA, K., KAMIYA, S., ABE, M., TANAKA, A., SHIBAYAMA, S., AND MURAKAMI, K. 1984. Design and implementation of the relational database engine. In *Proceedings of Conference on Fifth Generation Computer Systems*, pp. 419–435.

SCHNEIDER, D. A., AND DEWITT, D. J. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of SIGMOD*, pp. 110–121.

SEGEV, A. 1986. Optimization of join operations in horizontally partitioned database systems. *ACM Trans. Database Syst. 11*, 1 (Mar.), 48–80.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN,. D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management. In *Proceedings of SIGMOD*.

SHAPIRO, L. 1986. Join processing in database systems with large memories. *ACM Trans. Database Syst. 11*, 3 (Sept.), 239–264.

SHEKITA, E. J., AND CAREY, M. J. 1990. A performance evaluation of pointer-based joins. In *Proceedings of 1990 ACM SIGMOD Conference* (May), pp. 300–311.

STONEBRAKER, M., AND ROWE, L. A. 1986. The postgres papers. Univ. of California at Berkeley Tech. Rep. UCB/ERL M86/85.

SU, S. Y. W. 1988. *Database Computers: Principles, Architectures, and Techniques*, McGraw-Hill, New York.

SWAMI, A., AND GUPTA, A. 1988. Optimizing large join queries. In *Proceedings of SIGMOD*, pp. 8–17.

TANDEM DATABASE GROUP 1987. NonStop SQL: A distributed, high-performance, high-reliability,

implementation of SQL. In *Workshop on High Performance Transaction Systems* (Asilomar, Calif)

TONG, F., AND YAO, S. B., 1982. Performance analysis of database join processors In *National Computer Conference*, pp. 627-637.

ULLMAN, J. D. 1988 *Principles of Database and Knowledge-Base Systems*, vol. 1, Computer Science Press, Rockville, Md

VALDURIEZ, P. 1982. Semijoin algorithms for distributed databases. In *3rd International Symposium on Distributed Databases*.

VALDURIEZ, P 1986 Optimization of complex database queries using join indices. *Database Eng. 9*, 4 (Dec.), 10-16.

VALDURIEZ, P. 1987. Join indices *ACM Trans. Database Syst. 12*, 2 (June), 218-246.

VALDURIEZ, P., AND BORAL, H. 1986. Evaluation of recursive queries using join indices. In *Proceedings of Conference on Expert Database Systems*.

VALDURIEZ, P , AND GARDARIN, G. 1982. Multiprocessor join algorithms of relations In *Improving Usability and Responsiveness*. Academic Press, New York, pp. 219-237

VALDURIEZ, P , AND GARDARIN, G. 1984 Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst. 9*, 1 (Mar.), 133-161.

VALDURIEZ, P., AND VIEMONT, Y. 1984. A new multikey hashing scheme using predicate trees. In *Proceedings of SIGMOD*.

VALDURIEZ, P , KHOSHAFIAN, S., AND COPELAND, G. 1986. Implementation techniques of complex objects. In *Proceedings of Conference on Very Large Data Bases*.

WALTON, C B 1989. Investigating skew and scalability in parallel joins. Tech. Rep. TR-89-39, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, Tx.

WANG, X., AND LUK, W. S. 1988. Parallel join algorithms on a network of workstations. In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, pp. 87-96.

WHANG, K.-Y., WIEDERHOLD, G., AND SAGALOWICZ, D. 1985. The property of separability and its application to physical database design. In *Query Processing in Database Systems*, W Kim, D. S Reiner, and D S. Batory, Eds. Springer-Verlag, New York, pp. 297-317.

YAO, S B., TONG, F., AND SHENG Y. Z. 1981 The system architecture of a data base machine (DBM). *IEEE Database Eng. Bull. 4* 2, 53-62.

YOO, H., AND LAFORTUNE, S. 1989. An intelligent search method for query optimization by semijoins. *IEEE Trans. Knowl. Data Eng. 1*, 2 (June), 226-237.

YOSHIKAWA, M., AND KAMBAYASHI, Y. 1984 Processing inequality queries based on generalized semi-joins. In *Proceedings of Conference on Very Large Data Bases*, pp 416-428.

YU, C. T., CHANG, C. C., TEMPLETON, M., BRILL, D., AND LUND, E. 1985. Query processing in a fragmented relational database system: Mermaid. *IEEE Trans. Softw. Eng. SE-11* 8 (Aug.), 795-810.

YU, C. T., GUH, K.-C., ZHANG, W., TEMPLETON, M., BRILL, D., AND CHEN, A L. P. 1987. Algorithms to process distributed queries in fast local networks. *IEEE Trans. Comput. C-36*, 10 (Oct.), 1153-1164.

ZELLER, H., AND GRAY, J. 1990. An adaptive hash join algorithm for multiuser environments. In *Proceedings of Conference on Very Large Data Bases*, pp. 186-197.

BIBLIOGRAPHY

BANCILHON, F., AND SCHOLL, M. 1980. Design of a backend processor for a database machine. In *Proceedings of SIGMOD*.

BANERJEE, J., AND HSIAO, D K 1979 Concepts and capabilities of a database computer. *ACM Trans. Database Syst. 4*, 1 (Mar.).

BROWNSMITH, J. D. 1981. A simulation model of the MICRONET computer system during join processing. In *Proceedings of the Annual Simulation Symposium*, pp. 1-16.

BROWNSMITH, J D., AND SU, S. Y. W. 1980. Performance analysis of the equijoin operation in the MICRONET computer system In *Proceedings of ICC*, pp. 264-268.

CHASE, K. 1981. Join graphs and acyclic database schemes. In *Proceedings of Conference on Very Large Data Bases*, pp 95-100

CHIU, C. M , AND HO, Y. C 1980 A methodology for interpreting tree queries into optimal semi-join expressions. In *Proceedings of SIGMOD*, pp. 169-178.

CIACCIA, P., AND SCALAS, M R 1989. Optimization strategies for relational disjunctive queries. *IEEE Trans Softw. Eng. 15*, 10 (Oct.), 1217-1235.

CODD, E. F. 1979. Extending the data base relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4, 397-434.

DEWITT, D. J. 1979. DIRECT: A multiprocessor organization for supporting relational database management systems. *IEEE Trans. Comput. C-28*, 6, 395-406.

DEWITT, D. J., NAUGHTON, J. F., AND SCHNEIDER, D. A. 1991. An evaluation of non-equijoin algorithms. Tech. Rep. 1011, Univ. of Wisconsin-Madison, Madison, Wisc.

GARDY, D , AND PUECH, C 1989 On the effect of join operations on relation sizes. *ACM Trans. Database Syst. 14*, 4 (Dec.), 574-603.

GOTLIEB, L. R 1975. Computing joins of relations. In *Proceedings of SIGMOD*, pp. 55-63.

GRAEFE, G. 1990. Encapsulation of parallelism in

the Volcano query processing systems. In *Proceedings of SIGMOD*, 102-111.

GRAEFE, G. 1991. Heap-filter merge join: A new algorithm for joining medium-size inputs. *IEEE Trans. Softw. Eng. 17*, 9 (Sept.), 979-982.

HONEYMAN, P. 1980. Extension joins. In *Proceedings of Conferences on Very Large Data Bases*, pp. 239-244.

HONG, Y. C. 1984. A pipeline and parallel architecture for supporting database management systems. In *Proceedings of Conference on Data Engineering*, pp. 152-159.

KAMIBAYASHI, N., AND SEO, K. 1982. SPIRIT-III: An advanced relational database machine introducing a novel data staging architecture with tuple stream filters to preprocess relational algebra. In *National Computer Conference Proceedings*, pp. 605-616.

KELLER, A. 1985. Algorithms for translating view updates into database updates for views involving select. In *Proceedings of ACM Symposium on Principles of Database Systems*, pp. 154-163.

KENT, W. 1979. The entity join. In *Proceedings of Conference on Very large Data Bases*, pp. 232-238.

LACROIX, M., AND PIROTTE, A. 1976. Generalized joins. *SIGMOD Rec. 8*, 3, 14-15.

LU, H., TAN, K. L., AND SHAN, M.-C. 1990. Hash-based join algorithms for multiprocessor computers with shared memories. In *Proceedings of Conference on Very Large Data Bases*, pp. 198-209.

MAIER, D., SAGIV, Y., AND YANNAKIS, M. 1981. On the complexity of testing implications of functional and join dependencies. *J. ACM 28*, 4, 680-695.

MENON, M. J. AND HSIAO, D. K. 1983. Design and analysis of join operations of database machines. In *Advanced Database Machine Architecture*, D. K. Hsiao, Ed. Prentice-Hall, Englewood Cliffs, N.J., pp. 203-255.

MERRET, T. H. 1983. Why sort-merge gives the best implementation of the natural joins. *ACM SIGMOD Rec. 13*, 2 (Jan.), 39-51.

MERRET, T. H. 1984. Practical hardware for linear execution of relational database operations. *ACM SIGMOD Rec. 14*, 1 (Mar.) 39-44.

MERRETT, T, H., KAMBAYASHI, Y., AND YASUURA, H. 1981. Scheduling of page fetches in join operations. In *Proceedings of Conference on Very Large Data Bases*, pp. 488-497.

OMIECINSKI, E. R., AND LIN, E. T. 1989. Hash-based and index-based join algorithms for cube and ring connected multicomputers. *IEEE Trans. Knowl. Data Eng. 1*, 3 (Sept.), 329-343.

ONO, K., AND LOHMAN, G. M. 1990. Measuring the complexity of join enumeration in query optimization. In *Proceedings of Conference on Very Large Data Bases*, pp. 314-325.

QADAH, G. Z. 1984. Evaluation of performance of the equi-join operation on the Michigan relational database machine. In *Proceedings of Conference on Parallel Processing*, pp. 260-265.

QADAH, G. Z. 1985. The equijoin operation on a multiprocessor database machine. In *Proceedings of International Workshop on Database Machines*, Springer-Verlag, New York, pp. 35-67.

QADAH, G. Z., AND IRANI, K. B. 1985. A database machine for very large databases. *IEEE Trans. Comput. C-34*, 11, 1015-1025.

QADAH, G. Z., AND IRANI, K. B. 1988. The join algorithm on a shared-memory multiprocessor database machine. *IEEE Trans. Softw. Eng. 14*, 11 (Nov.), 1668-1683.

RISSANEN, J. 1979. Theory of joins for relational databases: A tutorial survey. In *Proceedings of Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 64. Springer-Verlag, New York, pp. 537-551.

ROSENTHAL, A. 1981. Note on the expected size of a join. *ACM SIGMOD Rec. 11*, 4 (July), 19-25.

SCHNEIDER, D. A., AND DEWITT, D. J. 1990. Trade-offs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of Conference on Very Large Data Bases*, pp. 469-480.

SCHUSTER, S. A., NGUYEN, H. B., OZKARAHAN, E. A., AND SMITH, K. C. 1979. RAP.2: An associative processor for databases and its applications. *IEEE Trans. Comput. C-28*, 6, 446-458.

SCIORE, E. 1982. A complete axiomatization for full join dependencies. *J. ACM 29*, 2 (Apr.), 373-393.

SHAW, D. E. ET AL. 1981. The NON-VON database machine: A brief overview. *Database Eng. 4*, 2

SHULTZ, R., AND MILLER, I. 1987. Tree structured multiple processor join methods. In *Proceedings of Conference on Data Engineering*, pp. 190-199.

SU, S. Y. W., NGUYEN, L. H., EMAN, A., AND LIPOVSKI, G. J. 1979. The architectural features and implementation techniques of multicell CASSM. *IEEE Trans. Comput. C-28*, 6 (June), 430-445.

THOM, J. A., RAMAMOHANARAO, K., AND NAISH, L. 1986. A superjoin algorithm for deductive databases. In *Proceedings of Conference on Very Large Databases*, pp. 189-196.

VARDI, M. Y. 1980. Axiomatization of functional and join dependencies in the relational model. Weizman Institute M.Sc. thesis, Rehovot, Israel.

VARDI, M. Y. 1983. Inferring multivalued dependencies from functional and join dependencies. *Acta Inf. 19*, 2, 305-324.

Received April 1990, final revision accepted July 1991.