Query Evaluation Techniques for Large Databases

GOETZ GRAEFE

Portland State University, Computer Science Department, P.O. Box 751, Portland, Oregon 97207–0751

Database management systems will continue to manage large data volumes. Thus, efficient algorithms for accessing and manipulating large sets and sequences will be required to provide acceptable performance. The advent of object-oriented and extensible database systems will not solve this problem. On the contrary, modern data models exacerbate the problem: In order to manipulate large sets of complex objects as efficiently as today's database systems manipulate simple records, query processing algorithms and software will become more complex, and a solid understanding of algorithm and architectural issues is essential for the designer of database management software.

This survey provides a foundation for the design and implementation of query execution facilities in new database management systems. It describes a wide array of practical query evaluation techniques for both relational and postrelational database systems, including iterative execution of complex query evaluation plans, the duality of sort- and hash-based set-matching algorithms, types of parallel query execution and their implementation, and special operators for emerging database application domains.

Categories and Subject Descriptors: E.5 [**Data**]: Files; H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Complex query evaluation plans, dynamic query evaluation plans; extensible database systems, iterators, object-oriented database systems, operator model of parallelization, parallel algorithms, relational database systems, set-matching algorithms, sort-hash duality

INTRODUCTION

Effective and efficient management of large data volumes is necessary in virtually all computer applications, from business data processing to library information retrieval systems, multimedia applications with images and sound, computer-aided design and manufacturing, real-time process control, and scientific computation. While database management systems are standard tools in business data processing, they are only slowly being introduced to all the other emerging database application areas.

In most of these new application domains, database management systems have traditionally not been used for two reasons. First, restrictive data definition and manipulation languages can make application development and maintenance unbearably cumbersome. Research into semantic and object-oriented data models and into persistent database programming languages has been addressing this problem and will eventually lead to acceptable solutions. Second, data vol-

CONTENTS

INTRODUCTION
1 ARCHITECTURE OF QUERY EXECUTION ENGINES
2. SORTING AND HASHING
  2.1 Sorting
  2.2 Hashing
3. DISK ACCESS
  3 1 File Scans
  3 2 Associative Access Using Indices
  3.3 Buffer Management
4. AGGREGATION AND DUPLICATE REMOVAL
  4 1 Aggregation Algorithms Based on Nested Loops
  4.2 Aggregation Algorithms Based on Sorting
  4 3 Aggregation Algorithms Based on Hashing
  4 4 A Rough Performance Comparison
  4 5 Additional Remarks on Aggregation
5 BINARY MATCHING OPERATIONS
  5 1 Nested-Loops Join Algorithms
  5 2 Merge-Join Algorithms
  5 3 Hash Join Algorithms
  5 4 Pointer-Based Joins
  5 5 A Rough Performance Comparison
6 UNIVERSAL QUANTIFICATION
7 DUALITY OF SORT- AND HASH-BASED QUERY PROCESSING ALGORITHMS
8 EXECUTION OF COMPLEX QUERY PLANS
9 MECHANISMS FOR PARALLEL QUERY EXECUTION
  9 1 Parallel versus Distributed Database Systems
  9 2 Forms of Parallelism
  9.3 Implementation Strategies
  9 4 Load Balancing and Skew
  9 5 Architectures and Architecture Independence
10 PARALLEL ALGORITHMS
  10.1 Parallel Selections and Updates
  10 2 Parallel Sorting
  10 3 Parallel Aggregation and Duplicate Removal
  10.4 Parallel Joins and Other Binary Matching Operations
  10 5 Parallel Universal Quantification
11 NONSTANDARD QUERY PROCESSING ALGORITHMS
  11 1 Nested Relations
  11 2 Temporal and Scientific Database Management
  11.3 Object-Oriented Database Systems
  11 4 More Control Operators
12. ADDITIONAL TECHNIQUES FOR PERFORMANCE IMPROVEMENT
  12 1 Precomputation and Derived Data
  12 2 Data Compression
  12.3 Surrogate Processing
  12 4 Bit Vector Filtering
  12.5 Specialized Hardware
SUMMARY AND OUTLOOK
ACKNOWLEDGMENTS
REFERENCES

umes might be so large or complex that the real or perceived performance advantage of file systems is considered more important than all other criteria, e.g., the higher levels of abstraction and programmer productivity typically achieved with database management systems. Thus, object-oriented database management systems that are designed for nontraditional database application domains and extensible database management system toolkits that support a variety of data models must provide excellent performance to meet the challenges of very large data volumes, and techniques for manipulating large data sets will find renewed and increased interest in the database community.

The purpose of this paper is to survey efficient algorithms and software architectures of database query execution engines for executing complex queries over large databases. A "complex" query is one that requires a number of query-processing algorithms to work together, and a "large" database uses files with sizes from several megabytes to many terabytes, which are typical for database applications at present and in the near future [Dozier 1992; Silberschatz et al. 1991]. This survey discusses a large variety of query execution techniques that must be considered when designing and implementing the query execution module of a new database management system: algorithms and their execution costs, sorting versus hashing, parallelism, resource allocation and scheduling issues in complex queries, special operations for emerging database application domains such as statistical and scientific databases, and general performance-enhancing techniques such as precomputation and compression. While many, although not all, techniques discussed in this paper have been developed in the context of relational database systems, most of them are applicable to and useful in the query processing facility for any database management system and any data model, provided the data model permits queries over "bulk" data types such as sets and lists.

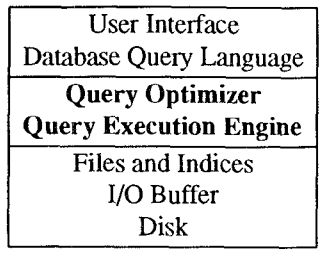


Figure 1. Query processing in a database system.

It is assumed that the reader possesses basic textbook knowledge of database query languages, in particular of relational algebra, and of file systems, including some basic knowledge of index structures. As shown in Figure 1, query processing fills the gap between database query languages and file systems. It can be divided into query optimization and query execution. A query optimizer translates a query expressed in a high-level query language into a sequence of operations that are implemented in the query execution engine or the file system. The goal of query optimization is to find a query evaluation plan that minimizes the most relevant performance measure, which can be the database user's wait for the first or last result item, CPU, I/O, and network time and effort (time and effort can differ due to parallelism), memory costs (as maximum allocation or as time-space product), total resource usage, even energy consumption (e.g., for battery-powered laptop systems or space craft), a combination of the above, or some other performance measure. Query optimization is a special form of planning, employing techniques from artificial intelligence such as plan representation, search including directed search and pruning, dynamic programming, branch-and-bound algorithms, etc. The query execution engine is a collection of query execution operators and mechanisms for operator communication and synchronization—it employs concepts from algorithm design, operating systems, networks, and parallel and distributed computation. The facilities of the query execution engine define the space of possible plans that can be chosen by the query optimizer.

A general outline of the steps required for processing a database query are shown in Figure 2. Of course, this sequence is only a general guideline, and different database systems may use different steps or merge multiple steps into one. After a query or request has been entered into the database system, be it interactively or by an application program, the query is parsed into an internal form. Next, the query is validated against the metadata (data about the data, also called schema or catalogs) to ensure that the query contains only valid references to existing database objects. If the database system provides a macro facility such as relational views, referenced macros and views are expanded into the query [Stonebraker 1975]. Integrity constraints might be expressed as views (externally or internally) and would also be integrated into the query at this point in most systems [Motro 1989]. The query optimizer then maps the expanded query expression into an optimized plan that operates directly on the stored database objects. This mapping process can be very complex and might require substantial search and cost estimation effort. (Optimization is not discussed in this paper; a survey can be found in Jarke and Koch [1984].) The optimizer's output is called a query execution plan, query evaluation plan, QEP, or simply plan. Using a simple tree traversal algorithm, this plan is translated into a representation ready for execution by the database's query execution engine; the result of this translation can be compiled machine code or a semicompiled or interpreted language or data structure.

This survey discusses only read-only queries explicitly; however, most of the techniques are also applicable to update requests. In most database management systems, update requests may include a search predicate to determine the database objects are to be modified. Standard query optimization and execution techniques apply to this search; the actual update procedure can be either ap-

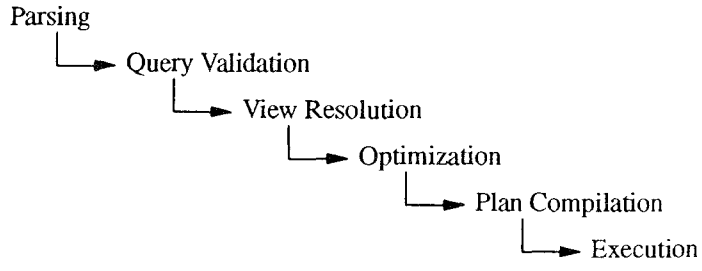


Figure 2. Query processing steps.

plied in a second phase, a method called *deferred updates*, or merged into the search phase if there is no danger of creating ambiguous update semantics.[1]

The problem of ensuring *ACID* semantics for updates—making updates *A*tomic (all-or-nothing semantics), *C*onsistent (translating any consistent database state into another consistent database state), *I*solated (from other queries and requests), and *D*urable (persistent across all failures)—is beyond the scope of this paper; suitable techniques have been described by many other authors, e.g., Bernstein and Goodman [1981], Bernstein et al. [1987], Gray and Reuter [1991], and Haerder and Reuter [1983].

Most research into providing ACID semantics focuses on efficient techniques for processing very large numbers of relatively small requests. For example, increasing the balance of one account and decreasing the balance of another account require exclusive access to only two database records and writing some information to an update log. Current research and development efforts in transaction processing target hundreds and even thousands of small transactions per second [Davis 1992; Serlin 1991].

[1] A standard example for this danger is the "Halloween" problem: Consider the request to "give all employees with salaries greater than $30,000 a 3% raise." If (i) these employees are found using an index on salaries, (ii) index entries are scanned in increasing salary order, and (iii) the index is updated immediately as index entries are found, then each qualifying employee will get an infinite number of raises.

Query processing, on the other hand, focuses on extracting information from a large amount of data without actually changing the database. For example, printing reports for each branch office with average salaries of employees under 30 years old requires shared access to a large number of records. Mixed requests are also possible, e.g., for crediting monthly earnings to a stock account by combining information about a number of sales transactions. The techniques discussed here apply to the search effort for such a mixed request, e.g., for finding the relevant sales transactions for each stock account.

Embedded queries, i.e., database queries that are contained in an application program written in a standard programming language such as Cobol, PL/1, C, or Fortran, are also not addressed specifically in this paper because all techniques discussed here can be used for interactive as well as embedded queries. Embedded queries usually are optimized when the program is compiled, in order to avoid the optimization overhead when the program runs. This method was pioneered in System R, including mechanisms for storing optimized plans and invalidating stored plans when they become infeasible, e.g., when an index is dropped from the database [Chamberlin et al. 1981b]. Of course, the cut between compile-time and run-time can be placed at any other point in the sequence in Figure 2.

Recursive queries are omitted from this survey, because the entire field of recursive query processing—optimization rules and heuristics, selectivity and cost

estimation, algorithms and their parallelization—is still developing rapidly (suffice it to point to two recent surveys by Bancilhon and Ramakrishnan [1986] and Cacace et al. [1993]).

The present paper surveys query execution techniques; other surveys that pertain to the wide subject of database systems have considered data models and query languages [Gallaire et al. 1984; Hull and King 1987; Jarke and Vassiliou 1985; McKenzie and Snodgrass 1991; Peckham and Maryanski 1988], access methods [Comer 1979; Enbody and Du 1988; Faloutsos 1985; Samet 1984; Sockut and Goldberg 1979], compression techniques [Bell et al. 1989; Lelewer and Hirschberg 1987], distributed and heterogeneous systems [Batini et al. 1986; Litwin et al. 1990; Sheth and Larson 1990; Thomas et al. 1990], concurrency control and recovery [Barghouti and Kaiser 1991; Bernstein and Goodman 1981; Gray et al. 1981; Haerder and Reuter 1983; Knapp 1987], availability and reliability [Davidson et al. 1985; Kim 1984], query optimization [Jarke and Koch 1984; Mannino et al. 1988; Yu and Chang 1984], and a variety of other database-related topics [Adam and Wortmann 1989; Atkinson and Bunemann 1987; Katz 1990; Kemper and Wallrath 1987; Lyytinen 1987; Teoroy et al. 1986]. Bitton et al. [1984] have discussed a number of parallel-sorting techniques, only a few of which are really used in database systems. Mishra and Eich's [1992] recent survey of relational join algorithms compares their behavior using diagrams derived from one by Kitsuregawa et al. [1983] and also describes join methods using index structures and join methods for distributed systems. The present survey is much broader in scope as it also considers system architectures for complex query plans and for parallel execution, selection and aggregation algorithms, the relationship of sorting and hashing as it pertains to database query processing, special operations for nontraditional data models, and auxiliary techniques such as compression.

Section 1 discusses the architecture of query execution engines. Sorting and hashing, the two general approaches to managing and matching elements of large sets, are described in Section 2. Section 3 focuses on accessing large data sets on disk. Section 4 begins the discussion of actual data manipulation methods with algorithms for aggregation and duplicate removal, continued in Section 5 with binary matching operations such as join and intersection and in Section 6 with operations for universal quantification. Section 7 reviews the many dualities between sorting and hashing and points out their differences that have an impact on the performance of algorithms based on either one of these approaches. Execution of very complex query plans with many operators and with nontrivial plan shapes is discussed in Section 8. Section 9 is devoted to mechanisms for parallel execution, including architectural issues and load balancing, and Section 10 discusses specific parallel algorithms. Section 11 outlines some nonstandard operators for emerging database applications such as statistical and scientific database management systems. Section 12 is a potpourri of additional techniques that enhance the performance of many algorithms, e.g., compression, precomputation, and specialized hardware. The final section contains a brief summary and an outlook on query processing research and its future.

For readers who are more interested in some topics than others, most sections are fairly self-contained. Moreover, the hurried reader may want to skip the derivation of cost functions;[2] their results and effects are summarized later in diagrams.

1. ARCHITECTURE OF QUERY EXECUTION ENGINES

This survey focuses on useful mechanisms for processing sets of items. These items can be records, tuples, entities, or objects. Furthermore, most of the tech-

[2] In any case, our cost functions cover only a limited, though important, aspect of query execution cost, namely I/O effort.

niques discussed in this survey apply to sequences, not only sets, of items, although most query processing research has assumed relations and sets. All query processing algorithm implementations iterate over the members of their input sets; thus, sets are always represented by sequences. Sequences can be used to represent not only sets but also other one-dimensional "bulk" types such as lists, arrays, and time series, and many database query processing algorithms and techniques can be used to manipulate these other bulk types as well as sets. The important point is to think of these algorithms as algebra operators consuming zero or more inputs (sets or sequences) and producing one (or sometimes more) outputs. A complete query execution engine consists of a collection of operators and mechanisms to execute complex expressions using multiple operators, including multiple occurrences of the same operator. Taken as a whole, the query processing algorithms form an algebra which we call the *physical algebra* of a database system.

The physical algebra is equivalent to, but quite different from, the *logical algebra* of the data model or the database system. The logical algebra is more closely related to the data model and defines what queries can be expressed in the data model; for example, the relational algebra is a logical algebra. A physical algebra, on the other hand, is system specific. Different systems may implement the same data model and the same logical algebra but may use very different physical algebras. For example, while one relational system may use only nested-loops joins, another system may provide both nested-loops join and merge-join, while a third one may rely entirely on hash join algorithms. (Join algorithms are discussed in detail later in the section on binary matching operators and algorithms.)

Another significant difference between logical and physical algebras is the fact that specific algorithms and therefore cost functions are associated only with physical operators, not with logical algebra operators. Because of the lack of an algorithm specification, a logical algebra expression is not directly executable and must be mapped into a physical algebra expression. For example, it is impossible to determine the execution time for the left expression in Figure 3, i.e., a logical algebra expression, without mapping it first into a physical algebra expression such as the query evaluation plan on the right of Figure 3. This mapping process can be trivial in some database systems but usually is fairly complex in real database systems because it involves algorithm choices and because logical and physical operators frequently do not map directly into one another, as shown in the following four examples. First, some operators in the physical algebra may implement multiple logical operators. For example, all serious implementations of relational join algorithms include a facility to output fewer than all attributes, i.e., a relational delta-project (a projection without duplicate removal) is included in the physical join operator. Second, some physical operators implement only part of a logical operator. For example, a duplicate removal algorithm implements only the "second half" of a relational projection operator. Third, some physical operators do not exist in the logical algebra. Concretely, a sort operator has no place in pure relational algebra because it is an algebra of sets, and sets are, by their definition, unordered. Finally, some properties that hold for logical operators do not hold, or only with some qualifications, for the counterparts in physical algebra. For example, while intersection and union are entirely symmetric and commutative, algorithms implementing them (e.g., nested loops or hybrid hash join) do not treat their two inputs equally.

The difference of logical and physical algebras can also be looked at in a different way. Any database system raises the level of abstraction above files and records; to do so, there are some logical type constructors such as tuple, relation, set, list, array, pointer, etc. Each logical type constructor is complemented by

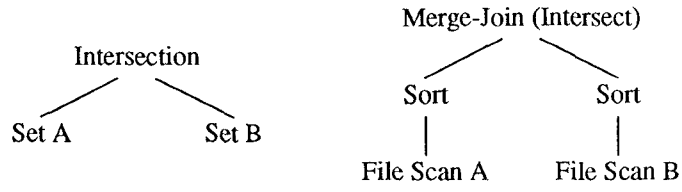


Figure 3. Logical and physical algebra expressions.

some operations that are permitted on instances of such types, e.g., attribute extraction, selection, insertion, deletion, etc.

On the physical or representation level, there is typically a smaller set of representation types and structures, e.g., file, record, record identifier (RID), and maybe very large byte arrays [Carey et al. 1986]. For manipulation, the representation types have their own operations, which will be different from the operations on logical types. Multiple logical types and type constructors can be mapped to the same physical concept. They may also be situations in which one logical type constructor can be mapped to multiple physical concepts, e.g., a set depending on its size. The mapping from logical types to physical representation types and structures is called physical database design. Query optimization is the mapping from logical to physical operations, and the query execution engine is the implementation of operations on physical representation types and of mechanisms for coordination and cooperation among multiple such operations in complex queries. The policies for using these mechanisms are part of the query optimizer.

Synchronization and data transfer between operators is the main issue to be addressed in the architecture of the query execution engine. Imagine a query with two joins, and consider how the result of the first join is passed to the second one. The simplest method is to create (write) and read a temporary file. The need for temporary files, whether they are kept in the buffer or not, is a direct result of executing an operator's input subplans completely before starting the operator. Alternatively, it is possible to create one process for each operator and then to use interprocess communication mechanisms (e.g., pipes) to transfer data between operators, leaving it to the operating system to schedule and suspend operator processes as pipes are full or empty. While such data-driven execution removes the need for temporary disk files, it introduces another cost, that of operating system scheduling and interprocess communication. In order to avoid both temporary files and operating system scheduling, Freytag and Goodman [1989] proposed writing rule-based translation programs that transform a plan represented as a tree structure into a single iterative program with nested loops and other control structures. However, the required rule set is not simple, in particular for algorithms with complex control logic such as sorting, merge-join, or even hybrid hash join (to be discussed later in the section on matching).

The most practical alternative is to implement all operators in such a way that they *schedule each other within a single operating system process*. The basic idea is to define a granule, typically a single record, and to iterate over all granules comprising an intermediate query result.[3] Each time an operator needs another granule, it calls its input (operator) to produce one. This call is a simple pro-

[3] It is possible to use multiple granule sizes within a single query-processing system and to provide special operators with the sole purpose of translating from one granule size to another. An example is a query processing system that uses records as an iteration granule except for the inputs of merge-join (see later in the section on binary matching), for which it uses "value packets," i.e., groups of records with equal join attribute values.

cedure call, much cheaper than interprocess communication since it does not involve the operating system. The calling operator waits (just as any calling routine waits) until the input operator has produced an item. That input operator, in a complex query plan, might require an item from its own input to produce an item; in that case, it calls its own input (operator) to produce one. Two important features of operators implemented in this way are that they can be combined into arbitrarily complex query evaluation plans and that any number of operators can execute and schedule each other in a single process without assistance from or interaction with the underlying operating system. This model of operator implementation and scheduling resembles very closely those used in relational systems, e.g., System R (and later SQL/DS and DB2), Ingres, Informix, and Oracle, as well as in experimental systems, e.g., the E programming language used in EXODUS [Richardson and Carey 1987], Genesis [Batory et al. 1988a; 1988b], and Starburst [Haas et al. 1989; 1990]. Operators implemented in this model are called *iterators*, streams, synchronous pipelines, row-sources, or similar names in the "lingo" of commercial systems.

To make the implementation of operators a little easier, it makes sense to separate the functions (a) to prepare an operator for producing data, (b) to produce an item, and (c) to perform final housekeeping. In a file scan, these functions are called *open*, *next*, and *close* procedures; we adopt these names for all operators. Table 1 gives a rough idea of what the *open*, *next*, and *close* procedures for some operators do, as well as the principal local state that needs to be saved from one invocation to the next. (Later sections will discuss sort and join operations in detail.) The first three examples are trivial, but the *hash join* operator shows how an operator can schedule its inputs in a nontrivial manner. The interesting observations are that (i) the entire query plan is executed within a single process, (ii) operators produce one item at a time on request, (iii) this model effectively implements, within a single process, (special-purpose) *coroutines* and *demand-driven dataflow*, (iv) items never wait in a temporary file or buffer between operators because they are never produced before they are needed, (v) therefore this model is very efficient in its time-space-product memory costs, (vi) iterators can schedule any tree, including bushy trees (see below), (vii) no operator is affected by the complexity of the whole plan, i.e., this model of operator implementation and synchronization works for simple as well as very complex query plans. As a final remark, there are effective ways to combine the iterator model with parallel query processing, as will be discussed in Section 9.

Since query plans are algebra expressions, they can be represented as trees. Query plans can be divided into prototypical shapes, and query execution engines can be divided into groups according to which shapes of plans they can evaluate. Figure 4 shows prototypical left-deep, right-deep, and bushy plans for a join of four inputs. Left-deep and right-deep plans are different because join algorithms use their two inputs in different ways; for example, in the nested-loops join algorithm, the outer loop iterates over one input (usually drawn as left input) while the inner loop iterates over the other input. The set of bushy plans is the most general as it includes the sets of both left-deep and right-deep plans. These names are taken from Graefe and DeWitt [1987]; left-deep plans are also called "linear processing trees" [Krishnamurthy et al. 1986] or "plans with no composite inner" [Ono and Lohman 1990].

For queries with common subexpressions, the query evaluation plan is not a tree but an acyclic directed graph (DAG). Most systems that identify and exploit common subexpressions execute the plan equivalent to a common subexpression separately, saving the intermediate result in a temporary file to be scanned repeatedly and destroyed after the last

Table 1. Examples of Iterator Functions

Iterator	*Open*	*Next*	*Close*	Local State
Print	*open* input	call *next* on input; format the item on screen	*close* input	
Scan	open file	read next item	close file	open file descriptor
Select	*open* input	call *next* on input until an item qualifies	*close* input	
Hash join (without overflow resolution)	allocate hash directory; *open* left "build" input; build hash table calling *next* on build input; *close* build input; *open* right "probe" input	call *next* on probe input until a match is found	*close* probe input; deallocate hash directory	hash directory
Merge-Join (without duplicates)	*open* both inputs	get *next* item from input with smaller key until a match is found	*close* both inputs	
Sort	*open* input; build all initial run files calling *next* on input; *close* input; merge run files until only one merge step is left	determine next output item; read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

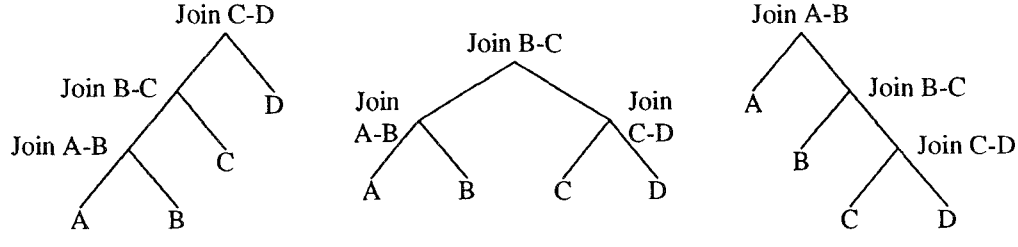


Figure 4. Left-deep, bushy, and right-deep plans.

scan. Each plan fragment that is executed as a unit is indeed a tree. The alternative is a "split" iterator that can deliver data to multiple consumers, i.e., that can be invoked as iterator by multiple consumer iterators. The split iterator paces its input subtree as fast as the fastest consumer requires it and holds items until the slowest consumer has consumed them. If the consumers request data at about the same rate, the split operator does not require a temporary spool file; such a file and its associated I/O cost are required only if the data rate required by the consumers diverges above some predefined threshold.

Among the implementations of iterators for query processing, one group can be called "stored-set oriented" and the other "algebra oriented." In System R, an example for the first group, complex join plans are constructed using binary join iterators that "attach" one more set (stored relation) to an existing intermediate result [Astrahan et al. 1976; Lorie and Nilsson 1979], a design that supports only left-deep plans. This design led to a significant simplification of the System R optimizer which could be based on dynamic programming techniques, but

it ignores the optimal plan for some queries [Selinger et al. 1979].[4] A similar design was used, although not strictly required by the design of the execution engine, in the Gamma database machine [DeWitt et al. 1986; 1990; Gerber 1986]. On the other hand, some systems use binary operators for which both inputs can be intermediate results, i.e., the output of arbitrarily complex subplans. This design is more general as it also permits bushy plans. Examples for this approach are the second query processing engine of Ingres based on Kooi's thesis [Kooi 1980; Kooi and Frankforth 1982], the Starburst execution engine [Haas et al. 1989], and the Volcano query execution engine [Graefe 1993b]. The tradeoff between left-deep and bushy query evaluation plans is reduction of the search space in the query optimizer against generality of the execution engine and efficiency for some queries. Right-deep plans have only recently received more interest and may actually turn out to be very efficient, in particular in systems with ample memory [Schneider 1990; Schneider and DeWitt 1990].

The remainder of this section provides more details of how iterators are implemented in the Volcano extensible query processing system. We use this system repeatedly as an example in this survey because it provides a large variety of mechanisms for database query processing, but mostly because its model of operator implementation and scheduling resembles very closely those used in many relational and extensible systems. The purpose of this section is to provide implementation concepts from which a new query processing engine could be derived.

Figure 5 shows how iterators are represented in Volcano. A box stands for a record structure in Volcano's implementation language (C [Kernighan and Ritchie 1978]), and an arrow represents a pointer. Each operator in a query evaluation plan consists of two record structures, a small structure of four points and a *state record*. The small structure is the same for all algorithms. It represents the stream or iterator abstraction and can be invoked with the *open*, *next*, and *close* procedures. The purpose of state records is similar to that of activation records allocated by compiler-generated code upon entry into a procedure. Both hold values local to the procedure or the iterator. Their main difference is that activation records reside on the stack and vanish upon procedure exit, while state records must persist from one invocation of the iterator to the next, e.g., from the invocation of *open* to each invocation of *next* and the invocation of *close*. Thus, state records do not reside on the stack but in heap space.

The type of state records is different for each iterator as it contains iterator-specific arguments and local variables (state) while the iterator is suspended, e.g., currently not active between invocations of the operator's *next* procedure. Query plan nodes are linked together by means of *input* pointers, which are also kept in the state records. Since pointers to functions are used extensively in this design, all operator code (i.e., the *open*, *next*, and *close* procedures) can be written in such a way that the names of input operators and their iterator procedures are not "hard-wired" into the code, and the operator modules do not need to be recompiled for each query. Furthermore, all operations on individual items, e.g., printing, are imported into Volcano operators as functions, making the operators independent of the semantics and representation of items in the data streams they are processing. This organization using function pointers for input operators is fairly standard in commercial database management systems.

In order to make this discussion more concrete, Figure 5 shows two operators in a query evaluation plan that prints se-

[4] Since each operator in such a query execution system will access a permanent relation, the name "access path selection" used for System R optimization, although including and actually focusing on join optimization, was entirely correct and more descriptive than "query optimization."

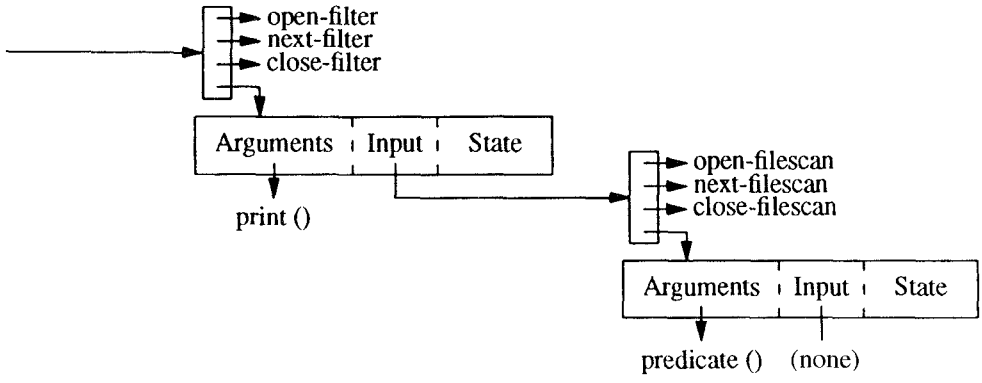


Figure 5. Two operators in a Volcano query plan.

lected records from a file. The purpose and capabilities of the *filter* operator in Volcano include printing items of a stream using a *print* function passed to the filter operator as one of its arguments. The small structure at the top gives access to the filter operator's iterator functions (the *open*, *next*, and *close* procedures) as well as to its state record. Using a pointer to this structure, the *open*, *next*, and *close* procedures of the filter operator can be invoked, and their local state can be passed to them as a procedure argument. The filter's iterator functions themselves, e.g., *open-filter*, can use the input pointer contained in the state record to invoke the input operator's functions, e.g., *open-file-scan*. Thus, the filter functions can invoke the file scan functions as needed and can pace the file scan according to the needs of the filter.

In this section, we have discussed general physical algebra issues and synchronization and data transfer between operators. Iterators are relatively straightforward to implement and are suitable building blocks for efficient, extensible query processing engines. In the following sections, we consider individual operators and algorithms including a comparison of sorting and hashing, detailed treatment of parallelism, special operators for emerging database applications such as scientific databases, and auxiliary techniques such as precomputation and compression.

2. SORTING AND HASHING

Before discussing specific algorithms, two general approaches to managing sets of data are introduced. The purpose of many query-processing algorithms is to perform some kind of matching, i.e., bringing items that are "alike" together and performing some operation on them. There are two basic approaches used for this purpose, sorting and hashing. This pair permeates many aspects of query processing, from indexing and clustering over aggregation and join algorithms to methods for parallelizing database operations. Therefore, we discuss these approaches first in general terms, without regard to specific algorithms. After a survey of specific algorithms for unary (aggregation, duplicate removal) and binary (join, semi-join, intersection, division, etc.) matching problems in the following sections, the duality of sort- and hash-based algorithms is discussed in detail.

2.1 Sorting

Sorting is used very frequently in database systems, both for presentation to the user in sorted reports or listings and for query processing in sort-based algorithms such as merge-join. Therefore, the performance effects of the many algorithmic tricks and variants of external sorting deserve detailed discussion in this survey. All sorting algorithms actually used in database systems use merg-

ing, i.e., the input data are written into initial sorted runs and then merged into larger and larger runs until only one run is left, the sorted output. Only in the unusual case that a data set is smaller than the available memory can in-memory techniques such as quicksort be used. An excellent reference for many of the issues discussed here is Knuth [1973], who analyzes algorithms much more accurately than we do in this introductory survey.

In order to ensure that the sort module interfaces well with the other operators, e.g., file scan or merge-join, sorting should be implemented as an iterator, i.e., with *open*, *next*, and *close* procedures as all other operators of the physical algebra. In the Volcano query-processing system (which is based on iterators), most of the sort work is done during *open-sort* [Graefe 1990a; 1993b]. This procedure consumes the entire input and leaves appropriate data structures for *next-sort* to produce the final sorted output. If the entire input fits into the sort space in main memory, *open-sort* leaves a sorted array of pointers to records in I/O buffer memory which is used by *next-sort* to produce the records in sorted order. If the input is larger than main memory, the *open-sort* procedure creates sorted runs and merges them until only one final merge phase is left. The last merge step is performed in the *next-sort* procedure, i.e., when demanded by the consumer of the sorted stream, e.g., a merge-join. The input to the sort module must be an iterator, and sort uses *open*, *next*, and *close* procedures to request its input; therefore, sort input can come from a scan or a complex query plan, and the sort operator can be inserted into a query plan at any place or at several places.

Table 2, which summarizes a taxonomy of parallel sort algorithms [Graefe 1990a], indicates some main characteristics of database sort algorithms. The first few items apply to any database sort and will be discussed in this section. The questions pertaining to parallel inputs and outputs and to data exchange will be considered in a later section on parallel algorithms, and the last question regarding substitute sorts will be touched on in the section on surrogate processing.

All sort algorithms try to exploit the duality between main-memory mergesort and quicksort. Both of these algorithms are recursive divide-and-conquer algorithms. The difference is that mergesort first divides physically and then merges on logical keys, whereas quicksort first divides on logical keys and then combines physically by trivially concatenating sorted subarrays. In general, one of the two phases—dividing and combining—is based on logical keys, whereas the other arranges data items only physically. We call these the logical and the physical phases. Sorting algorithms for very large data sets stored on disk or tape are also based on dividing and combining. Usually, there are two distinct subalgorithms, one for sorting within main memory and one for managing subsets of the data set on disk or tape. The choices for mapping logical and physical phases to dividing and combining steps are independent for these two subalgorithms. For practical reasons, e.g., ensuring that a run fits into main memory, the disk management algorithm typically uses physical dividing and logical combining (merging). A point of practical importance is the fan-in or degree of merging, but this is a parameter rather than a defining algorithm property.

There are two alternative methods for creating initial runs, also called "level-0 runs" here. First, an in-memory sort algorithm can be used, typically quicksort. Using this method, each run will have the size of allocated memory, and the number of initial runs W will be $W = \lceil R/M \rceil$ for input size R and memory size M. (Table 3 summarizes variables and their meaning in cost calculations in this survey.) Second, runs can be produced using replacement selection [Knuth 1973]. Replacement selection starts by filling memory with items which are organized into a priority heap, i.e., a data structure that efficiently supports the operations *insert* and *remove-smallest*. Next, the item with the smallest key is

Table 2. A Taxonomy of Database Sorting Algorithms

Determinant	Possible Options
Input division	Logical keys (partitioning) or physical division
Result combination	Logical keys (merging) or physical concatenation
Main-memory sort	Quicksort or replacement selection
Merging	Eager or lazy or semi-eager; lazy and semi-eager with or without optimizations
Read-ahead	No read-ahead or double-buffering or read-ahead with forecasting
Input	Single-stream or parallel
Output	Single-stream or parallel
Number of data exchanges	One or multiple
Data exchange	Before or after local sort
Sort objects	Original records or *key-RID* pairs (substitute sort)

removed from the priority heap and written to a run file and then immediately replaced in the priority heap with another item from the input. With high probability, this new item has a key larger than the item just written and therefore will be included in the same run file. Notice that if this is the case, the first run file will be larger than memory. Now the second item (the currently smallest item in the priority heap) is written to the run file and is also replaced immediately in memory by another item from the input. This process repeats, always keeping the memory and the priority heap entirely filled. If a new item has a key smaller than the last key written, the new item cannot be included in the current run file and is marked for the next run file. In comparisons among items in the heap, items marked for the current run file are always considered "smaller" than items marked for the next run file. Eventually, all items in memory are marked for the next run file, at which point the current run file is closed, and a new one is created.

Using replacement selection, run files are typically larger than memory. If the input is already sorted or almost sorted, there will be only one run file. This situation could arise, for example, if a file is sorted on field A but should be sorted on A as major and B as the minor sort key. If the input is sorted in reverse order, which is the worst case, each run file will be exactly as large as memory. If the input is random, the average run file will be twice the size of memory, except the first few runs (which get the process started) and the last run. On the average, the expected number of runs is about $W = \lceil R/(2 \times M) \rceil + 1$, i.e., about half as many runs as created with quicksort. A more detailed discussion and an analysis of replacement selection were provided by Knuth [1973].

An additional difference between quicksort and replacement selection is the resulting I/O pattern during run creation. Quicksort results in bursts of reads and writes for entire memory loads from the input file and to initial run files, while replacement selection alternates between individual read and write operations. If only a single device is used, quicksort may result in faster I/O because fewer disk arm movements are required. However, if different devices are used for input and temporary files, or if the input comes as a stream from another operator, the alternating behavior of replacement selection may permit more overlap of I/O and processing and therefore result in faster sorting.

The problem with replacement selection is memory management. If input items are kept in their original pages in the buffer (in order to save copying data, a real concern for large data volumes) each page must be kept in the buffer until its last record has been written to a run file. On the average, half a page's records will be in the priority heap. Thus, the priority heap must be reduced to half the size (the number of items in the heap

Table 3. Variables, Their Meaning, and Units

Variables	Description	Units
M	Memory size	pages
R, S	Inputs or their sizes	pages
C	Cluster or unit of I/O	pages
F, K	Fan-in or fan-out	(none)
W	Number of level-0 run files	(none)
L	Number of merge levels	(none)

is one half the number of records that fit into memory), canceling the advantage of longer and fewer run files. The solution to this problem is to copy records into a holding space and to keep them there while they are in the priority heap and until they are written to a run file. If the input items are of varying sizes, memory management is more complex than for quicksort because a new item may not fit into the space vacated in the holding space by the last item written into a run file. Solutions to this problem will introduce memory management overhead and some amount of fragmentation, i.e., the size of runs will be less than twice the size of memory. Thus, the advantage of having fewer runs must be balanced with the different I/O pattern and the disadvantage of more complex memory management.

The level-0 runs are *merged* into level-1 runs, which are merged into level-2 runs, etc., to produce the sorted output. During merging, a certain amount of buffer memory must be dedicated to each input run and the merge output. We call the unit of I/O a *cluster* in this survey, which is a number of pages located contiguously on disk. We indicate the cluster size with C, which we measure in pages just like memory and input sizes. The number of I/O clusters that fit in memory is the quotient of memory size and cluster size. The maximal merge fan-in F, i.e., the number of runs that can be merged at one time, is this quotient minus one cluster for the output. Thus, $F = \lfloor M/C - 1 \rfloor$. Since the sizes of runs grow by a factor F from level to level, the number of merge levels L, i.e., the number of times each item is written to a run file, is logarithmic with the input size, namely $L = \lceil \log_F(W) \rceil$.

There are four considerations that can improve the merge efficiency. The first two issues pertain to scheduling of I/O operations. First, scans are faster if read-ahead and write-behind are used; therefore, double-buffering using two pages of memory per input run and two for the merge output might speed the merge process [Salzberg 1990; Salzberg et al. 1990]. The obvious disadvantage is that the fan-in is cut in half. However, instead of reserving $2 \times F + 2$ clusters, a predictive method called *forecasting* can be employed in which the largest key in each input buffer is used to determine from which input run the next cluster will be read. Thus, the fan-in can be set to any number in the range $\lfloor M/(2 \times C) - 2 \rfloor \leq F \leq \lfloor M/C - 1 \rfloor$. One or two read-ahead buffers per input disk are sufficient, and $F = \lfloor M/C \rfloor - 3$ will be reasonable in most cases because it uses maximal fan-in with one forecasting input buffer and double-buffering for the merge output.

Second, if the operating system and the I/O hardware support them, using large cluster sizes for the run files is very beneficial. Larger cluster sizes will reduce the fan-in and therefore may increase the number of merge levels.[5] However, each merging level is performed much faster because fewer I/O operations and disk seeks and latency delays are required. Furthermore, if the unit of I/O is equal to a disk track, rotational latencies can be avoided entirely with a sufficiently smart disk con-

[5] In files storing permanent data, large clusters (units of I/O) containing many records may also create artificial buffer contention (if much more disk space is copied into the buffer than truly necessary for one record) and "false sharing" in environments with page (cluster) locks, i.e., artificial concurrency conflicts. Since run files in a sort operation are not shared but temporary, these problems do not exist in this context.

troller. Usually, relatively small fan-ins with large cluster sizes are the optimal choice, even if the sort requires multiple merge levels [Graefe 1990a]. The precise tradeoff depends on disk seek, latency, and transfer times. It is interesting to note that the optimal cluster size and fan-in basically do not depend on the input size.

As a concrete example, consider sorting a file of $R = 50$ MB $= 51{,}200$ KB using $M = 160$ KB of memory. The number of runs created by quicksort will be $W = \lceil 51200/160 \rceil = 320$. Depending on the disk access and transfer times (e.g., 25 ms disk seek and latency, 2 ms transfer time for a page of 4 KB), $C = 16$ KB will typically be a good cluster size for fast merging. If one cluster is used for read-ahead and two for the merge output, the fan-in will be $F = \lfloor 160/16 \rfloor - 3 = 7$. The number of merge levels will be $L = \lceil \log_7(320) \rceil = 3$. If a 16 KB I/O operation takes $T = 33$ ms, the total I/O time, including a factor of two for writing and reading at each merge level, for the entire sort will be $2 \times L \times \lceil R/C \rceil \times T = 10.56$ min.

An entirely different approach to determining optimal cluster sizes and the amount of memory allocated to forecasting and read-ahead is based on processing and I/O bandwidths and latencies. The cluster sizes should be set such that the I/O bandwidth matches the processing bandwidth of the CPU. Bandwidths for both I/O and CPU are measured here in record or bytes per unit time; instructions per unit time (MIPS) are irrelevant. It is interesting to note that the CPU's processing bandwidth is largely determined by how fast the CPU can assemble new pages, in other words, how fast the CPU can copy records within memory. This performance measure is usually ignored in modern CPU and cache designs geared towards high MIPS or MFLOPS numbers [Ousterhout 1990].

Tuning the sort based on bandwidth and latency proceeds in three steps. First, the cluster size is set such that the processing and I/O bandwidths are equal or very close to equal. If the sort is I/O bound, the cluster size is increased for less disk access overhead per record and therefore faster I/O; if the sort is CPU bound, the cluster size is decreased to slow the I/O in favor of a larger merge fan-in. Next, in order to ensure that the two processing components (I/O and CPU) never (or almost never) have to wait for one another, the amount of space dedicated to read-ahead is determined as the I/O time for one cluster multiplied by the processing bandwidth. Typically, this will result in one cluster of read-ahead space per disk used to store and read inputs run into a merge. Of course, in order to make read-ahead effective, forecasting must be used. Finally, the same amount of buffer space is allocated for the merge output (access latency times bandwidth) to ensure that merge processing never has to wait for the completion of output I/O. It is an open issue whether these two alternative approaches to tuning cluster size and read-ahead space result in different allocations and sorting speeds or whether one of them is more effective than the other.

The third and fourth merging issues focus on using (and exploiting) the maximal fan-in as effectively and often as possible. Both issues require adjusting the fan-in of the first merge step using the formula given below, either the first merge step of all merge steps or, in semi-eager merging [Graefe 1990a], the first merge step after the end of the input has been reached. This adjustment is used for only one merge step, called the *initial merge* here, not for an entire merge level.

The third issue to be considered is that the number of runs W is typically not a power of F; therefore, some merges proceed with fewer than F inputs, which creates the opportunity for some optimization. Instead of always merging runs of only one level together, the optimal strategy is to merge as many runs as possible using the smallest run files available. The only exception is the fan-in of the first merge, which is determined to ensure that all subsequent merges will use the full fan-in F.

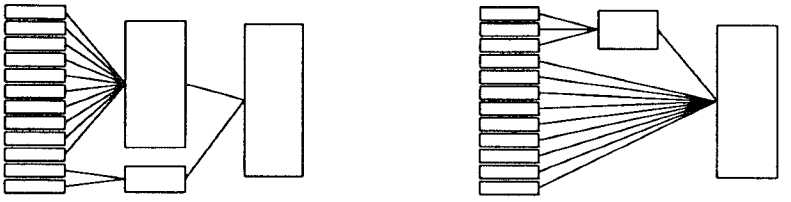


Figure 6. Naive and optimized merging.

Let us explain this idea with the example shown in Figure 6. Consider a sort with a maximal fan-in $F = 10$ and an input file that requires $W = 12$ initial runs. Instead of merging only runs of the same level as shown in Figure 6, merging is delayed until the end of the input has been reached. In the first merge step, only 3 of the 12 runs are combined, and the result is then merged with the other 9 runs, as shown in Figure 6. The I/O cost (measured by the number of memory loads that must be written to any of the runs created) for the first strategy is $12 + 10 + 2 = 24$, while for the second strategy it is $12 + 3 = 15$. In other words, the first strategy requires 60% more I/O to temporary files than the second one. The general rule is to merge just the right number of runs after the end of the input file has been reached, and to always merge the smallest runs available for merging. More detailed examples are given in Graefe [1990a]. One consequence of this optimization is that the merge depth L, i.e., the number of run files a record is written to during the sort or the number of times a record is written to and read from disk, is not uniform for all records. Therefore, it makes sense to calculate an average merge depth (as required in cost estimation during query optimization), which may be a fraction. Of course, there are much more sophisticated merge optimizations, e.g., cascade and polyphase merges [Knuth 1973].

Fourth, since some operations require multiple sorted inputs, for example merge-join (to be discussed in the section on matching) and sort output can be passed directly from the final merge into the next operation (as is natural when using iterators), memory must be divided among multiple final merges. Thus, the final fan-in f and the "normal" fan-in F should be specified separately in an actual sort implementation. Using a final fan-in of 1 also allows the sort operator to produce output into a very slow operator, e.g., a display operation that allows scrolling by a human user, without occupying a lot of buffer memory for merging input runs over an extended period of time.[6]

Considering the last two optimization options for merging, the following formula determines the fan-in of the first merge. Each merge with normal fan-in F will reduce the number of run files by $F - 1$ (removing F runs, creating one new one). The goal is to reduce the number of runs from W to f and then to 1 (the final output). Thus, the first merge should reduce the number of runs to $f + k(F - 1)$ for some integer k. In other words, the first merge should use a fan-in of $F_0 = ((W - f - 1) \bmod (F - 1)) + 2$. In the example of Figure 6, $(12 - 10 - 1) \bmod (10 - 1) + 2$ results in a fan-in for the initial merge of $F_0 = 3$. If the sort of Figure 6 were the input into a merge-join and if a final fan-in of 5 were desired, the initial merge should proceed with a fan-in of $F_0 = (12 - 5 - 1) \bmod (10 - 1) + 2 = 8$.

If multiple sort operations produce input data for a common consumer operator, e.g., a merge-join, the two final fan-ins should be set proportionally to the

[6] There is a similar case of resource sharing among the operators producing a sort's input and the run generation phase of the sort. We will come back to these issues later in the section on executing and scheduling complex queries and plans.

size of the two inputs. For example, if two merge-join inputs are 1 MB and 9 MB, and if 20 clusters are available for inputs into the two final merges, then 2 clusters should be allocated for the first and 18 clusters for the second input (1/9 = 2/18).

Sorting is sometimes criticized because it requires, unlike hybrid hashing (discussed in the next section), that the entire input be written to run files and then retrieved for merging. This difference has a particularly large effect for files only slightly larger than memory, e.g., 1.25 times the size of memory. Hybrid hashing determines dynamically how much of the input data truly must be written to temporary disk files. In the example, only slightly more than one quarter of the memory size must be written to temporary files on disk while the remainder of the file remains in memory. In sorting, the entire file (1.25 memory sizes) is written to one or two run files and then read for merging. Thus, sorting seems to require five times more I/O for temporary files in this example than hybrid hashing. However, this is not necessarily true. The simple trick is to write initial runs in decreasing (reverse) order. When the input is exhausted, and merging in increasing order commences, buffer memory is still full of useful pages with small sort keys that can be merged immediately without I/O and that never have to be written to disk. The effect of writing runs in reverse order is comparable to that of hybrid hashing, i.e., it is particularly effective if the input is only slightly larger than the available memory.

To demonstrate the effect of cluster size optimizations (the second of the four merging issues discussed above), we sorted 100,000 100-byte records, about 10 MB, with the Volcano query processing system, which includes all merge optimizations described above with the exception of read-ahead and forecasting. (This experiment and a similar one were described in more detail earlier [Graefe 1990a; Graefe et al. 1993].) We used a sort space of 40 pages (160 KB) within a 50-page (200 KB) I/O buffer, varying the cluster size from 1 page (4 KB) to 15 pages (60 KB). The initial run size was 1,600 records, for a total of 63 initial runs. We counted the number of I/O operations and the transferred pages for all run files, and calculated the total I/O cost by charging 25 ms per I/O operation (for seek and rotational latency) and 2 ms for each transferred page (assuming 2 MB/sec transfer rate). As can be seen in Table 4 and Figure 7, there is an optimal cluster size with minimal I/O cost. The curve is not as smooth as might have been expected from the approximate cost function because the curve reflects all real-system effects such as rounding (truncating) the fan-in if the cluster size is not an exact divisor of the memory size, the effectiveness of merge optimizations varying for different fan-ins, and internal fragmentation in clusters. The detailed data in Table 4, however, reflect the trends that larger clusters and smaller fan-ins clearly increase the amount of data transferred but not the number of I/O operations (disk and latency time) until the fan-in has shrunk to very small values, e.g., 3. It is clearly suboptimal to always choose the smallest cluster size (1 page) in order to obtain the largest fan-in and fewest merge levels. Furthermore, it seems that the range of cluster sizes that result in near-optimal total I/O costs is fairly large; thus it is not as important to determine the exact value as it is to use a cluster size "in the right ball park." The optimal fan-in is typically fairly small; however, it is not e or 3 as derived by Bratbergsengen [1984] under the (unrealistic) assumption that the cost of an I/O operation is independent of the amount of data being transferred.

2.2 Hashing

For many matching tasks, hashing is an alternative to sorting. In general, when equality matching is required, hashing should be considered because the expected complexity of set algorithms based on hashing is $O(N)$ rather than

Table 4. Effect of Cluster Size Optimizations

Cluster Size [×4 KB]	Fan-in	Average Depth	Disk Operations	Pages Transferred [×4 KB]	Total I/O Cost [sec]
1	40	1.376	6874	6874	185.598
2	20	1.728	4298	8596	124.642
3	13	1.872	3176	9528	98.456
4	10	1.936	2406	9624	79.398
5	8	2.000	1984	9920	69.440
6	6	2.520	2132	12792	78.884
7	5	2.760	1980	13860	77.220
8	5	2.760	1718	13744	70.438
9	4	3.000	1732	15588	74.476
10	4	3.000	1490	14900	67.050
11	3	3.856	1798	19778	84.506
12	3	3.856	1686	20232	82.614
13	3	3.856	1628	21164	83.028
14	2	5.984	2182	30548	115.646
15	2	5.984	2070	31050	113.850

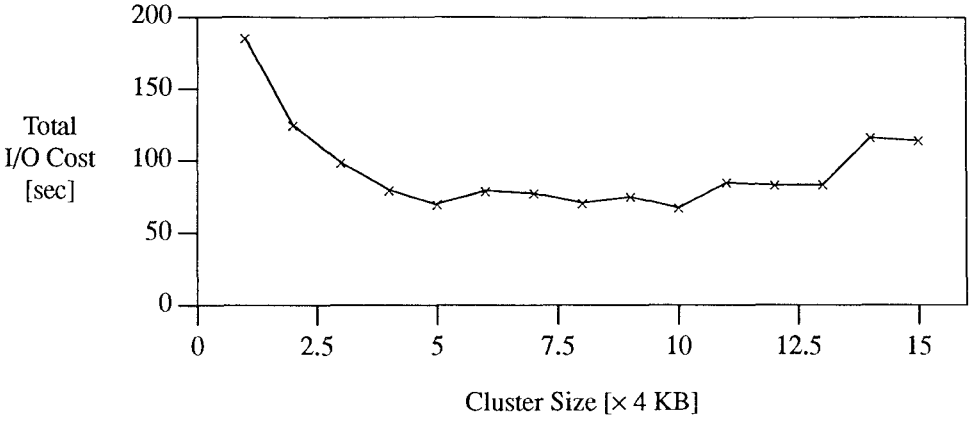


Figure 7. Effect of cluster size optimizations.

$O(N \log N)$ as for sorting. Of course, this makes intuitive sense if hashing is viewed as radix sorting on a virtual key [Knuth 1973].

Hash-based query processing algorithms use an in-memory hash table of database objects to perform their matching task. If the entire hash table (including all records or items) fits into memory, hash-based query processing algorithms are very easy to design, understand, and implement, and they outperform sort-based alternatives. Note that for binary matching operations, such as join or intersection, only one of the two inputs must fit into memory. However, if the required hash table is larger than memory, *hash table overflow* occurs and must be dealt with.

There are basically two methods for managing hash table overflow, namely *avoidance* and *resolution*. In either case, the input is divided into multiple partition files such that partitions can be processed independently from one another, and the concatenation of the results of all partitions is the result of the entire operation. Partitioning should ensure that the partitioning files are of roughly even size and can be done using either hash parti-

tioning or range partitioning, i.e., based on keys estimated to be quantiles. Usually, partition files can be processed using the original hash-based algorithm. The maximal partitioning *fan-out* F, i.e., number of partition files created, is determined by the memory size M divided over the cluster size C minus one cluster for the partitioning input, i.e., $F = \lfloor M/C - 1 \rfloor$, just like the fan-in for sorting.

In hash table overflow avoidance, the input set is partitioned into F partition files before any in-memory hash table is built. If it turns out that fewer partitions than have been created would have been sufficient to obtain partition files that will fit into memory, bucket tuning (collapsing multiple small buckets into larger ones) and dynamic destaging (determining which buckets should stay in memory) can improve the performance of hash-based operations [Kitsuregawa et al. 1989a; Nakayama et al. 1988].

Algorithms based on hash table overflow resolution start with the assumption that overflow will not occur, but resort to basically the same set of mechanisms as hash table overflow avoidance once it does occur. No real system uses this naive hash table overflow resolution because so-called hybrid hashing is as efficient but more flexible. Hybrid hashing combines in-memory hashing and overflow resolution [DeWitt et al. 1984; Shapiro 1986]. Although invented for relational join and known as hybrid hash join, hybrid hashing is equally applicable to all hash-based query processing algorithms.

Hybrid hash algorithms start out with the (optimistic) premise that no overflow will occur; if it does, however, they partition the input into multiple partitions of which only one is written immediately to temporary files on disk. The other $F - 1$ partitions remain in memory. If another overflow occurs, another partition is written to disk. If necessary, all F partitions are written to disk. Thus, hybrid hash algorithms use all available memory for in-memory processing, but at the same time are able to process large input files by overflow resolution. Figure 8 shows the idea of hybrid hash algorithms. As many hash buckets as possible are kept in memory, e.g., as linked lists as indicated by solid arrows. The other hash buckets are spooled to temporary disk files, called the overflow or partition files, and are processed in later stages of the algorithm. Hybrid hashing is useful if the input size R is larger than the memory size M but smaller than the memory size multiplied by the fan-out F, i.e., $M < R \leq F \times M$.

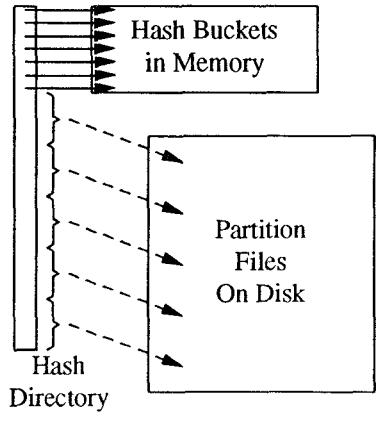


Figure 8. Hybrid hashing.

In order to predict the number of I/O operations (which actually is not necessary for execution because the algorithm adapts to its input size but may be desirable for cost estimation during query optimization), the number of required partition files on disk must be determined. Call this number K, which must satisfy $0 \leq K \leq F$. Presuming that the assignment of buckets to partitions is optimal and that each partition file is equal to the memory size M, the amount of data that may be written to K partition files is equal to $K \times M$. The number of required I/O buffers is 1 for the input and K for the output partitions, leaving $M - (K + 1) \times C$ memory for the hash table. The optimal K for a given input size R is the minimal K for which $K \times M + (M - (K + 1) \times C) \geq R$. Solving this inequality and taking the smallest such K results in $K = \lceil (R - M + C)/(M - C) \rceil$. The minimal possible I/O cost, including a factor of 2 for writing

and reading the partition files and measured in the amount of data that must be written or read, is $2 \times (R - (M - (K + 1) \times C))$. To determine the I/O time, this amount must be divided by the cluster size and multiplied with the I/O time for one cluster.

For example, consider an input of $R = 240$ pages, a memory of $M = 80$ pages, and a cluster size of $C = 8$ pages. The maximal fan-out is $F = \lfloor 80/8 - 1 \rfloor = 9$. The number of partition files that need to be created on disk is $K = \lceil (240 - 80 + 8)/(80 - 8) \rceil = 3$. In other words, in the best case, $K \times C = 3 \times 8 = 24$ pages will be used as output buffers to write $K = 3$ partition files of no more than $M = 80$ pages, and $M - (K + 1) \times C = 80 - 4 \times 8 = 48$ pages of memory will be used as the hash table. The total amount of data written to and read from disk is $2 \times (240 - (80 - 4 \times 8)) = 384$ pages. If writing or reading a cluster of $C = 8$ pages takes 40 msec, the total I/O time is $384/8 \times 40 = 1.92$ sec.

In the calculation of K, we assumed an optimal assignment of hash buckets to partition files. If buckets were assigned in the most straightforward way, e.g., by dividing the hash directory into F equal-size regions and assigning the buckets of one region to a partition as indicated in Figure 8, all partitions were of nearly the same size, and either all or none of them will fit into their output cluster and therefore into memory. In other words, once hash table overflow occurred, all input was written to partition files. Thus, we presumed in the earlier calculations that hash buckets were assigned more intelligently to output partitions.

There are three ways to assign hash buckets to partitions. First, each time a hash table overflow occurs, a fixed number of hash buckets is assigned to a new output partition. In the Gamma database machine, the number of disk partitions is chosen "such that each bucket [bucket here means what is called an output partition in this survey] can reasonably be expected to fit in memory" [DeWitt and Gerber 1985], e.g., 10% of the hash buckets in the hash directory for a fan-out of 10 [Schneider 1990]. In other words, the fan-out is set a priori by the query optimizer based on the expected (estimated) input size. Since the page size in Gamma is relatively small, only a fraction of memory is needed for output buffers, and an in-memory hash table can be used even while output partitions are being written to disk. Second, in bucket tuning and dynamic destaging [Kitsuregawa et al. 1989a; Nakayama 1988], a large number of small partition files is created and then collapsed into fewer partition files no larger than memory. In order to obtain a large number of partition files and, at the same time, retain some memory for a hash table, the cluster size is set quite small, e.g., $C = 1$ page, and the fan-out is very large though not maximal, e.g., $F = M/C/2$. In the example above, $F = 40$ output partitions with an average size of $R/F = 6$ pages could be created, even though only $K = 3$ output partitions are required. The smallest partitions are assigned to fill an in-memory hash table of size $M - K \times C = 80 - 3 \times 1 = 77$ pages. Hopefully, the dynamic destaging rule—when an overflow occurs, assign the largest partition still in memory to disk—ensures that indeed the smallest partitions are retained in memory. The partitions assigned to disk are collapsed into $K = 3$ partitions of no more than $M = 80$ pages, to be processed in $K = 3$ subsequent phases. In binary operations such as intersection and relational join, bucket tuning is quite effective for *skew* in the first input, i.e., if the hash value distribution is nonuniform and if the partition files are of uneven sizes. It avoids spooling parts of the second (typically larger) input to temporary partition files because the partitions in memory can be matched immediately using a hash table in the memory not required as output buffer and because a number of small partitions have been collapsed into fewer, larger partitions, increasing the memory available for the hash table. For skew in the second input, bucket tuning and dynamic destaging have no advantage. Another disadvan-

tage of bucket tuning and dynamic destaging is that the cluster size has to be relatively small, thus requiring a large number of I/O operations with disk seeks and rotational latencies to write data to the overflow files. Third, statistics gathered before hybrid hashing commences can be used to assign hash buckets to partitions [Graefe 1993a].

Unfortunately, it is possible that one or more partition files are larger than memory. In that case, partitioning is used recursively until the file sizes have shrunk to memory size. Figure 9 shows how a hash-based algorithm for a unary operation, such as aggregation or duplicate removal, partitions its input file over multiple recursion levels. The recursion terminates when the files fit into memory. In the deepest recursion level, hybrid hashing may be employed.

If the partitioning (hash) function is good and creates a uniform hash value distribution, the file size in each recursion level shrinks by a factor equal to the fan-out, and therefore the number of recursion levels L is logarithmic with the size of the input being partitioned. After L partitioning levels, each partition file is of size $R' = R/F^L$. In order to obtain partition files suitable for hybrid hashing (with $M < R' \leq F \times M$), the number of full recursion levels L, i.e., levels at which hybrid hashing is not applied, is $L = \lceil \log_F(R/M) \rceil$. The I/O cost of the remaining step using hybrid hashing can be estimated using the hybrid hash formula above with R replaced by R' and multiplying the cost with F^L because hybrid hashing is used for this number of partition files. Thus, the total I/O cost for partitioning an input and using hybrid hashing in the deepest recursion level is

$$
\begin{aligned}
&2 \times R \times L + 2 \times F^L \\
&\quad \times (R' - (M - K \times C)) \\
&\quad = 2 \times (R \times (L + 1) - F^L \\
&\quad \times (M - K \times C)) \\
&\quad = 2 \times (R \times (L + 1) - F^L \times (M \\
&\quad - \lceil (R' - M)/(M - C) \rceil \times C)).
\end{aligned}
$$

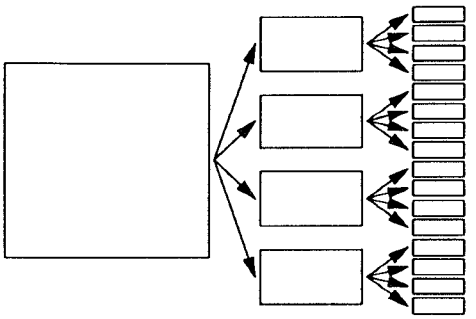


Figure 9. Recursive partitioning.

A major problem with hash-based algorithms is that their performance depends on the quality of the hash function. In many situations, fairly simple hash functions will perform reasonably well. Remember that the purpose of using hash-based algorithms usually is to find database items with a specific key or to bring like items together; thus, methods as simple as using the value of a join key as a hash value will frequently perform satisfactorily. For string values, good hash values can be determined by using binary "exclusive or" operations or by determining cyclic redundancy check (CRC) values as used for reliable data storage and transmission. If the quality of the hash function is a potential problem, universal hash functions should be considered [Carter and Wegman 1979].

If the partitioning is skewed, the recursion depth may be unexpectedly high, making the algorithm rather slow. This is analogous to the worst-case performance of quicksort, $O(N^2)$ comparisons for an array of N items, if the partitioning pivots are chosen extremely poorly and do not divide arrays into nearly equal subarrays.

Skew is the major danger for inferior performance of hash-based query-processing algorithms. There are several ways to deal with skew. For hash-based algorithms using overflow avoidance, bucket tuning and dynamic destaging are quite effective. Another method is to obtain statistical information about hash values and to use it to carefully assign

hash buckets to partitions. Such statistical information can be kept in the form of histograms and can either come from permanent system catalogs (metadata), from sampling the input, or from previous recursion levels. For example, for an intermediate query processing result for which no statistical parameters are known a priori, the first partitioning level might have to proceed naively pretending that the partitioning hash function is perfect, but the second and further recursion levels should be able to use statistics gathered in earlier levels to ensure that each partitioning step creates even partitions, i.e., that the data is partitioned with maximal effectiveness [Graefe 1993a]. As a final resort, if skew cannot be managed otherwise, or if distribution skew is not the problem but duplicates are, some systems resort to algorithms that are not affected by data or hash value skew. For example, Tandem's hash join algorithm resorts to nested-loops join (to be discussed later) [Zeller and Gray 1990].

As for sorting, larger cluster sizes result in faster I/O at the expense of smaller fan-outs, with the optimal fan-out being fairly small [Graefe 1993a; Graefe et al. 1993]. Thus, multiple recursion levels are not uncommon for large files, and statistics gathered on one level to limit skew effects on the next level are a realistic method for large files to control the performance penalties of uneven partitioning.

3. DISK ACCESS

All query evaluation systems have to access base data stored in the database. For databases in the megabyte to terabyte range, base data are typically stored on secondary storage in the form of rotating random-access disks. However, deeper storage hierarchies including optical storage, (maybe robot-operated) tape archives, and remote storage servers will also have to be considered in future high-functionality high-volume database management systems, e.g., as outlined by Stonebraker [1991]. Research into database systems supporting and exploiting a deep storage hierarchy is still in its infancy.

On the other hand, some researchers have considered in-memory or main-memory databases, motivated both by the desire for faster transaction and query-processing performance and by the decreasing cost of semi-conductor memory [Analyti and Pramanik 1992; Bitton et al. 1987; Bucheral et al. 1990; DeWitt et al. 1984; Gruenwald and Eich 1991; Kumar and Burger 1991; Lehman and Carey 1986; Li and Naughton 1988; Severance et al. 1990; Whang and Krishnamurthy 1990]. However, for most applications, an analysis by Gray and Putzolo [1987] demonstrated that main memory is cost effective only for the most frequently accessed data. The time interval between accesses with equal disk and memory costs was five minutes for their values of memory and disk prices and was expected to grow as main-memory prices decrease faster than disk prices. For the purposes of this survey, we will presume a disk-based storage architecture and will consider disk I/O one of the major costs of query evaluation over large databases.

3.1 File Scans

The first operator to access base data is the file scan, typically combined with a built-in selection facility. There is not much to be said about file scan except that it can be made very fast using read-ahead, particularly, large-chunk ("track-at-a-crack") read-ahead. In some database systems, e.g., IBM's DB2, the read-ahead size is coordinated with the free space left for future insertions during database reorganization. If a free page is left after every 15 full data pages, the read-ahead unit of 16 pages (64 KB) ensures that overflow records are immediately available in the buffer.

Efficient read-ahead requires contiguous file allocation, which is supported by many operating systems. Such contiguous disk regions are frequently called extents. The UNIX operating system does not provide contiguous files, and many

database systems running on UNIX use "raw" devices instead, even though this means that the database management system must provide operating-system functionality such as file structures, disk space allocation, and buffering.

The disadvantages of large units of I/O are buffer fragmentation and the waste of I/O and bus bandwidth if only individual records are required. Permitting different page sizes may seem to be a good idea, even at the added complexity in the buffer manager [Carey et al. 1986; Sikeler 1988], but this does not solve the problem of mixed sequential scans and random record accesses within one file. The common solution is to choose a middle-of-the-road page size, e.g., 8 KB, and to support multipage read-ahead.

3.2 Associative Access Using Indices

In order to reduce the number of accesses to secondary storage (which is relatively slow compared to main memory), most database systems employ associative search techniques in the form of indices that map key or attribute values to locator information with which database objects can be retrieved. The best known and most often used database index structure is the B-tree [Bayer and McCreighton 1972; Comer 1979]. A large number of extensions to the basic structure and its algorithms have been proposed, e.g., B^+-trees for faster scans, fast loading from a sorted file, increased fanout and reduced depth by prefix and suffix truncation, B^*-trees for better space utilization in random insertions, and top-down B-trees for better locking behavior through preventive maintenance [Guibas and Sedgewick 1978]. Interestingly, B-trees seem to be having a renaissance as a research subject, in particular with respect to improved space utilization [Baeza-Yates and Larson 1989], concurrency control [Srinivasan and Carey 1991], recovery [Lanka and Mays 1991], parallelism [Seeger and Larson 1991], and on-line creation of B-trees for very large databases [Srinivasan and Carey 1992]. On-line reorganization and modification of storage structures, though not a new idea [Omiecinski 1985], is likely to become an important research topic within database research over the next few years as databases become larger and larger and are spread over many disks and many nodes in parallel and distributed systems.

While most current database system implementations only use some form of B-trees, an amazing variety of index structures has been described in the literature [Becker et al. 1991; Beckmann et al. 1990; Bentley 1975; Finkel and Bentley 1974; Guenther and Bilmes 1991; Gunther and Wong 1987; Gunther 1989; Guttman 1984; Henrich et al. 1989; Hoel and Samet 1992; Hutflesz et al. 1988a; 1988b; 1990; Jagadish 1991; Kemper and Wallrath 1987; Kolovson and Stonebraker 1991; Kriegel and Seeger 1987; 1988; Lomet and Salzberg 1990a; Lomet 1992; Neugebauer 1991; Robinson 1981; Samet 1984; Six and Widmayer 1988]. One of the few multidimensional index structures actually implemented in a complete database management system are R-trees in Postgres [Guttman 1984; Stonebraker et al. 1990b].

Table 5 shows some example index structures classified according to four characteristics, namely their support for ordering and sorted scans, their dynamic-versus-static behavior upon insertions and deletions, their support for multiple dimensions, and their support for point data versus range data. We omitted hierarchical concatenation of attributes and uniqueness, because all index structures can be implemented to support these. The indication "no range data" for multidimensional index structures indicates that range data are not part of the basic structure, although they can be simulated using twice the number of dimensions. We included a reference or two with each structure; we selected original descriptions and surveys over the many subsequent papers on special aspects such as performance analyses, multidisk and multiprocessor implementations, page placement on disk, concurrency control, recovery, order-preserving

Table 5. Classification of Some Index Structures

Structure	Ordered	Dynamic	Multi-Dim.	Range Data	References
ISAM	Yes	No	No	No	[Larson 1981]
B-trees	Yes	Yes	No	No	[Bayer and McCreighton 1972; Comer 1979]
Quad-tree	Yes	Yes	Yes	No	[Finkel and Bentley 1974; Samet 1984]
kD-trees	Yes	Yes	Yes	No	[Bentley 1975]
KDB-trees	Yes	Yes	Yes	No	[Robinson 1981]
hB-trees	Yes	Yes	Yes	No	[Lomet and Salzberg 1990a]
R-trees	Yes	Yes	Yes	Yes	[Guttman 1984]
Extendible Hashing	No	Yes	No	No	[Fagin et al. 1979]
Linear Hashing	No	Yes	No	No	[Litwin 1980]
Grid Files	Yes	Yes	Yes	No	[Nievergelt et al. 1984]

hashing, mapping range data of N dimensions into point data of $2N$ dimensions, etc.—this list suggests the wealth of subsequent research, in particular on B-trees, linear hashing, and refined multidimensional index structures.

Storage structures typically thought of as index structures may be used as primary structures to store actual data or as redundant structures ("access paths") that do not contain actual data but pointers to the actual data items in a separate data file. For example, Tandem's NonStop SQL system uses B-trees for actual data as well as for redundant index structures. In this case, a redundant index structure contains not absolute locations of the data items but keys used to search the primary B-tree. If indices are redundant structures, they can still be used to cluster the actual data items, i.e., the order or organization of index entries determines the order of items in the data file. Such indices are called *clustering* indices; other indices are called *nonclustering* indices. Clustering indices do not necessarily contain an entry for each data item in the primary file, but only one entry for each page of the primary file; in this case, the index is called *sparse*. Nonclustering indices must always be *dense*, i.e., there are the same number of entries in the index as there are items in the primary file.

The common theme for all index structures is that they associatively map some attribute of a data object to some locator information that can then be used to retrieve the actual data object. Typically, in relational systems, an attribute value is mapped to a tuple or record identifier (TID or RID). Different systems use different approaches, but it seems that most new designs do not firmly attach the record lookup to the index scan.

There are several advantages to separating index scan and record lookup. First, it is possible to scan an index only without ever retrieving records from the underlying data file. For example, if only salary values are needed (e.g., to determine the count or sum of all salaries), it is sufficient to access the salary index only without actually retrieving the data records. The advantages are that (i) fewer I/Os are required (consider the number of I/Os for retrieving N successive index entries and those to retrieve N index entries plus N full records, in particular if the index is nonclustering [Mackert and Lohman 1989] and (ii) the remaining I/O operations are basically sequential along the leaves of the index (at least for B^+-trees; other index types behave differently). The optimizers of several commercial relational products have recently been revised to recognize situations in which an index-only scan is sufficient. Second, even if none of the existing indices is sufficient by itself, multiple indices may be "joined" on equal RIDs to obtain all attributes required for a query (join algorithms are discussed below in the section on binary matching). For ex-

ample, by matching entries in indices on salaries and on names by equal RIDs, the correct salary-name pairs are established. If a query requires only names and salaries, this "join" has made accessing the underlying data file obsolete. Third, if two or more indices apply to individual clauses of a query, it may be more effective to take the union or intersection of RID lists obtained from two index scans than using only one index (algorithms for union and intersection are also discussed in the section on binary matching). Fourth, joining two tables can be accomplished by joining the indices on the two join attributes followed by record retrievals in the two underlying data sets; the advantage of this method is that only those records will be retrieved that truly contribute to the join result [Kooi 1980]. Fifth, for nonclustering indices, sets of RIDs can be sorted by physical location, and the records can be retrieved very efficiently, reducing substantially the number of disk seeks and their seek distances. Obviously, several of these techniques can be combined. In addition, some systems such as Rdb/VMS and DB2 use very sophisticated implementations of multiindex scans that decide dynamically, i.e., during run-time, which indices to scan, whether scanning a particular index reduces the resulting RID list sufficiently to offset the cost of the index scan, and whether to use bit vector filtering for the RID list intersection (see a later section on bit vector filtering) [Antoshenkov 1993; Mohan et al. 1990].

Record access performance for nonclustering indices can be addressed without performing the entire index scan first (as required if all RIDs are to be sorted) by using a "window" of RIDs. Instead of obtaining one RID from the index scan, retrieving the record, getting the next RID from the index scan, etc., the lookup operator (sometimes called "functional join") could load N RIDs, sort them into a priority heap, retrieve the most conveniently located record, get another RID, insert it into the heap, retrieve a record, etc. Thus, a functional join operator using a window always has N open references to items that must be retrieved, giving the functional join operator significant freedom to fetch items from disk efficiently. Of course, this technique works most effectively if no other transactions or operators use the same disk drive at the same time.

This idea has been generalized to assemble complex objects. In object-oriented systems, objects can contain pointers to (identifiers of) other objects or components, which in turn may contain further pointers, etc. If multiple objects and all their unresolved references can be considered concurrently when scheduling disk accesses, significant savings in disk seek times can be achieved [Keller et al. 1991].

3.3 Buffer Management

I/O cost can be further reduced by caching data in an I/O buffer. A large number of buffer management techniques have been devised; we give only a few references. Effelsberg and Haerder [1984] survey many of the buffer management issues, including those pertaining to issues of recovery, e.g., write-ahead logging. In a survey paper on the interactions of operating systems and database management systems, Stonebraker [1981] pointed out that the "standard" buffer replacement policy, LRU (least recently used), is wrong for many database situations. For example, a file scan reads a large set of pages but uses them only once, "sweeping" the buffer clean of all other pages, even if they might be useful in the future and should be kept in memory. Sacco and Schkolnick [1982; 1986] focused on the nonlinear performance effects of buffer allocation to many relational algorithms, e.g., nested-loops join. Chou [1985] and Chou and DeWitt [1985] combined these two ideas in their DBMIN algorithm which allocates a fixed number of buffer pages to each scan, depending on its needs, and uses a local replacement policy for each scan appropriate to its reference pattern. A recent study into buffer allocation is by Faloutsos et al. [1991] and Ng et al. [1991] on using

marginal gain for buffer allocation. A very promising research direction for buffer management in object-oriented database systems is the work by Palmer and Zdonik [1991] on saving reference patterns and using them to predict future object faults and to prevent them by prefetching the required pages.

The interactions of index retrieval and buffer management were studied by Sacco [1987] as well as Mackert and Lohman [1989], and several authors studied database buffer management and virtual memory provided by the operating system [Sherman and Brice 1976; Stonebraker 1981; Traiger 1982].

On the level of buffer manager implementation, most database buffer managers do not provide *read* and *write* interfaces to their client modules but fixing and unfixing, also called pinning and unpinning. The semantics of fixing is that a fixed page is not subject to replacement or relocation in the buffer pool, and a client module may therefore safely use a memory address within a fixed page. If the buffer manager needs to replace a page but all its buffer frames are fixed, some special action must occur such as dynamic growth of the buffer pool or transaction abort.

The iterator implementation of query evaluation algorithms can exploit the buffer's fix/unfix interface by passing pointers to items (records, objects) fixed in the buffer from iterator to iterator. The receiving iterator then owns the fixed item; it may unfix it immediately (e.g., after a predicate fails), hold on to the fixed record for a while (e.g., in a hash table), or pass it on to the next iterator (e.g., if a predicate succeeds). Because the iterator control and interaction of operators ensure that items are never produced and fixed before they are required, the iterator protocol is very efficient in its buffer usage.

Some implementors, however, have felt that intermediate results should not be materialized or kept in the database system's I/O buffer, e.g., in order to ease implementation of transaction (ACID) semantics, and have designed a separate memory management scheme for intermediate results and items passed from iterator to iterator. The cost of this decision is additional in-memory copying as well as the possible inefficiencies associated with, in effect, two buffer and memory managers.

4. AGGREGATION AND DUPLICATE REMOVAL

Aggregation is a very important statistical concept to summarize information about large amounts of data. The idea is to represent a set of items by a single value or to classify items into groups and determine one value per group. Most database systems support aggregate functions for minimum, maximum, sum, count, and average (arithmetic mean). Other aggregates, e.g., geometric mean or standard deviation, are typically not provided, but may be constructed in some systems with extensibility features. Aggregation has been added to both relational calculus and algebra and adds the same expressive power to each of them [Klug 1982].

Aggregation is typically supported in two forms, called *scalar aggregates* and *aggregate functions* [Epstein 1979]. Scalar aggregates calculate a single scalar value from a unary input relation, e.g., the sum of the salaries of all employees. Scalar aggregates can easily be determined using a single pass over a data set. Some systems exploit indices, in particular for minimum, maximum, and count.

Aggregate functions, on the other hand, determine a set of values from a binary input relation, e.g., the sum of salaries for each department. Aggregate functions are relational operators, i.e., they consume and produce relations. Figure 10 shows the output of the query "count of employees by department." The "by-list" or grouping attributes are the key of the new relation, the Department attribute in this example.

Algorithms for aggregate functions require grouping, e.g., employee items may be grouped by department, and then one

Department	Count
Toy	3
Shoe	9
Hardware	7

Figure 10. Count of employees by department.

output item is calculated per group. This grouping process is very similar to duplicate removal in which equal data items must be brought together, compared, and removed. Thus, aggregate functions and duplicate removal are typically implemented in the same module. There are only two differences between aggregate functions and duplicate removal. First, in duplicate removal, items are compared on all their attributes, but only on the attributes in the by-list of aggregate functions. Second, an identical item is immediately dropped from further consideration in duplicate removal whereas in aggregate functions some computation is performed before the second item of the same group is dropped. Both differences can easily be dealt with using a switch in an actual algorithm implementation. Because of their similarity, duplicate removal and aggregation are described and used interchangeably here.

In most existing commercial relational systems, aggregation and duplicate removal algorithms are based on sorting, following Epstein's [1979] work. Since aggregation requires that all data be consumed before any output can be produced, and since main memories were significantly smaller 15 years ago when the prototypes of these systems were designed, these implementations used temporary files for output, not streams and iterator algorithms. However, there is no reason why aggregation and duplicate removal cannot be implemented using iterators exploiting today's memory sizes.

4.1 Aggregation Algorithms Based on Nested Loops

There are three types of algorithms for aggregation and duplicate removal based on nested loops, sorting, and hashing. The first algorithm, which we call nested-loops aggregation, is the most simple-minded one. Using a temporary file to accumulate the output, it loops for each input item over the output file accumulated so far and either aggregates the input item into the appropriate output item or creates a new output item and appends it to the output file. Obviously, this algorithm is quite inefficient for large inputs, even if some performance enhancements can be applied.[7] We mention it here because it corresponds to the algorithm choices available for relational joins and other binary matching problems (discussed in the next section), which are the nested-loops join and the more efficient sort-based and hash-based join algorithms. As for joins and binary matching, where the nested-loops algorithm is the only algorithm that can evaluate any join predicate, the nested-loops aggregation algorithm can support unusual aggregations where the input items are not divided into disjoint equivalence classes but where a single input item may contribute to multiple output items. While such aggregations are not supported in today's database systems, classifications that do not divide the input into equivalence classes can be useful in both commercial and scientific applications. If the number of classifications is small enough that all output items can be kept in memory, the performance of this algorithm is acceptable. However, for the more standard database aggregation problems, sort-based and hash-based duplicate removal and aggregation algorithms are more appropriate.

[7] The possible improvements are (i) looping over pages or clusters rather than over records of input and output items (block nested loops), (ii) speeding the inner loop by an index (index nested loops), a method that has been used in some commercial relational systems, (iii) bit vector filtering to determine without inner loop or index lookup that an item in the outer loop cannot possibly have a match in the inner loop. All three of these issues are discussed later in this survey as they apply to binary operations such as joins and intersection.

4.2 Aggregation Algorithms Based on Sorting

Sorting will bring equal items together, and duplicate removal will then be easy. The cost of duplicate removal is dominated by the sort cost, and the cost of this naive duplicate removal algorithm based on sorting can be assumed to be that of the sort operation. For aggregation, items are sorted on their grouping attributes.

This simple method can be improved by detecting and removing duplicates as early as possible, easily implemented in the routines that write run files during sorting. With such "early" duplicate removal or aggregation, a run file can never contain more items than the final output (because otherwise it would contain duplicates!), which may speed up the final merges significantly [Bitton and DeWitt 1983].

As for any external sort operation, the optimizations discussed in the section on sorting, namely read-ahead using forecasting, merge optimizations, large cluster sizes, and reduced final fan-in for binary consumer operations, are fully applicable when sorting is used for aggregation and duplicate removal. However, to limit the complexity of the formulas, we derive I/O cost formulas without the effects of these optimizations.

The amount of I/O in sort-based aggregation is determined by the number of merge levels and the effect of early duplicate removal on each merge step. The total number of merge levels is unaffected by aggregation; in sorting with quicksort and without optimized merging, the number of merge levels is $L = \lceil \log_F(R/M) \rceil$ for input size R, memory size M, and fan-in F. In the first merge levels, the likelihood is negligible that items of the same group end up in the same run file, and we therefore assume that the sizes of run files are unaffected until their sizes would exceed the size of the final output. Runs on the first few merge levels are of size $M \times F^i$ for level i, and runs of the last levels have the same size as the final output. Assuming the output cardinality (number of items) is G times less than the input cardinality ($G = R/O$), where G is called the average group size or the reduction factor, only the last $\lceil \log_F(G) \rceil$ merge levels, including the final merge, are affected by early aggregation because in earlier levels more than G runs exist, and items from each group are distributed over all those runs, giving a negligible chance of early aggregation.

In the first merge levels, all input items participate, and the cost for these levels can be determined without explicitly calculating the size and number of run files on these levels. In the affected levels, the size of the output runs is constant, equal to the size of the final output $O = R/G$, while the number of run files decreases by a factor equal to the fan-in F in each level. The number of affected levels that create run files is $L_2 = \lceil \log_F(G) \rceil - 1$; the subtraction of 1 is necessary because the final merge does not create a run file but the output stream. The number of unaffected levels is $L_1 = L - L_2$. The number of input runs is W/F^i on level i (recall the number of initial runs $W = R/M$ from the discussion of sorting). The total cost,[8] including a factor 2 for writing and reading, is

$$2 \times R \times L_1 + 2 \times O \times \sum_{i=L_1}^{L-1} W/F^i$$
$$= 2 \times R \times L_1 + 2 \times O \times W \times (1/F^{L_1} - 1/F^L)/(1 - 1/F).$$

For example, consider aggregating R = 100 MB input into O = 1 MB output (i.e., reduction factor G = 100) using a system with M = 100 KB memory and fan-in F = 10. Since the input is W = 1,000 times the size of memory, L = 3 merge levels will be needed. The last $L_2 = \log_F(G) - 1 = 1$ merge level into temporary run files will permit early aggregation. Thus, the total I/O will be

[8] Using $\sum_{i=0}^{N} a^i = (1 - a^{N+1})/(1 - a)$ and $\sum_{i=K}^{N} a^i = \sum_{i=0}^{N} a^i - \sum_{i=0}^{K-1} a^i = (a^K - a^{N+1})/(1 - a)$.

$$2 \times 100 \times 2 + 2 \times 1 \times 1000 \times (1/10^2 - 1/10^3)/(1 - 1/10) = 400 + 2 \times 1000 \times 0.009/0.9 = 420 \text{ MB}$$

which has to be divided by the cluster size used and multiplied by the time to read or write a cluster to estimate the I/O time for aggregation based on sorting. Naive separation of sorting and subsequent aggregation would have required reading and writing the entire input file three times, for a total of 600 MB I/O. Thus, early aggregation realizes almost 30% savings in this case.

Aggregate queries may require that duplicates be removed from the input set to the aggregate functions,[9] e.g., if the SQL *distinct* keyword is used. If such an aggregate function is to be executed using sorting, early aggregation can be used only for the duplicate removal part. However, the sort order used for duplicate removal can be suitable to permit the subsequent aggregation as a simple filter operation on the duplicate removal's output stream.

4.3 Aggregation Algorithms Based on Hashing

Hashing can also be used for aggregation by hashing on the grouping attributes. Items of the same group (or duplicate items in duplicate removal) can be found and aggregated when inserting them into the hash table. Since only output items, not input items, are kept in memory, hash table overflow occurs only if the output does not fit into memory. However, if overflow does occur, the partition files (all partitioning files in any one recursion level) will basically be as large as the entire input because once a partition is being written to disk, no further aggregation can occur until the partition files are read back into memory.

The amount of I/O for hash-based aggregation depends on the number of partitioning (recursion) levels required before the output (not the input) of one partition fits into memory. This will be the case when partition files have been reduced to the size $G \times M$. Since the partitioning files shrink by a factor of F at each level (presuming hash value skew is absent or effectively counteracted), the number of partitioning (recursion) levels is $\lceil \log_F(R/G/M) \rceil = \lceil \log_F(O/M) \rceil$ for input size R, output size O, reduction factor G, and fan-out F. The costs at each level are proportional to the input file size R. The total I/O volume for hashing with overflow avoidance, including a factor of 2 for writing and reading, is

$$2 \times R \times \lceil \log_F(O/M) \rceil.$$

The last partitioning level may use hybrid hashing, i.e., it may not involve I/O for the entire input file. In that case, $L = \lfloor \log_F(O/M) \rfloor$ complete recursion levels involving all input records are required, partitioning the input into files of size $R' = R/F^L$. In each remaining hybrid hash aggregation, the size limit for overflow files is $M \times G$ because such an overflow file can be aggregated in memory. The number of partition files K must satisfy $K \times M \times G + (M - K \times C) \times G \geq R'$, meaning $K = \lceil (R'/G - M)/(M - C) \rceil$ partition files will be created. The total I/O cost for hybrid hash aggregation is

$$2 \times R \times L + 2 \times F^L \times (R' - (M - K \times C) \times G) = 2 \times (R \times (L + 1) - F^L \times (M - K \times C) \times G) = 2 \times (R \times (L + 1) - F^L \times (M - \lceil (R'/G - M)/(M - C) \rceil \times C) \times G).$$

[9] Consider two queries, both counting salaries per department. In order to determine the number of (salaried) employees per department, all salaries are counted without removing duplicate salary values. On the other hand, in order to assess salary differentiation in each department, one might want to determine the number of distinct salary levels in each department. For this query, only distinct salaries are counted, i.e., duplicate department-salary pairs must be removed prior to counting. (This refers to the latter type of query.)

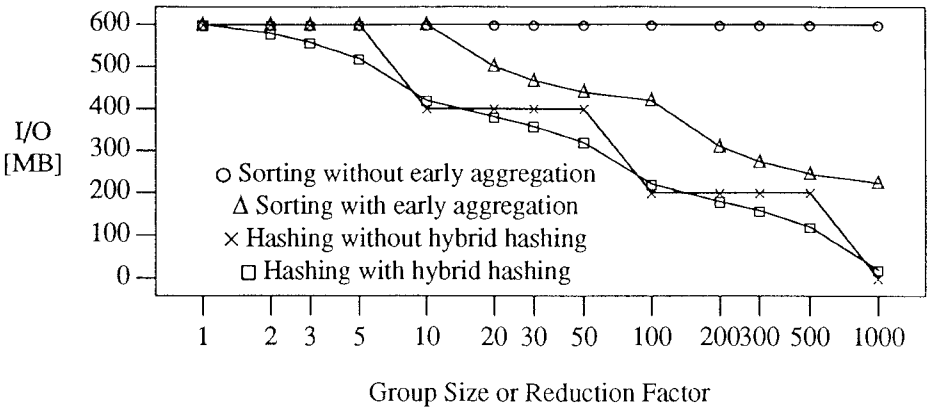


Figure 11. Performance of sort- and hash-based aggregation.

As for sorting, if an aggregate query requires duplicate removal for the input set to the aggregate function,[10] the group size or reduction factor of the duplicate removal step determines the performance of hybrid hash duplicate removal. The subsequent aggregation can be performed after the duplicate removal as an additional operation within each hash bucket or as a simple filter operation on the duplicate removal's output stream.

4.4 A Rough Performance Comparison

It is interesting to note that the performance of both sort-based and hash-based aggregation is logarithmic and improves with increasing reduction factors. Figure 11 compares the performance of sort- and hash-based aggregation[11] using the formulas developed above for 100 MB input data, 100 KB memory, clusters of 8 KB, fan-in or fan-out of 10, and varying group sizes or reduction factors. The output size is the input size divided by the group size.

It is immediately obvious in Figure 11 that sorting without early aggregation is not competitive because it does not limit the sizes of run files, confirming the results of Bitton and DeWitt [1983]. The other algorithms all exhibit similar, though far from equal, performance improvements for larger reduction factors. Sorting with early aggregation improves once the reduction factor is large enough to affect not only the final but also previous merge steps. Hashing without hybrid hashing improves in steps as the number of partitioning levels can be reduced, with "step" points where $G = F^i$ for some i. Hybrid hashing exploits all available memory to improve performance and generally outperforms overflow avoidance hashing. At points where overflow avoidance hashing shows a step, hybrid hashing has no effect, and the two hashing schemes have the same performance.

While hash-based aggregation and duplicate removal seem superior in this rough analytical performance comparison, recall that the cost formula for sort-based aggregation does not include the effects of replacement selection or the merge optimizations discussed earlier in the section on sorting; therefore, Figure 11 shows an upper bound for the I/O cost of sort-based aggregation and duplicate removal. Furthermore, since the cost formula for hashing presumes optimal assignments of hash buckets to output partitions, the real costs of sort- and hash-based aggregation will be much more similar than they appear in Figure 11. The important point is that both their

[10] See footnote 9.

[11] Aggregation by nested-loops methods is omitted from Figure 11 because it is not competitive for large data sets.

costs are logarithmic with the input size, improve with the group size or reduction factor, and are quite similar overall.

4.5 Additional Remarks on Aggregation

Some applications require multilevel aggregation. For example, a report generation language might permit a request like "sum (employee.salary by employee.id by employee.department by employee.division)" to create a report with an entry for each employee and a sum for each department and each division. In fact, specifying such reports concisely was the driving design goal for the report generation language RPG. In SQL, this requires multiple cursors within an application program, one for each level of detail. This is very undesirable for two reasons. First, the application program performs what is essentially a join of three inputs. Such joins should be provided by the database system, not required to be performed within application programs. Second, the database system more likely than not executes the operations for these cursors independently from one another, resulting in three sort operations on the employee file instead of one.

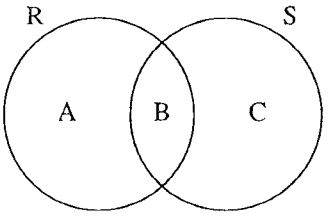
If complex reporting applications are to be supported, the query language should support direct requests (perhaps similar to the syntax suggested above), and the sort operator should be implemented such that it can perform the entire operation in a single sort and one final pass over the sorted data. An analogous algorithm based on hashing can be defined; however, if the aggregated data are required in sort order, sort-based aggregation will be the algorithm of choice.

For some applications, exact aggregate functions are not required; reasonably close approximations will do. For example, exploratory (rather than final precise) data analysis is frequently very useful in "approaching" a new set of data [Tukey 1977]. In real-time systems, precision and response time may be reasonable tradeoffs. For database query optimization, approximate statistics are a sufficient basis for selectivity estimation, cost calculation, and comparison of alternative plans. For these applications, faster algorithms can be designed that rely either on a single sequential scan of the data (no run files, no overflow files) or on sampling [Astrahan et al. 1987; Hou and Ozsoyoglu 1991; 1993; Hou et al. 1991].

5. BINARY MATCHING OPERATIONS

While aggregation is essential to *condense* information, there are a number of database operations that *combine* information from two inputs, files, or sets and therefore are essential for database systems' ability to provide more than reliable shared storage and to perform inferences, albeit limited. A group of operators that all do basically the same task are called the one-to-one match operations here because an input item contributes to the output depending on its match with one other item. The most prominent among these operations is the relational join. Mishra and Eich [1992] have recently written a survey of join algorithms, which includes an interesting analysis and comparison of algorithms focusing on how data items from the two inputs are compared with one another. The other one-to-one match operations are left and right semi-joins, left, right, and symmetric outer-joins, left and right anti-semi-joins, symmetric anti-join, intersection, union, left and right differences, and symmetric or anti-difference.[12] Figure 12 shows the basic

[12] The anti-semijoin of R and S is $R\ \overline{SEMIJOIN}\ S = R - (R\ SEMIJOIN\ S)$, i.e., the items in R without matches in S. The (symmetric) anti-join contains those items from both inputs that do not have matches, suitably padded as in outer joins to make them union compatible. Formally, the (symmetric) anti-join of R and S is $R\ \overline{JOIN}\ S = (R\ \overline{SEMIJOIN}\ S) \cup (S\ \overline{SEMIJOIN}\ R)$ with the tuples of the two union arguments suitably extended with *null* values. The symmetric or anti-difference is the union of the two differences. Formally the anti-difference of R and S is $(R \cup S) - (R \cap S) = (R - S) \cup (S - R)$ [Maier 1983]. Among these three operations, the anti-semijoin is probably the most useful one, as in the query to "find the courses that don't have any enrollment."



Output	Match on all Attributes	Match on some Attributes
A	Difference	Anti-semi-join
B	Intersection	Join, semi-join
C	Difference	Anti-semi-join
A, B		Left outer join
A, C	Symmetric difference	Anti-join
B, C		Right outer join
A, B, C	Union	Symmetric outer join

Figure 12. Binary one-to-one matching.

principle underlying all these operations, namely separation of the matching and nonmatching components of two sets, called R and S in the figure, and production of appropriate subsets, possibly after some transformation and combination of records as in the case of a join. If the sets R and S have different schemas as in relational joins, it might make sense to think of the set B as two sets B_R and B_S, i.e., the matching elements from R and S. This distinction permits a clearer definition of left semi-join and right semi-join, etc. Since all these operations require basically the same steps and can be implemented with the same algorithms, it is logical to implement them in one general and efficient module. For simplicity, only join algorithms are discussed here. Moreover, we discuss algorithms for only one join attribute since the algorithms for multi-attribute joins (and their performance) are not different from those for single-attribute joins.

Since set operations such as intersection and difference will be used and must be implemented efficiently for any data model, this discussion is relevant to relational, extensible, and object-oriented database systems alike. Furthermore, binary matching problems occur in some surprising places. Consider an object-oriented database system that uses a table to map logical object identifiers (OIDs) to physical locations (record identifiers or RIDs). Resolving a set of OIDs to RIDs can be regarded (as well as optimized and executed) as a semi-join of the mapping table and the set of OIDs, and all conventional join strategies can be employed. Another example that can occur in a database management system for any data model is the use of multiple indices in a query: the pointer (OID or RID) lists obtained from the indices must be intersected (for a conjunction) or united (for a disjunction) to obtain the list of pointers to items that satisfy the whole query. Moreover, the actual lookup of the items using the pointer list can be regarded as a semi-join of the underlying data set and the list, as in Kooi's [1980] thesis and the Ingres product [Kooi and Frankforth 1982] and a recent study by Shekita and Carey [1990]. Finally, many *path expressions* in object-oriented database systems such as "employee.department.manager.office.location" can frequently be interpreted, optimized, and executed as a sequence of one-to-one match operations using existing join and semi-join algorithms. Thus, even if rela-

tional systems were completely abolished and replaced by object-oriented database systems, set matching and join techniques developed in the relational context would continue to be important for the performance of database systems.

Most of today's commercial database systems use only nested loops and merge-join because an analysis performed in connection with the System R project determined that of all the join methods considered, one of these two always provided either the best or very close to the best performance [Blasgen and Eswaran 1976; 1977]. However, the System R study did not consider hash join algorithms, which are now regarded as more efficient in many cases.

There continues to be a strong interest in join techniques, although the interest has shifted over the last 20 years from basic algorithmic concerns to parallel techniques and to techniques that adapt to unpredictable run-time situations such as data skew and changing resource availability. Unfortunately, many new proposed techniques fail a very simple test (which we call the "Guy Lohman test for join techniques" after the first person who pointed this test out to us), making them problematic for nontrivial queries. The crucial test question is: Does this new technique apply to joining three inputs without interrupting data flow between the join operators? For example, a technique fails this test if it requires materializing the entire intermediate join result for random sampling of both join inputs or for obtaining exact knowledge about both join input sizes. Given its importance, this test should be applied to both proposed query optimization and query execution techniques.

For the I/O cost formulas given here, we assume that the left and right inputs have R and S pages, respectively, and that the memory size is M pages. We assume that the algorithms are implemented as iterators and omit the cost of reading stored inputs and writing an operation's output from the cost formulas because both inputs and output may be iterators, i.e., these intermediate results are never written to disk, and because these costs are equal for all algorithms.

5.1 Nested-Loops Join Algorithms

The simplest and, in some sense, most direct algorithm for binary matching is the nested-loops join: for each item in one input (called the outer input), scan the entire other input (called the inner input) and find matches. The main advantage of this algorithm is its simplicity. Another advantage is that it can compute a Cartesian product and any Θ-join of two relations, i.e., a join with an arbitrary two-relation comparison predicate. However, Cartesian products are avoided by query optimizers because their outputs tend to contain many data items that will eventually not satisfy a query predicate verified later in the query evaluation plan.

Since the inner input is scanned repeatedly, it must be stored in a file, i.e., a temporary file if the inner input is produced by a complex subplan. This situation does not change the cost of nested loops; it just replaces the first read of the inner input with a write.

Except for very small inputs, the performance of nested-loops join is disastrous because the inner input is scanned very often, once for each item in the outer input. There are a number of improvements that can be made to this *naive nested-loops join*. First, for one-to-one match operations in which a single match carries all necessary information, e.g., semi-join and intersection, a scan of the inner input can be terminated after the first match for an item of the outer input. Second, instead of scanning the inner input once for each item from the outer input, the inner input can be scanned once for each page of the outer input, an algorithm called *block nested-loops join* [Kim 1980]. Third, the performance can be improved further by filling all of memory except K pages with pages of the outer input and by using the remaining K pages to scan the inner input and to save pages of the inner input in memory. Finally, scans of the inner input can be

made a little faster by scanning the inner input alternatingly forward and backward, thus reusing the last page of the previous scan and therefore saving one I/O per inner scan. The I/O cost for this version of nested-loops join is the product of the number of scans (determined by the size of the outer input) and the cost per scan of the inner input, plus K I/Os because the first inner scan has to scan or save the entire inner input. Thus, the total cost for scanning the inner input repeatedly is $\lceil R/(M-K) \rceil \times (S-K) + K$. This expression is minimized if $K = 1$ and $R \geq S$, i.e., the larger input should be the outer.

If the critical performance measure is not the amount of data read in the repeated inner scans but the number of I/O operations, more than one page should be moved in each I/O, even if more memory has to be dedicated to the inner input and less to the outer input, thus increasing the number of passes over the inner input. If C pages are moved in each I/O on the inner input and $M - C$ pages for the outer input, the number of I/Os is $\lceil R/(M-C) \rceil \times (S/C) + 1$, which is minimized if $C = M/2$. In other words, in order to minimize the number of large-chunk I/O operations, the cluster size should be chosen as half the available memory size [Hagmann 1986].

Finally, index nested-loops join exploits a permanent or temporary index on the inner input's join attribute to replace file scans by index lookups. In principle, each scan of the inner input in naive nested-loops join is used to find matches, i.e., to provide associativity. Not surprisingly, since all index structures are designed and used for the purpose of associativity, any index structure supporting the join predicate (such as $=$, $\leq$, etc.) can be used for index nested-loops join. The fastest indices for exact match queries are hash indices, but any index structure can be used, ordered or unordered (hash), single- or multi-attribute, single- or multidimensional. Therefore, indices on frequently used join attributes (keys and foreign keys in relational systems) may be useful. Index nested-loops join is also used sometimes with indices built on the fly, i.e., indices built on intermediate query processing results.

A recent investigation by DeWitt et al. [1993] demonstrated that index nested-loops join can be the fastest join method if one of the inputs is so small and if the other indexed input is so large that the number of index and data page retrievals, i.e., about the product of the index depth and the cardinality of the smaller input, is smaller than the number of pages in the larger input.

Another interesting idea using two ordered indices, e.g., a B-tree on each of the two join columns, is to switch roles of inner and outer join inputs after each index lookup, which leads to the name "zig-zag join." For example, for a join predicate $R.a = S.a$, a scan in the index on $R.a$ finds the lower join attribute value in R, which is then looked up in the index on $S.a$. A continuing scan in the index on $S.a$ yields the next possible join attribute value, which is looked up in the index on $R.a$, etc. It is not immediately clear under which circumstances this join method is most efficient.

For complex queries, N-ary joins are sometimes written as a single module, i.e., a module that performs index lookups into indices of multiple relations and joins all relations simultaneously. However, it is not clear how such a multi-input join implementation is superior to multiple index nested-loops joins.

5.2 Merge-Join Algorithms

The second commonly used join method is the merge-join. It requires that both inputs are sorted on the join attribute. Merging the two inputs is similar to the merge process used in sorting. An important difference, however, is that one of the two merging scans (the one which is advanced on equality, usually called the inner input) must be backed up when both inputs contain duplicates of a join attribute value and when the specific one-to-one match operation requires that all matches be found, not just one match. Thus, the control logic for merge-join variants for join and semi-join are slightly different. Some systems include the no-

tion of "value packet," meaning all items with equal join attribute values [Kooi 1980; Kooi and Frankforth 1982]. An iterator's *next* call returns a value packet, not an individual item, which makes the control logic for merge-join much easier. If (or after) both inputs have been sorted, the merge-join algorithm typically does not require any I/O, except when "value packets" are larger than memory. (See footnote 1.)

An input may be sorted because a stored database file was sorted, an ordered index was used, an input was sorted explicitly, or the input came from an operation that produced sorted output, e.g., another merge-join. The last point makes merge-join an efficient algorithm if items from multiple sources are matched on the same join attribute(s) in multiple binary steps because sorting intermediate results is not required for later merge-joins, which led to the concept of *interesting orderings* in the System R query optimizer [Selinger et al. 1979]. Since set operations such as intersection and union can be evaluated using any sort order, as long as the same sort order is present in both inputs, the effect of interesting orderings for one-to-one match operators based on merge-join can always be exploited for set operations.

A combination of nested-loops join and merge-join is the *heap-filter merge-join* [Graefe 1991]. It first sorts the smaller inner input by the join attribute and saves it in a temporary file. Next, it uses all available memory to create sorted runs from the larger outer input using replacement selection. As discussed in the section on sorting, there will be about $W = R/(2 \times M) + 1$ such runs for outer input size R. These runs are not written to disk; instead, they are joined immediately with the sorted inner input using merge-join. Thus, the number of scans of the inner input is reduced to about one half when compared to block nested loops. On the other hand, when compared to merge-join, it saves writing and reading temporary files for the larger outer input.

Another derivation of merge-join is the hybrid join used in IBM's DB2 product [Cheng et al. 1991], combining elements from index nested-loops join, merge-join, and techniques joining sorted lists of index leaf entries. After sorting the outer input on its join attribute, hybrid join uses a merge algorithm to "join" the outer input with the leaf entries of a preexisting B-tree index on the join attribute of the inner input. The result file contains entire tuples from the outer input and record identifiers (RIDs, physical addresses) for tuples of the inner input. This file is then sorted on the physical locations, and the tuples of the inner relation can then be retrieved from disk very efficiently. This algorithm is not entirely new as it is a special combination of techniques explored by Blasgen and Eswaran [1976; 1977], Kooi [1980], and Whang et al. [1984; 1985]. Blasgen and Eswaran considered the manipulation of RID lists but concluded that either merge-join or nested-loops join is the optimal choice in almost all cases; based on this study, only these two algorithms were implemented in System R [Astrahan et al. 1976] and subsequent relational database systems. Kooi's optimizer treated an index similarly to a base relation and the lookup of data records from index entries as a join; this naturally permitted joining two indices or an index with a base relation as in hybrid join.

5.3 Hash Join Algorithms

Hash join algorithms are based on the idea of building an in-memory hash table on one input (the smaller one, frequently called the *build input*) and then probing this hash table using items from the other input (frequently called the *probe input*). These algorithms have only recently found greater interest [Bratbergsengen 1984; DeWitt et al. 1984; DeWitt and Gerber 1985; DeWitt et al. 1986; Fushimi et al. 1986; Kitsuregawa et al. 1983; 1989a; Nakayama et al. 1988; Omiecinski 1991; Schneider and DeWitt 1989; Shapiro 1986; Zeller and Gray 1990]. One reason is that they work very fast, i.e., without any temporary files, if the build input does indeed fit into memory, independently of the size of the probe input.

However, they require overflow avoidance or resolution methods for larger build inputs, and suitable methods were developed and experimentally verified only in the mid-1980's, most notably in connection with the Grace and Gamma database machine projects [DeWitt et al. 1986; 1990; Fushimi et al. 1986; Kitsuregawa et al. 1983].

In hash-based join methods, build and probe inputs are partitioned using the same partitioning function, e.g., the join key value modulo the number of partitions. The final join result can be formed by concatenating the join results of pairs of partitioning files. Figure 13 shows the effect of partitioning the two inputs of a binary operation such as join into hash buckets and partitions. (This figure was adapted from a similar diagram by Kitsuregawa et al. [1983]. Mishra and Eich [1992] recently adapted and generalized it in their survey and comparison of relational join algorithms.) Without partitioning, each item in the first input must be compared with each item in the second input; this would be represented by complete shading of the entire diagram. With partitioning, items are grouped into partition files, and only pairs in the series of small rectangles (representing the partitions) must be compared.

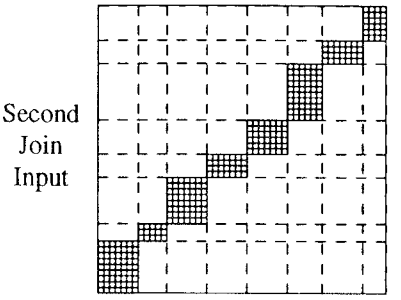


Figure 13. Effect of partitioning for join operations.

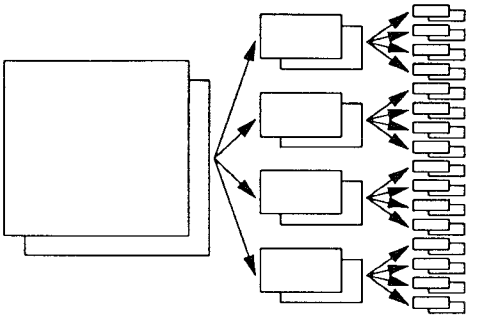


Figure 14. Recursive partitioning in binary operations.

If a build partition file is still larger than memory, recursive partitioning is required. Recursive partitioning is used for both build- and probe-partitioning files using the same hash and partitioning functions. Figure 14 shows how both input files are partitioned together. The partial results obtained from pairs of partition files are concatenated to form the result of the entire match operation. Recursive partitioning stops when the build partition fits into memory. Thus, the recursion depth of partitioning for binary match operators depends only on the size of the build input (which therefore should be chosen to be the smaller input) and is independent of the size of the probe input. Compared to sort-based binary matching operators, i.e., variants of merge-join in which the number of merge levels is determined for each input file individually, hash-based binary matching operators are particularly effective when the input sizes are very different [Bratbergsengen 1984; Graefe et al. 1993].

The I/O cost for binary hybrid hash operations can be determined by the number of complete levels (i.e., levels without hash table) and the fraction of the input remaining in memory in the deepest recursion level. For memory size M, cluster size C, partitioning fan-out $F = \lfloor M/C - 1 \rfloor$, build input size R, and probe input size S, the number of complete levels is $L = \lfloor \log_F(R/M) \rfloor$, after which the build input partitions should be of size $R' = R/F^L$. The I/O cost for the binary operation is the cost of partitioning the build input divided by the size of the build input and multiplied by the sum of the input sizes. Adapting the

cost formula for unary hashing discussed earlier, the total amount of I/O for a recursive binary hash operation is

$$2 \times (R \times (L + 1) - F^L \times (M - \lceil (R' - M + C)/(M - C) \rceil \times C))/R \times (R + S)$$

which can be approximated with $2 \times \log_F(R/M) \times (R + S)$. In other words, the cost of binary hash operations on large inputs is logarithmic; the main difference to the cost of merge-join is that the recursion depth (the logarithm) depends only on one file, the build input, and is not taken for each file individually.

As for all operations based on partitioning, partitioning (hash) value skew is the main danger to effectiveness. When using statistics on hash value distributions to determine which buckets should stay in memory in hybrid hash algorithms, the goal is to avoid as much I/O as possible with the least memory "investment." Thus, it is most effective to retain those buckets in memory with few build items but many probe items or, more formally, the buckets with the smallest value for $r_i/(r_i + s_i)$ where r_i and s_i indicate the total size of a bucket's build and probe items [Graefe 1993b].

5.4 Pointer-Based Joins

Recently, links between data items have found renewed interest, be it in object-oriented systems in the form of object identifiers (OIDs) or as access paths for faster execution of relational joins. In a sense, links represent a limited form of precomputed results, somewhat similar to indices and join indices, and have the usual cost-versus-benefit tradeoff between query performance enhancement and maintenance effort. Kooi [1980] modeled the retrieval of actual records after index searches as "TID joins" (tuple identifiers permitting direct record access) in his query optimizer for Ingres; together with standard join commutativity and associativity rules, this model permitted exploring joins of indices of different relations (joining lists of key-TID pairs) or joins of one relation with another relation's index. In the Genesis data model and database system, Batory et al. [1988a; 1988b] modeled joins in a functional way, borrowing from research into the database languages FQL [Buneman et al. 1982; Buneman and Frankel 1979], DAPLEX [Shipman 1981], and Gem [Tsur and Zaniolo 1984; Zaniolo 1983] and permitting pointer-based join implementations in addition to traditional value-based implementations such as nested-loops join, merge-join, and hybrid hash join.

Shekita and Carey [1990] recently analyzed three pointer-based join methods based on nested-loops join, merge-join, and hybrid hash join. Presuming relations R and S, with a pointer to an S tuple embedded in each R tuple, the nested-loops join algorithm simply scans through R and retrieves the appropriate S tuple for each R tuple. This algorithm is very reminiscent of unclustered index scans and performs similarly poorly for larger set sizes. Their conclusion on naive pointer-based join algorithms is that "it is unwise for object-oriented database systems to support only pointer-based join algorithms."

The merge-join variant starts with sorting R on the pointers (i.e., according to the disk address they point to) and then retrieves all S items in one elevator pass over the disk, reading each S page at most once. Again, this idea was suggested before for unclustered index scans, and variants similar to heap-filter merge-join [Graefe 1991] and complex object assembly using a window and priority heap of open references [Keller et al. 1991] can be designed.

The hybrid hash join variant partitions only relation R on pointer values, ensuring that R tuples with S pointers to the same page are brought together, and then retrieves S pages and tuples. Notice that the two relations' roles are fixed by the direction of the pointers, whereas for standard hybrid hash join the smaller relation should be the build input. Differ-

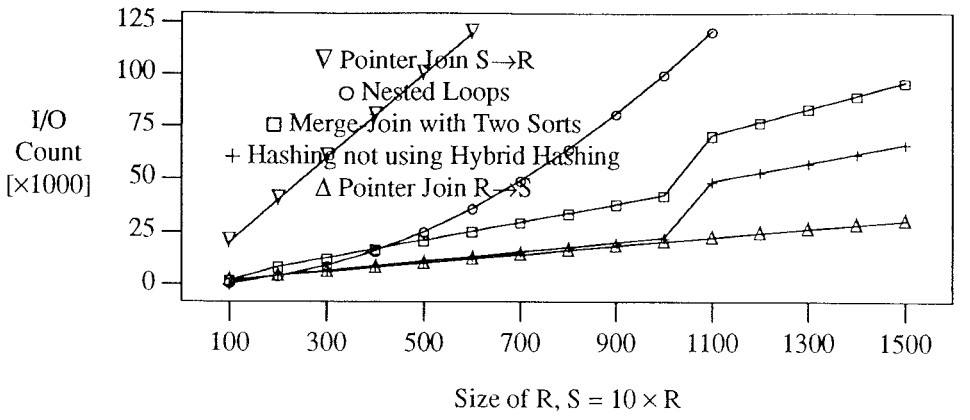


Figure 15. Performance of alternative join methods

ently than standard hybrid hash join, relation S is not partitioned. This algorithm performs somewhat faster than pointer-based merge-join if it keeps some partitions of R in memory and if sorting writes all R tuples into runs before merging them.

Pointer-based join algorithms tend to outperform their standard value-based counterparts in many situations, in particular if only a small fraction of S actually participates in the join and can be selected effectively using the pointers in R. Historically, due to the difficulty of correctly maintaining pointers (nonessential links), they were rejected as a relational access method in System R [Chamberlin et al. 1981a] and subsequently in basically all other systems, perhaps with the exception of Kooi's modified Ingres [Kooi 1980; Kooi and Frankforth 1982]. However, they were reevaluated and implemented in the Starburst project, both as a test of Starburst's extensibility and as a means of supporting "more object-oriented" modes of operation [Haas et al. 1990].

5.5 A Rough Performance Comparison

Figure 15 shows an approximate performance comparison using the cost formulas developed above for block nested-loops join; merge-join with sorting both inputs without optimized merging; hash join without hybrid hashing, bucket tuning, or dynamic destaging; and pointer joins with pointers from R to S and from S to R without grouping pointers to the same target page together. This comparison is not precise; its sole purpose is to give a rough idea of the relative performance of the algorithm groups, deliberately ignoring the many tricks used to improve and fine-tune the basic algorithms. The relation sizes vary; S is always 10 times larger than R. The memory size is 100 KB; the cluster size is 8 KB; merge fan-in and partitioning fan-out are 10; and the number of R-records per cluster is 20.

It is immediately obvious in Figure 15 that nested-loops join is unsuitable for medium-size and large relations, because the cost of nested-loops join is proportional to the size of the Cartesian product of the two inputs. Both merge-join (sorting) and hash join have logarithmic cost functions; the sudden rise in merge-join and hash join cost around $R = 1000$ is due to the fact that additional partitioning or merging levels become necessary at that point. The sort-based merge-join is not quite as fast as hash join because the merge levels are determined individually for each file, including the bigger S file, while only the smaller build relation R determines the partitioning depth of hash join. Pointer joins are competitive with hash and merge-joins due to their linear cost func-

tion, but only when the pointers are embedded in the smaller relation R. When S-records point to R-records, the cost of the pointer join is even higher than for nested-loops join.

The important point of Figure 15 is to illustrate that pointer joins can be very efficient or very inefficient, that one-to-one match algorithms based on nested-loops join are not competitive for medium-size and large inputs, and that sort- and hash-based algorithms for one-to-one match operations both have logarithmic cost growth. Of course, this comparison is quite naive since it uses only the simplest form of each algorithm. Thus, a comparison among alternative algorithms in a query optimizer must use the precise cost function for the available algorithm variant.

6. UNIVERSAL QUANTIFICATION[13]

Universal quantification permits queries such as "find the students who have taken *all* database courses"; the difference to one-to-one match operations is that a student qualifies because his or her transcript matches an entire set of courses, not only one item as in an existentially quantified query (e.g., "find students who have taken a (at least one) database course") that can be executed using a semi-join. In the past, universal quantification has been largely ignored for four reasons. First, typical database applications, e.g., record-keeping and accounting applications, rarely require universal quantification. Second, it can be circumvented using a complex expression involving a Cartesian product. Third, it can be circumvented using complex aggregation expressions. Fourth, there seemed to be a lack of efficient algorithms.

The first reason will not remain true for database systems supporting logic programming, rules, and quantifiers, and algorithms for universal quantification will become more important. The second reason is valid; however, the substitute expressions are very slow to execute because of the Cartesian product. The third reason is also valid, but replacing a universal quantifier may require very complex aggregation clauses that are easy to "get wrong" for the database user. Furthermore, they might be too complex for the optimizer to recognize as universal quantification and to execute with a direct algorithm. The fourth reason is not valid; universal quantification algorithms can be very efficient (in fact, as fast as semi-join, the operator for existential quantification), useful for very large inputs, and easy to parallelize. In the remainder of this section, we discuss sort- and hash-based direct and indirect (aggregation-based) algorithms for universal quantification.

In the relational world, universal quantification is expressed with the universal quantifier in relational calculus and with the division operator in relational algebra. We will explain algorithms for universal quantification using relational terminology. The running example in this section uses the relations *Student* (*student-id*, *name*, *major*), *Course* (*course-no*, *title*), *Transcript* (*student-id*, *course-no*, *grade*), and *Requirement* (*major*, *course-no*) with the obvious key attributes. The query to find the students who have taken all courses can be expressed in relational algebra as

$$\pi_{student\text{-}id,\, course\text{-}no} Transcript \div \pi_{course\text{-}no} Course.$$

The projection of the *Transcript* relation is called the dividend, the projection of the *Course* relation the divisor, and the result relation the quotient. The quotient attributes are those attributes of the dividend that do not appear in the divisor. The dividend relation semi-joined with the divisor relation and projected on the quotient attributes, in the example the set of *student-ids* of *Student*s who have taken at least one course, is called the set of quotient candidates here.

[13] This section is a summary of earlier work [Graefe 1989; Graefe and Cole 1993].

Some universal quantification queries seem to require relational division but actually do not. Consider the query for the students who have taken all courses required for their major. This query can be answered with a sequence of one-to-one match operations. A join of *Student* and *Requirement* projected on the *student-id* and *course-no* attributes minus the *Transcript* relation can be projected on *student-id*s to obtain a set of students who have not taken all their requirements. An anti-semi-join of the *Student* relation with this set finds the students who have satisfied all their requirements. This sequence will have acceptable performance because its required set-matching algorithms (join, difference, anti-semi-join) all belong to the family of one-to-one match operations, for which efficient algorithms are available as discussed in the previous section.

Division algorithms differ not only in their performance but also in how they fit into complex queries. Prior to the division, selections on the dividend, e.g., only *Transcript* entries with "A" grades, or on the divisor, e.g., only the database courses, may be required. Restrictions on the dividend can easily be enforced without much effect on the division operation, while restrictions on the divisor can imply a significant difference for the query evaluation plan. Subsequent to the division operation, the resulting quotient relation (e.g., a set of *student-id*s) may be joined with another relation, e.g., the *Student* relation to obtain student *names*. Thus, obtaining the quotient in a form suitable for further processing (e.g., join or semi-join with a third relation) can be advantageous.

Typically, universal quantification can easily be replaced by aggregations. (Intuitively, *all* universal quantification can be replaced by aggregation. However, we have not found a proof for this statement.) For example, the example query about database courses can be restated as "find the students who have taken as many database courses as there are database courses." When specifying the aggregate function, it is important to count only database courses both in the dividend (the *Transcript* relation) and in the divisor (the *Course* relation). Counting only database courses might be easy for the divisor relation, but requires a semi-join of the dividend with the divisor relation to propagate the restriction on the divisor to the dividend if it is not known a priori whether or not referential integrity holds between the dividend's divisor attributes and the divisor, i.e., whether or not there are divisor attribute values in the dividend that cannot be found in the divisor. For example, *course-no*s in the *Transcript* relation that do not pertain to database courses (and are therefore not in the divisor) must be removed from the dividend by a semi-join with the divisor. In general, if the divisor is the result of a prior selection, any referential integrity constraints known for stored relations will not hold and must be explicitly enforced using a semi-join. Furthermore, in order to ensure correct counting, duplicates have to be removed from either input if the inputs are projections on nonkey attributes.

There are four methods to compute the quotient of two relations, a sort- and a hash-based direct method, and sort- and hash-based aggregation. Table 6 shows this classification of relational division algorithms. Methods for sort- and hash-based aggregation and the possible sort- or hash-based semi-join have already been discussed, including their variants for inputs larger than memory and their cost functions. Therefore, we focus here on the direct division algorithms.

The sort-based direct method, proposed by Smith and Chang [1975] and called *naive division* here, sorts the divisor input on all its attributes and the dividend relation with the quotient attributes as major and the divisor attributes as minor sort keys. It then proceeds with a merging scan of the two sorted inputs to determine which items belong in the quotient. Notice that the scan can be programmed such that it ignores duplicates in either input (in case those had not been removed yet in the sort) as well as dividend items that do

Table 6. Classification of Relational Division Algorithms

	Based on Sorting	Based on Hashing
Direct	Naive divison	Hash-division
Indirect by semi-join and aggregation	Sorting with duplicate removal, merge-join, sorting with aggregation	Hash-based duplicate removal, hybrid hash join, hash-based aggregation

not refer to items in the divisor. Thus, neither a preceding semi-join nor explicit duplicate removal steps are necessary for naive division. The I/O cost of naive division is the cost of sorting the two inputs plus the cost of repeated scans of the divisor input.

Figure 16 shows two tables, a dividend and a divisor, properly sorted for naive division. Concurrent scans of the "Jack" tuples (only one) in the dividend and of the entire divisor determine that "Jack" is not part of the quotient because he has not taken the "Readings in Databases" course. A continuing scan through the "Jill" tuples in the dividend and a new scan of the entire divisor include "Jill" in the output of the naive division. The fact that "Jill" has also taken an "Intro to Graphics" course is ignored by a suitably general scan logic for naive division.

The hash-based direct method, called *hash-division*, uses two hash tables, one for the divisor and one for the quotient candidates. While building the divisor table, a unique sequence number is assigned to each divisor item. After the divisor table has been built, the dividend is consumed. For each quotient candidate, a bit map is kept with one bit for each divisor item. The bit map is indexed with the sequence numbers assigned to the divisor items. If a dividend item does not match with an item in the divisor table, it can be ignored immediately. Otherwise, a quotient candidate is either found or created, and the bit corresponding to the matching divisor item is set. When the entire dividend has been consumed, the quotient consists of those quotient candidates for which all bits are set.

This algorithm can ignore duplicates in the divisor (using hash-based duplicate removal during insertion into the divisor table) and automatically ignores duplicates in the dividend as well as dividend items that do not refer to items in the divisor (e.g., the AI course in the example). Thus, neither prior semi-join nor duplicate removal are required. However, if both inputs are known to be duplicate free, the bit maps can be replaced by counters. Furthermore, if referential integrity is known to hold, the divisor table can be omitted and replaced by a single counter. Hash-division, including these variants, has been implemented in the Volcano query execution engine and has shown better performance than the other three algorithms [Graefe 1989; Graefe and Cole 1993]. In fact, the performance of hash-division is almost equal to a hash-based join or semi-join of dividend and divisor relations (a semi-join corresponds to existential quantification), making universal quantification and relational division realistic operations and algorithms to use in database applications.

Student	Course
Jack	Intro to Databases
Jill	Intro to Databases
Jill	Intro to Graphics
Jill	Readings in Databases

Course
Intro to Databases
Readings in Databases

Figure 16. Sorted inputs into naive division

The aspect of hash-division that makes it an efficient algorithm is that the set of matches between a quotient candidate and the divisor is represented efficiently

using a bit map. Bit maps are one of the standard data structures to represent sets, and just as bit maps can be used for a number of set operations, the bit maps associated with each quotient candidate can also be used for a number of operations similar to relational division. For example, Carlis [1986] proposed a generalized division operator called "HAS" that included relational division as a special case. The hash-division algorithm can easily be extended to compute quotient candidates in the dividend that match a majority or given fraction of divisor items as well as (with one more bit in each bit map) quotient candidates that do or do not match exactly the divisor items.

For real queries containing a division, consider the operation that frequently follows a division. In the example, a user is typically not really interested in *student-id*s only but in information about the students. Thus, in many cases, relational division results will be used to select items from another relation using a semi-join. The sort-based algorithms produce their output sorted, which will facilitate a subsequent (semi-) merge-join. The hash-based algorithms produce their output in hash order; if overflow occurred, there is no predictable order at all. However, both aggregation-based and direct hash-based algorithms use a hash table on the quotient attributes, which may be used immediately for a subsequent (semi-) join. It seems quite straightforward to use the same hash table for the aggregation and a subsequent join as well as to modify hash-division such that it removes quotient candidates from the quotient table that do not belong to the final quotient and then performs a semi-join with a third input relation.

If the two hash tables do not fit into memory, the divisor table or the quotient table or both can be partitioned, and individual partitions can be held on disk for processing in multiple steps. In *divisor partitioning*, the final result consists of those items that are found in all partial results; the final result is the intersection of all partial results. For example, if the *Course* relations in the example above are partitioned into undergraduate and graduate courses, the final result consists of the students who have taken all undergraduate courses and all graduate courses, i.e., those that can be found in the division result of each partition. In *quotient partitioning*, the entire divisor must be kept in memory for all partitions. The final result is the concatenation (union) of all partial results. For example, if *Transcript* items are partitioned by odd and even *student-id*s, the final result is the union (concatenation) of all students with odd *student-id* who have taken all courses and those with even *student-id* who have taken all courses. If warranted by the input data, divisor partitioning and quotient partitioning can be combined.

Hash-division can be modified into an algorithm for duplicate removal. Consider the problem of removing duplicates from a relation R(X, Y) where X and Y are suitably chosen attribute groups. This relation can be stored using two hash tables, one storing all values of X (similar to the divisor table) and assigning each of them a unique sequence number, the other storing all values of Y and bit maps that indicate which X values have occurred with each Y value. Consider a brief example for this algorithm: Say relation R(X, Y) contains 1 million tuples, but only 100,000 tuples if duplicates were removed. Let X and Y be each 100 bytes long (total record size 200), and assume there are 4,000 unique values of each X and Y. For the standard hash-based duplicate removal algorithm, 100,000 × 200 bytes of memory are needed for duplicate removal without use of temporary files. For the redesigned hash-division algorithm, 2 × 4,000 × 100 bytes are needed for data values, 4,000 × 4 for unique sequence numbers, and 4,000 × 4,000 bits for bit maps. Thus, the new algorithm works efficiently with less than 3 MB of memory while conventional duplicate removal requires slightly more than 19 MB of memory, or seven times more than the duplicate removal algorithm adapted from hash-division. Clearly, choosing attribute groups X and Y to find attribute groups with relatively few unique values

is crucial for the performance and memory efficiency of this new algorithm. Since such knowledge is not available in most systems and queries (even though some efficient and helpful algorithms exist, e.g., Astrahan et al. [1987]), optimizer heuristics for choosing this algorithm might be difficult to design and verify.

To summarize the discussion on universal quantification algorithms, aggregation can be used in systems that lack direct division algorithms, and hash-division performs universal quantification and relational division generally, i.e., it covers cases with duplicates in the inputs and with referential integrity violations, and efficiently, i.e., it permits partitioning and using hybrid hashing techniques similar to hybrid hash join, making universal quantification (division) as fast as existential quantification (semi-join). As will be discussed later, it can also be effectively parallelized.

7. DUALITY OF SORT- AND HASH-BASED QUERY PROCESSING ALGORITHMS[14]

We conclude the discussion of individual query processing by outlining the many existing similarities and dualities of sort- and hash-based query-processing algorithms as well as the points where the two types of algorithms differ. The purpose is to contribute to a better understanding of the two approaches and their tradeoffs. We try to discuss the approaches in general terms, ignoring whether the algorithms are used for relational join, union, intersection, aggregation, duplicate removal, or other operations. Where appropriate, however, we indicate specific operations.

Table 7 gives an overview of the features that correspond to one another. Both approaches permit in-memory versions for small data sets and disk-based versions for larger data sets. If a data set fits into memory, quicksort is the sort-based method to manage data sets while classic (in-memory) hashing can be used as a hashing technique. It is interesting to note that both quicksort and classic hashing are also used in memory to operate on subsets after "cutting" an entire large data set into pieces. The cutting process is part of the divide-and-conquer paradigm employed for both sort- and hash-based query-processing algorithms. This important similarity of sorting and hashing has been observed before, e.g., by Bratbergsengen [1984] and Salzberg [1988]. There exists, however, an important difference. In the sort-based algorithms, a large data set is divided into subsets using a physical rule, namely into chunks as large as memory. These chunks are later combined using a logical step, merging. In the hash-based algorithms, large inputs are cut into subsets using a logical rule, by hash values. The resulting partitions are later combined using a physical step, i.e., by simply concatenating the subsets or result subsets. In other words, a single-level merge in a sort algorithm is a dual to partitioning in hash algorithms. Figure 17 illustrates this duality and the opposite directions.

This duality can also be observed in the behavior of a disk arm performing the I/O operations for merging or partitioning. While writing initial runs after sorting them with quicksort, the I/O is sequential. During merging, read operations access the many files being merged and require random I/O capabilities. During partitioning, the I/O operations are random, but when reading a partition later on, they are sequential.

For both approaches, sorting and hashing, the amount of available memory limits not only the amount of data in a basic unit processed using quicksort or classic hashing, but also the number of basic units that can be accessed simultaneously. For sorting, it is well known that merging is limited to the quotient of memory size and buffer space required for each run, called the merge fan-in.

[14] Parts of this section have been derived from Graefe et al. [1993], which also provides experimental evidence for the relative performance of sort- and hash-based query processing algorithms and discusses simple cases of transfering tuning ideas from one type of algorithm to the other. The discussion of this section is continued in Graefe [1993a; 1993c].

Table 7. Duality of Sort- and Hash-Based Algorithms

Aspect	Sorting	Hashing
In-memory algorithm	Quicksort	Classic Hash
Divide-and-conquer paradigm	Physical division, logical combination	Logical division, physical combination
Large inputs	Single-level merge	Partitioning
I/O Patterns	Sequential write, random read	Random write, sequential read
Temporary files accessed simultaneously	Fan-in	Fan-out
I/O Optimizations	Read-ahead, forecasting	Write-behind
	Double-buffering, striping merge output	Double-buffering, striping partitioning input
Very large inputs	Multi-level merge	Recursive partitioning
	Merge levels	Recursion depth
	Nonoptimal final fan-in	Nonoptimal hash table size
Optimizations	Merge optimizations	Bucket tuning
Better use of memory	Reverse runs & LRU	Hybrid hashing
	Replacement selection	?
	?	Single input in memory
Aggregation and duplicate removal	Aggregation in replacement selection	Aggregation in hash table
Algorithm phases	Run generation, intermediate and final merge	Initial and intermediate partitioning, in-memory (hybrid) hashing
Resource sharing	Eager merging	Depth-first partitioning
	Lazy merging	Breadth-first partitioning
Partitioning skew and effectiveness	Merging run files of different sizes	Uneven output file sizes
"Item value"	log (run size)	log (build partition size/original build input size)
Bit vector filtering	For both inputs and on each merge level?	For both inputs and on each recursion level
Interesting orderings. multiple joins	Multiple merge-joins without sorting intermediate results	N-ary partitioning and joins
Interesting orderings: grouping/aggregation followed by join	Sorted grouping on foreign key useful for subsequent join	Grouping while building the hash table in hash join
Interesting orderings in index structures	B-trees feeding into a merge-join	Merging in hash value order

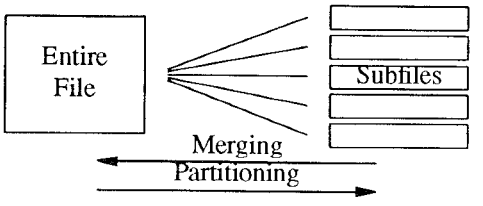


Figure 17. Duality of partitioning and merging.

Similarly, partitioning is limited to the same fraction, called the fan-out since the limitation is encountered while writing partition files.

In order to keep the merge process active at all times, many merge implementations use read-ahead controlled by forecasting, trading reduced I/O delays for a reduced fan-in. In the ideal case, the bandwidths of I/O and processing (merging) match, and I/O latencies for both the merge input and output are hidden by read-ahead and double-buffering, as mentioned earlier in the section on sorting. The dual to read-ahead during merging is write-behind during partitioning, i.e., keeping a free output buffer that can be allocated to an output file while the previous page for that file is being written to disk. There is no dual to forecasting because it is trivial that the next

output partition to write to is the one for which an output cluster has just filled up. Both read-ahead in merging and write-behind in partitioning are used to ensure that the processor never has to wait for the completion of an I/O operation. Another dual is double-buffering and striping over multiple disks for the output of sorting and the input of partitioning.

Considering the limitation on fan-in and fan-out, additional techniques must be used for very large inputs. Merging can be performed in multiple levels, each combining multiple runs into larger ones. Similarly, partitioning can be repeated recursively, i.e., partition files are repartitioned, the results repartitioned, etc., until the partition files fit into main memory. In sorting and merging, the runs grow in each level by a factor equal to the fan-in. In partitioning, the partition files decrease in size by a factor equal to the fan-out in each recursion level. Thus, the number of levels during merging is equal to the recursion depth during partitioning. There are two exceptions to be made regarding hash value distributions and relative sizes of inputs in binary operations such as join; we ignore those for now and will come back to them later.

If merging is done in the most naive way, i.e., merging all runs of a level as soon as their number reaches the fan-in, the last merge on each level might not be optimal. Similarly, if the highest possible fan-out is used in each partitioning step, the partition files in the deepest recursion level might be smaller than memory, and less than the entire memory is used when processing these files. Thus, in both approaches the memory resources are not used optimally in the most naive versions of the algorithms.

In order to make best use of the final merge (which, by definition, includes all output items and is therefore the most expensive merge), it should proceed with the maximal possible fan-in. Making best use of the final merge can be ensured by merging fewer runs than the maximal fan-in after the end of the input file has been reached (as discussed in the earlier section on sorting). There is no direct dual in hash-based algorithms for this optimization. With respect to memory utilization, the fact that a partition file and therefore a hash table might actually be smaller than memory is the closest to a dual. Utilizing memory more effectively and using less than the maximal fan-out in hashing has been addressed in research on bucket tuning [Kitsuregawa et al. 1989a] and on histogram-driven recursive hybrid hash join [Graefe 1993a].

The development of hybrid hash algorithms [DeWitt et al. 1984; Shapiro 1986] was a consequence of the advent of large main memories that had led to the consideration of hash-based join algorithms in the first place. If the data set is only slightly larger than the available memory, e.g., 10% larger or twice as large, much of the input can remain in memory and is never written to a disk-resident partition file. To obtain the same effect for sort-based algorithms, if the database system's buffer manager is sufficiently smart or receives and accepts appropriate hints, it is possible to retain some or all of the pages of the last run written in memory and thus achieve the same effect of saving I/O operations. This effect can be used particularly easily if the initial runs are written in reverse (descending) order and scanned backward for merging. However, if one does not believe in buffer hints or prefers to absolutely ensure these I/O savings, then using a final memory-resident run explicitly in the sort algorithm and merging it with the disk-resident runs can guarantee this effect.

Another well-known technique to use memory more effectively and to improve sort performance is to generate runs twice as large as main memory using a priority heap for replacement selection [Knuth 1973], as discussed in the earlier section on sorting. If the runs' sizes are doubled, their number is cut in half. Therefore, merging can be reduced by some amount, namely $\log_F(2) = 1/\log_2(F)$ merge levels. This optimization for sorting has no direct dual in the

realm of hash-based query-processing algorithms.

If two sort operations produce input data for a binary operator such as a merge-join and if both sort operators' final merges are interleaved with the join, each final merge can employ only half the memory. In hash-based one-to-one match algorithms, only one of the two inputs resides in and consumes memory beyond a single input buffer, not both as in two final merges interleaved with a merge-join. This difference in the use of the two inputs is a distinct advantage of hash-based one-to-one match algorithms that does not have a dual in sort-based algorithms.

Interestingly, these two differences of sort- and hash-based one-to-one match algorithms cancel each other out. Cutting the number of runs in half (on each merge level, including the last one) by using replacement selection for run generation exactly offsets this disadvantage of sort-based one-to-one match operations.

Run generation using replacement selection has a second advantage over quicksort; this advantage has a direct dual in hashing. If a hash table is used to compute an aggregate function using grouping, e.g., sum of salaries by department, hash table overflow occurs only if the operation's output does not fit in memory. Consider, for example, the sum of salaries by department for 100,000 employees in 1,000 departments. If the 1,000 result records fit in memory, classic hashing (without overflow) is sufficient. On the other hand, if sorting based on quicksort is used to compute this aggregate function, the input must fit into memory to avoid temporary files.[15] If replacement selection is used for run generation, however, the same behavior as with classic hashing is easy to achieve.

[15] A scheme using quicksort and avoiding temporary I/O in this case can be devised but would be extremely cumbersome; we do not know of any report or system with such a scheme.

If an iterator interface is used for both its input and output, and therefore multiple operators overlap in time, a sort operator can be divided into three distinct algorithm phases. First, input items are consumed and sorted into initial runs. Second, intermediate merging reduces the number of runs such that only one final merge step is left. Third, the final merge is performed on demand from the consumer of the sorted data stream. During the first phase, the sort iterator has to share resources, most notably memory and disk bandwidth, with its producer operators in a query evaluation plan. Similarly, the third phase must share resources with the consumers.

In many sort implementations, namely those using eager merging, the first and second phase interleave as a merge step is initiated whenever the number of runs on one level becomes equal to the fan-in. Thus, some intermediate merge steps cannot use all resources. In lazy merging, which starts intermediate merges only after all initial runs have been created, the intermediate merges do not share resources with other operators and can use the entire memory allocated to a query evaluation plan; thus, intermediate merges can be more effective in lazy merging than in eager merging.

Hash-based query processing algorithms exhibit three similar phases. First, the first partitioning step executes concurrently with the input operator or operators. Second, intermediate partitioning steps divide the partition files to ensure that they can be processed with hybrid hashing. Third, hybrid and in-memory hash methods process these partition files and produce output passed to the consumer operators. As in sorting, the first and third phases must share resources with other concurrent operations in the same query evaluation plan.

The standard implementation of hash-based query processing algorithms for very large inputs uses recursion, i.e., the original algorithm is invoked for each partition file (or pair of partition files). While conceptually simple, this method has the disadvantage that output is

produced before all intermediate partitioning steps are complete. Thus, the operators that consume the output must allocate resources to receive this output, typically memory (e.g., a hash table). Further intermediate partitioning steps will have to share resources with the consumer operators, making them less effective. We call this direct recursive implementation of hash-based partitioning *depth-first* partitioning and consider its behavior as well as its resource sharing and performance effects a dual to eager merging in sorting. The alternative schedule is *breadth-first* partitioning, which completes each level of partitioning before starting the next one. Thus, hybrid and in-memory hashing are not initiated until all partition files have become small enough to permit hybrid and in-memory hashing, and intermediate partitioning steps never have to share resources with consumer operators. Breadth-first partitioning is a dual to lazy merging, and it is not surprising that they are both equally more effective than depth-first partitioning and eager merging, respectively.

It is well known that partitioning skew reduces the effectiveness of hash-based algorithms. Thus, the situation shown in Figure 18 is undesirable. In the extreme case, one of the partition files is as large as the input, and an entire partitioning step has been wasted. It is less well recognized that the same issue also pertains to sort-based query processing algorithms [Graefe 1993c]. Unfortunately, in order to reduce the number of merge steps, it is often necessary to merge files from different merge levels and therefore of different sizes. In other words, the goals of optimized merging and of maximal merge effectiveness do not always match, and very sophisticated merge plans, e.g., polyphase merging, might be required [Knuth 1973].

The same effect can also be observed if "values" are attached to items in runs and in partition files. Values should reflect the work already performed on an item. Thus, the value should increase with run sizes in sorting while the value

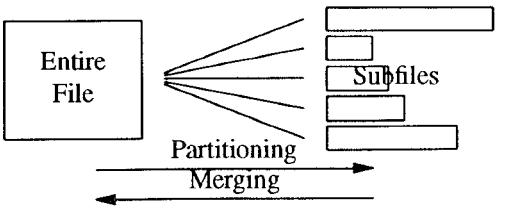


Figure 18. Partitioning skew.

must increase as partition files get smaller in hash-based query processing algorithms. For sorting, a suitable choice for such a value is the logarithm of the run size [Graefe 1993c]. The value of a sorted run is the product of the run's size and the size's logarithm. The optimal merge effectiveness is achieved if each item's value increases in each merge step by the logarithm of the fan-in, and the overall value of all items increases with this logarithm multiplied with the data volume participating in the merge step. However, only if all runs in a merge step are of the same size will the value of all items increase with the logarithm of the fan-in.

In hash-based query processing, the corresponding value is the fraction of a partition size relative to the original input size [Graefe 1993c]. Since only the build input determines the number of recursion levels in binary hash partitioning, we consider only the build partition. If the partitioning is skewed, i.e., output partition files are not of uniform length, the overall effectiveness of the partitioning step is not optimal, i.e., equal to the logarithm of the partitioning fan-out. Thus, preventing or managing skew in partitioning hash functions is very important [Graefe 1993a].

Bit vector filtering, which will be discussed later in more detail, can be used for both sort- and hash-based one-to-one match operations, although it has been used mainly for parallel joins to date. Basically, a bit vector filter is a large array of bits initialized by hashing items in the first input of a one-to-one match operator and used to detect items in the second input that cannot possibly have a

match in the first input. In effect, bit vector filtering reduces the second input to the items that truly participate in the binary operation plus some "false passes" due to hash collisions in the bit vector filter. In a merge-join with two sort operations, if the bit vector filter is used before the second sort, bit vector filtering is as effective as in hybrid hash join in reducing the cost of processing the second input. In merge-join, it can also be used symmetrically as shown in Figure 19. Notice that for the right input, bit vector filtering reduces the sort input size, whereas for the left input, it only reduces the merge-join input. In recursive hybrid hash join, bit vector filtering can be used in each recursion level. The effectiveness of bit vector filtering increases in deeper recursion levels, because the number of distinct data values in each partition file decreases, thus reducing the number of hash collisions and false passes if bit vector filters of the same size are used in each recursion level. Moreover, it can be used in both directions, i.e., to reduce the second input using a bit vector filter based on the first input and to reduce the first input (in the next recursion level) using a bit vector filter based on the second input. The same effect could be achieved for sort-based binary operations requiring multilevel sorting and merging, although to do so implies switching back and forth between the two sorts for the two inputs after each merge level. Not surprisingly, switching back and forth after each merge level would be the dual to the partitioning process of both inputs in recursive hybrid hash join. However, sort operators that switch back and forth on each merge level are not only complex to implement but may also inhibit the merge optimizations discussed earlier.

The final entries in Table 7 concern *interesting orderings* used in the System R query optimizer [Selinger et al. 1979] and presumably other query optimizers as well. A strong argument in favor of sorting and merge-join is the fact that merge-join delivers its output in sorted order; thus, multiple merge-joins on the same attribute can be performed without sorting intermediate join results. For joining three relations, as shown in Figure 20, pipelining data from one merge-join to the next without sorting translates into a 3:4 advantage in the number of sort operations compared to two joins on different join keys, because the intermediate result O_1 does not need to be sorted. For joining N relations on the same key, only N sorts are required instead of $2 \times N - 2$ for joins on different attributes. Since set operations such as the union or intersection of N sets can always be performed using a merge-join algorithm without sorting intermediate results, the effect of interesting orderings is even more important for set operations than for relational joins.

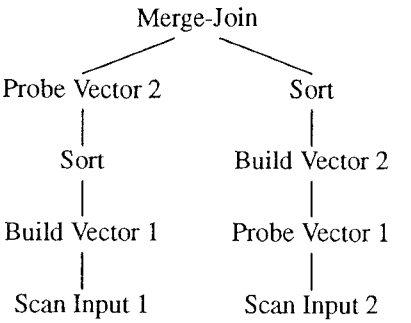


Figure 19. Merge-join with symmetric bit vector filtering.

Hash-based algorithms tend to produce their outputs in a very unpredictable order, depending on the hash function and on overflow management. In order to take advantage of multiple joins on the same attribute (or of multiple intersections, etc.) similar to the advantage derived from interesting orderings in sort-based query processing, the equality of attributes has to be exploited during the logical step of hashing, i.e., during partitioning. In other words, such set operations and join queries can be executed effectively by a hash join algorithm that recursively partitions N inputs concurrently. The recursion terminates when $N - 1$ inputs fit into memory and when the Nth input is used to probe $N - 1$

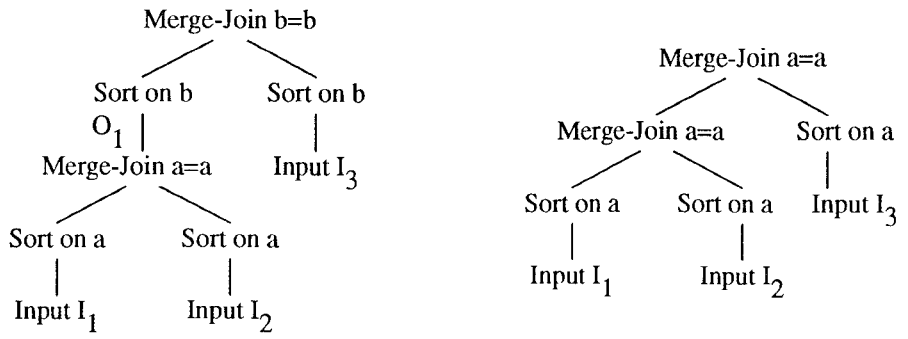


Figure 20. The effect of interesting orderings.

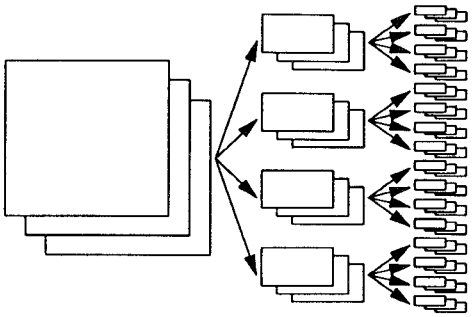


Figure 21. Partitioning in a multiinput hash join.

hash tables. Thus, the basic operation of this N-ary join (intersection, etc.) is an N-ary join of an N-tuple of partition files, not pairs as in binary hash join with one build and one probe file for each partition. Figure 21 illustrates recursive partitioning for a join of three inputs. Instead of partitioning and joining a pair of inputs and pairs of partition files as in traditional binary hybrid hash join, there are file triples (or N-tuples) at each step.

However, N-ary recursive partitioning is cumbersome to implement, in particular if some of the "join" operations are actually semi-join, outer join, set intersection, union, or difference. Therefore, until a clean implementation method for hash-based N-ary matching has been found, it might well be that this distinction, joins on the same or on different attributes, contributes to the right choice between sort- and hash-based algorithms for complex queries.

Another situation with interesting orderings is an aggregation followed by a join. Many aggregations condense information about individual entities; thus, the aggregation operation is performed on a relation representing the "many" side of a many-to-one relationship or on the relation that represents relationship instances of a many-to-many relationship. For example, students' grade point averages are computed by grouping and averaging transcript entries in a many-to-many relationship called *transcript* between *students* and *courses*. The important point to note here and in many similar situations is the grouping attribute is a foreign key. In order to relate the aggregation output with other information pertaining to the entities about which information was condensed, aggregations are frequently followed by a join. If the grouping operation is based on sorting (on the grouping attribute, which very frequently is a foreign key), the natural sort order of the aggregation output can be exploited for an efficient merge-join without sorting.

While this seems to be an advantage of sort-based aggregation and join, this combination of operations also permits a special trick in hash-based query processing [Graefe 1993b]. Hash-based aggregation is based on identifying items of the same group while building the hash table. At the end of the operation, the hash table contains all output items hashed on the grouping attribute. If the grouping attribute is the join attribute in the next operation, this hash table can immediately be probed with the other

join input. Thus, the combined aggregation-join operation uses only one hash table, not two hash tables as two separate operations would do. The differences to two separate operations are that only one join input can be aggregated efficiently and that the aggregated input must be the join's build input. Both issues could be addressed by symmetric hash joins with a hash table on each of the inputs which would be as efficient as sorting and grouping both join inputs.

A third use of interesting orderings is the positive interaction of (sorted, B-tree) index scans and merge-join. While it has not been reported explicitly in the literature, the leaves and entries of two hash indices can be merge-joined just like those of two B-trees, provided the same hash function was used to create the indices. For example, it is easy to imagine "merging" the leaves (data pages) of two extendible hash indices [Fagin et al. 1979], even if the key cardinalities and distributions are very different.

In summary, there exist many dualities between sorting using multilevel merging and recursive hash table overflow management. Two special cases exist which favor one or the other, however. First, if two join inputs are of different size (and the query optimizer can reliably predict this difference), hybrid hash join outperforms merge-join because only the smaller of the two inputs determines what fraction of the input files has to be written to temporary disk files during partitioning (or how often each record has to be written to disk during recursive partitioning), while each file determines its own disk I/O in sorting [Bratbergsengen 1984]. For example, sorting the larger of two join inputs using multiple merge levels is more expensive than writing a small fraction of that file to hash overflow files. This performance advantage of hashing grows with the relative size difference of the two inputs, not with their absolute sizes or with the memory size.

Second, if the hash function is very poor, e.g., because of a prior selection on the join attribute or a correlated attribute, hash partitioning can perform very poorly and create significantly higher costs than sorting and merge-join. If the quality of the hash function cannot be predicted or improved (tuned) dynamically [Graefe 1993a], sort-based query-processing algorithms are superior because they are less vulnerable to nonuniform data distributions. Since both cases, join of differently sized files and skewed hash value distributions, are realistic situations in database query processing, we recommend that both sort- and hash-based algorithms be included in a query-processing engine and chosen by the query optimizer according to the two cases above. If both cases arise simultaneously, i.e., a join of differently sized inputs with unpredictable hash value distribution, the query optimizer has to estimate which one poses the greater danger to system performance and choose accordingly.

The important conclusion from these dualities is that neither the absolute input sizes nor the absolute memory size nor the input sizes relative to the memory size determine the choice between sort- and hash-based query-processing algorithms. Instead, the choice should be governed by the sizes of the two inputs into binary operators relative to each other and by the danger of performance impairments due to skewed data or hash value distributions. Furthermore, because neither algorithm type outperforms the other in all situations, both should be available in a query execution engine for a choice to be made in each case by the query optimizer.

8. EXECUTION OF COMPLEX QUERY PLANS

When multiple operators such as aggregations and joins execute concurrently in a pipelined execution engine, physical resources such as memory and disk bandwidth must be shared by all operators. Thus, optimal scheduling of multiple operators and the division and allocation of resources in a complex plan are important issues.

In earlier relational execution engines, these issues were largely ignored for two

reasons. First, only left-deep trees were used for query execution, i.e., the right (inner) input of a binary operator had to be a scan. In other words, concurrent execution of multiple subplans in a single query was not possible. Second, under the assumption that sorting was needed at each step and considering that sorting for nontrivial file sizes requires that the entire input be written to temporary files at least once, concurrency and the need for resource allocation were basically absent. Today's query execution engines consider more join algorithms that permit extensive pipelining, e.g., hybrid hash join, and more complex query plans, including bushy trees. Moreover, today's systems support more concurrent users and use parallel-processing capabilities. Thus, resource allocation for complex queries is of increasing importance for database query processing.

Some researchers have considered resource contention among multiple query processing operators with the focus on buffer management. The goal in these efforts was to assign disk pages to buffer slots such that the benefit of each buffer slot would be maximized, i.e., the number of I/O operations avoided in the future. Sacco and Schkolnick [1982; 1986] analyzed several database algorithms and found that their cost functions exhibit steps when plotted over available buffer space, and they suggested that buffer space should be allocated at the low end of a step for the least buffer use at a given cost. Chou [1985] and Chou and DeWitt [1985] took this idea further by combining it with separate page replacement algorithms for each relation or scan, following observations by Stonebraker [1981] on operating system support for database systems, and with load control, calling the resulting algorithm DBMIN. Faloutsos et al. [1991] and Ng et al. [1991] generalized this goal and used the classic economic concepts of decreasing marginal gain and balanced marginal gains for maximal overall gain. Their measure of gain was the reduction in the number of page faults. Zeller and Gray [1990] designed a hash join algorithm that adapts to the current memory and buffer contention each time a new hash table is built. Most recently, Brown et al. [1992] have considered resource allocation tradeoffs among short transactions and complex queries.

Schneider [1990] and Schneider and DeWitt [1990] were the first to systematically examine execution schedules and costs for right-deep trees, i.e., query evaluation plans with multiple binary hash joins for which all build phases proceed concurrently or at least could proceed concurrently (notice that in a left-deep plan, each build phase receives its data from the probe phase of the previous join, limiting left-deep plans to two concurrent joins in different phases). Among the most interesting findings are that through effective use of bit vector filtering (discussed later), memory requirements for right-deep plans might actually be comparable to those of left-deep plans [Schneider 1991]. This work has recently been extended by Chen et al. [1992] to bushy plans interpreted and executed as multiple right-deep subplans.

For binary matching iterators to be used in bushy plans, we have identified several concerns. First, some query-processing algorithms include a point at which all data are in temporary files on disk and at which no intermediate result data reside in memory. Such "stop" points can be used to switch efficiently between different subplans. For example, if two subplans produce and sort two merge-join inputs, stopping work on the first subplan and switching to the second one should be done when the first sort operator has all its data in sorted runs and when only the final merge is left but no output has been produced yet. Figure 22 illustrates this point in time. Fortunately, this timing can be realized naturally in the iterator implementation of sorting if input runs for the final merge are opened in the first call of the *next* procedure, not at the end of the *open* phase. A similar stop point is available in hash join when using overflow avoidance.

Second, since hybrid hashing produces some output data before the memory contents (output buffers and hash table) can

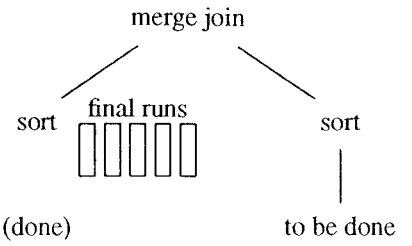


Figure 22. The stop point during sorting.

be discarded and since, therefore, such a stop point does not occur in hybrid hash join, implementations of hybrid hash join and other binary match operations should be parameterized to permit overflow avoidance as a run time option to be chosen by the query optimizer. This dynamic choice will permit the query optimizer to force a stop point in some operators while using hybrid hash in most operations.

Third, binary-operator implementations should include a switch that controls which subplan is initiated first. In Table 1 with algorithm outlines for iterators' *open*, *next*, and *close* procedures, the hash join *open* procedure executes the entire build-input plan first before opening the probe input. However, there might be situations in which it would be better to *open* the probe input before executing the build input. If the probe input does not hold any resources such as memory between *open* and *next* calls, initiating the probe input first is not a problem. However, there are situations in which it creates a big benefit, in particular in bushy query evaluation plans and in parallel systems to be discussed later.

Fourth, if multiple operators are active concurrently, memory has to be divided among them. If two sorts produce input data for a merge-join, which in turn passes its output into another sort using quicksort, memory should be divided proportionally to the sizes of the three files involved. We believe that for multiple sorts producing data for multiple merge-joins on the same attribute, proportional memory division will also work best. If a sort in its run generation phase shares resources with other operations, e.g., a sort following two sorts in their final merges and a merge-join, it should also use resources proportional to its input size. For example, if two merge-join inputs are of the same size and if the merge-join output which is sorted immediately following the merge-join is as large as the two inputs together, the two final merges should each use one quarter of memory while the run generation (quicksort) should use one half of memory.

Fifth, in recursive hybrid hash join, the recursion levels should be executed level by level. In the most straightforward recursive algorithm, recursive invocation of the original algorithm for each output partition results in depth-first partitioning, and the algorithm produces output as soon as the first leaf in the recursion tree is reached. However, if the operator that consumes the output requires memory as soon as it receives input, for example, hybrid hash join (ii) in Figure 23 as soon as hybrid hash join (i) produces output, the remaining partitioning operations in the producer operator (hybrid hash join (i)) must share memory with the consumer operator (hybrid hash join (ii)), effectively cutting the partitioning fan-out in the producer in half. Thus, hash-based recursive matching algorithms should proceed in three distinct phases—consuming input and initial partitioning, partitioning into files suitable for hybrid hash join, and final hybrid hash join for all partitions—with phase two completed entirely before phase three commences. This sequence of partitioning steps was introduced as breadth-first partitioning in the previous section as opposed to depth-first partitioning used in the most straightforward recursive algorithms. Of course, the topmost operator in a query evaluation plan does not have a consumer operator with which it shares resources; therefore, this operator should use depth-first partitioning in order to provide a better response time, i.e., earlier delivery of the first data item.

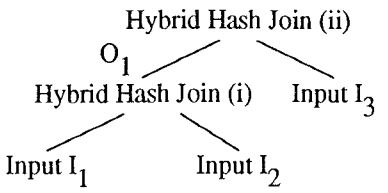


Figure 23. Plan for joining three inputs.

Sixth, the allocation of resources other than memory, e.g., disk bandwidth and disk arms for seeking in partitioning and merging, is an open issue that should be addressed soon, because the different improvement rates in CPU and disk speeds will increase the importance of disk performance for overall query processing performance. One possible alleviation of this problem might come from disk arrays configured exclusively for performance, not for reliability. Disk arrays might not deliver the entire performance gain the large number of disk drives could provide if it is not possible to disable a disk array's parity mechanisms and to access specific disks within an array, particularly during partitioning and merging.

Finally, scheduling bushy trees in multiprocessor systems is not entirely understood yet. While all considerations discussed above apply in principle, multiprocessors permit truly concurrent execution of multiple subplans in a bushy tree. However, it is a very hard problem to schedule two or more subplans such that their result streams are available at the right times and at the right rates, in particular in light of the unavoidable errors in selectivity and cost estimation during query optimization [Christodoulakis 1984; Ioannidis and Christodoulakis 1991].

The last point, estimation errors, leads us to suspect that plans with 30 (or even 100) joins or other operations cannot be optimized completely before execution. Thus, we suspect that a technique reminiscent of Ingres Decomposition [Wong and Youssefi 1976; Youssefi and Wong 1979] will prove to be more effective. One of the principal ideas of Ingres Decomposition is a repetitive cycle consisting of three steps. First, the next step is selected, e.g., a selection or join. Second, the chosen step is executed into a temporary table. Third, the query is simplified by removing predicates evaluated in the completed execution step and replacing one range variable (relation) in the query with the new temporary table. The justification and advantage of this approach are that all earlier selectivities are known for each decision, because the intermediate results are materialized. The disadvantage is that data flow between operators cannot be exploited, resulting in a significant cost for writing and reading intermediate files. For very complex queries, we suggest modifying Decomposition to decide on and execute multiple steps in each cycle, e.g., 3 to 9 joins, instead of executing only one selection or join as in Ingres. Such a hybrid approach might very well combine the advantages of a priori optimization, namely, in-memory data flow between iterators, and optimization with exactly known intermediate result sizes.

An optimization and execution environment even further tuned for very complex queries would anticipate possible outcomes of executing subplans and provide multiple alternative subsequent plans. Figure 24 shows the structure of such a dynamic plan for a complex query. First, subplan A is executed, and statistics about its result are gathered while it is saved on disk. Depending on these statistics, either B or C is executed next. If B is chosen and executed, one of D, E, and F will complete the query; in the case of C instead of B, it will be G or H. Notice that each letter A–H can be an arbitrarily complex subplan, although probably not more than 10 operations due to the limitations of current selectivity estimation methods. Unfortunately, realization of such sophisticated query optimizers will require further research, e.g., into determination of when separate cases are warranted and limitation of the possibly exponential growth in the number of subplans.

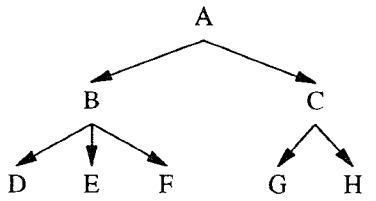


Figure 24. A decision tree of partial plans.

9. MECHANISMS FOR PARALLEL QUERY EXECUTION

Considering that all high-performance computers today employ some form of parallelism in their processing hardware, it seems obvious that software written to manage large data volumes ought to be able to exploit parallel execution capabilities [DeWitt and Gray 1992]. In fact, we believe that five years from now it will be argued that a database management system without parallel query execution will be as handicapped in the market place as one without indices.

The goal of parallel algorithms and systems is to obtain speedup and scaleup, and speedup results are frequently used to demonstrate the accomplishments of a design and its implementation. Speedup considers additional hardware resources for a constant problem size; linear speedup is considered optimal. In other words, N times as many resources should solve a constant-size problem in $1/N$ of the time. Speedup can also be expressed as parallel efficiency, i.e., a measure of how close a system comes to linear speedup. For example, if solving a problem takes 1,200 seconds on a single machine and 100 seconds on 16 machines, the speedup is somewhat less than linear. The parallel efficiency is $(1 \times 1200)/(16 \times 100) = 75\%$.

An alternative measure for a parallel system's design and implementation is scaleup, in which the problem size is altered with the resources. Linear scaleup is achieved when N times as many resources can solve a problem with N times as much data in the same amount of time. Scaleup can also be expressed using parallel efficiency, but since speedup and scaleup are different, it should always be clearly indicated which parallel efficiency measure is being reported.

A third measure for the success of a parallel algorithm based on Amdahl's law is the fraction of the sequential program for which linear speedup was attained, defined by $p = f \times s/d + (1 - f) \times s$ for sequential execution time s, parallel execution time p, and degree of parallelism d. Resolved for f, this is $f = (s - p)/(s - s/d) = ((s - p)/s)/((d - 1)/d)$. For the example above, this fraction is $f = ((1200 - 100)/1200)/((16 - 1)/16) = 97.78\%$. Notice that this measure gives much higher percentage values than the parallel efficiency calculated earlier; therefore, the two measures should not be confused.

For query processing problems involving sorting or hashing in which multiple merge or partitioning levels are expected, the speedup can frequently be more than linear, or superlinear. Consider a sorting problem that requires two merge levels in a single machine. If multiple machines are used, the sort problem can be partitioned such that each machine sorts a fraction of the entire data amount. Such partitioning will, in a good implementation, result in linear speedup. If, in addition, each machine has its own memory such that the total memory in the system grows with the size of the machine, fewer than two merge levels will suffice, making the speedup superlinear.

9.1 Parallel versus Distributed Database Systems

It might be useful to start the discussion of parallel and distributed query processing with a distinction of the two concepts. In the database literature, "distributed" usually implies "locally autonomous," i.e., each participating system is a complete database management system in itself, with access control, metadata (catalogs), query processing, etc. In other words, each node in a distributed database management system can function entirely on its own, whether or not the other nodes are present or accessible. Each node per-

forms its own access control, and cooperation of each node in a distributed transaction is voluntary. Examples of distributed (research) systems are R* [Haas et al. 1982; Traiger et al. 1982], distributed Ingres [Epstein and Stonebraker 1980; Stonebraker 1986a], and SDD-1 [Bernstein et al. 1981; Rothnie et al. 1980]. There are now several commercial distributed relational database management systems. Ozsu and Valduriez [1991a; 1991b] have discussed distributed database systems in much more detail. If the cooperation among multiple database systems is only limited, the system can be called a "federated" database system [Sheth and Larson 1990].

In parallel systems, on the other hand, there is only one locus of control. In other words, there is only one database management system that divides individual queries into fragments and executes the fragments in parallel. Access control to data is independent of where data objects currently reside in the system. The query optimizer and the query execution engine typically assume that all nodes in the system are available to participate in efficient execution of complex queries, and participation of nodes in a given transaction is either presumed or controlled by a global resource manager, but is not based on voluntary cooperation as in distributed systems. There are several parallel research prototypes, e.g., Gamma [DeWitt et al. 1986; DeWitt et al. 1990], Bubba [Boral 1988; Boral et al. 1990], Grace [Fushimi et al. 1986; Kitsuregawa et al. 1983], and Volcano [Graefe 1990b; 1993b; Graefe and Davison 1993], and products, e.g., Tandem's NonStop SQL [Englert et al. 1989; Zeller 1990], Teradata's DBC/1012 [Neches 1984; 1988; Teradata 1983], and Informix [Davison 1992].

Both distributed database systems and parallel database systems have been designed in various kinds, which may create some confusion. Distributed systems can be either homogeneous, meaning that all participating database management systems are of the same type (the hardware and the operating system may even be of the same types), or heterogeneous, meaning that multiple database management systems work together using standardized interfaces but are internally different.[16] Furthermore, distributed systems may employ parallelism, e.g., by pipelining datasets between nodes with the receiver already working on some items while the producer is still sending more. Parallel systems can be based on shared-memory (also called shared-everything), shared-disk (multiple processors sharing disks but not memory), distributed-memory (without sharing disks, also called shared-nothing), or hierarchical computer architectures consisting of multiple clusters, each with multiple CPUs and disks and a large shared memory. Stonebraker [1986b] compared the first three alternatives using several aspects of database management, and came to the conclusion that distributed memory is the most promising database management system platform. Each of these approaches has advantages and disadvantages; our belief is that the hierarchical architecture is the most general of these architectures and should be the target architecture for new database software development [Graefe and Davison 1993].

9.2 Forms of Parallelism

There are several forms of parallelism that are interesting to designers and implementors of query processing systems. *Interquery* parallelism is a direct result of the fact that most database management systems can service multiple requests concurrently. In other words, multiple queries (transactions) can be executing concurrently within a single database management system. In this form of parallelism, resource contention

[16] In some organizations, two different database management systems may run on the same (fairly large) computer. Their interactions could be called "nondistributed heterogeneous." However, since the rules governing such interactions are the same as for distributed heterogeneous systems, the case is usually ignored in research and system design.

is of great concern, in particular, contention for memory and disk arms.

The other forms of parallelism are all based on the use of algebraic operations on sets for database query processing, e.g., selection, join, and intersection. The theory and practice of exploiting other "bulk" types such as lists for parallel database query execution are only now developing. *Interoperator* parallelism is basically pipelining, or parallel execution of different operators in a single query. For example, the iterator concept discussed earlier has also been called "synchronous pipelines" [Pirahesh et al. 1990]; there is no reason not to consider asynchronous pipelines in which operators work independently connected by a buffering mechanism to provide flow control.

Interoperator parallelism can be used in two forms, either to execute producers and consumers in pipelines, called *vertical interoperator* parallelism here, or to execute independent subtrees in a complex bushy-query evaluation plan concurrently, called *horizontal interoperator* or *bushy* parallelism here. A simple example for bushy parallelism is a merge-join receiving its input data from two sort processes. The main problem with bushy parallelism is that it is hard or impossible to ensure that the two subplans start generating data at the right time and generate them at the right rates. Note that the right time does not necessarily mean the same time, e.g., for the two inputs of a hash join, and that the right rates are not necessarily equal, e.g., if two inputs of a merge-join have different sizes. Therefore, bushy parallelism presents too many open research issues and is hardly used in practice at this time.

The final form of parallelism in database query processing is *intraoperator* parallelism in which a single operator in a query plan is executed in multiple processes, typically on disjoint pieces of the problem and disjoint subsets of the data. This form, also called parallelism based on *fragmentation* or *partitioning*, is enabled by the fact that query processing focuses on sets. If the underlying data represented sequences or time series in a scientific database management system, partitioning into subsets to be operated on independently would not be feasible or would require additional synchronization when putting the independently obtained results together.

Both vertical interoperator parallelism and intraoperator parallelism are used in database query processing to obtain higher performance. Beyond the obvious opportunities for speedup and scaleup that these two concepts offer, they both have significant problems. Pipelining does not easily lend itself to load balancing because each process or processor in the pipeline is loaded proportionally to the amount of data it has to process. This amount cannot be chosen by the implementor or the query optimizer and cannot be predicted very well. For intraoperator, partitioning-based parallelism, load balance and performance are optimal if the partitions are all of equal size; however, this can be hard to achieve if value distributions in the inputs are skewed.

9.3 Implementation Strategies

The purpose of the query execution engine is to provide mechanisms for query execution from which the query optimizer can choose—the same applies for the means and mechanisms for parallel execution. There are two general approaches to parallelizing a query execution engine, which we call the *bracket* and *operator models* and which are used, for example, in the Gamma and Volcano systems, respectively.

In the bracket model, there is a generic process template that can receive and send data and can execute exactly one operator at any point of time. A schematic diagram of a template process is shown in Figure 25, together with two possible operators, join and aggregation. In order to execute a specific operator, e.g., a join, the code that makes up the generic template "loads" the operator into its place (by switching to this operator's code) and initiates the operator which then controls execution; network I/O on the receiving

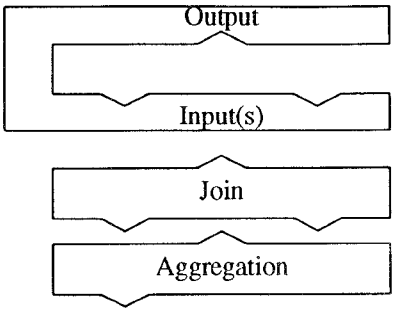


Figure 25. Bracket model of parallelization.

and sending sides is performed as a service to the operator on its request and initiation and is implemented as procedures to be called by the operator. The number of inputs that can be active at any point of time is limited to two since there are only unary and binary operators in most database systems. The operator is surrounded by generic template code, which shields it from its environment, for example, the operator(s) that produce its input and consume its output. For parallel query execution, many templates are executed concurrently in the system, using one process per template. Because each operator is written with the implicit assumption that this operator controls all activities in its process, it is not possible to execute two operators in one process without resorting to some thread or coroutine facility i.e., a second implementation level of the process concept.

In a query-processing system using the bracket model, operators are coded in such a way that network I/O is their only means of obtaining input and delivering output (with the exception of scan and store operators). The reason is that each operator is its own locus of control, and network flow control must be used to coordinate multiple operators, e.g., to match two operators' speed in a producer-consumer relationship. Unfortunately, this coordination requirement also implies that passing a data item from one operator to another always involves expensive interprocess communication system calls, even in the cases when an entire query is evaluated on a single CPU (and could therefore be evaluated in a single process, without interprocess communication and operating system involvement) or when data do not need to be repartitioned among nodes in a network. An example for the latter is the query "joinCselAselB" in the Wisconsin Benchmark, which requires joining three inputs on the same attribute [DeWitt 1991], or any other query that permits interesting orderings [Selinger et al. 1979], i.e., any query that uses the same join attribute for multiple binary joins. Thus, in queries with multiple operators (meaning almost all queries), interprocess communication and its overhead are mandatory in the bracket model rather than optional.

An alternative to the bracket model is the operator model. Figure 26 shows a possible parallelization of a join plan using the operator model, i.e., by inserting "parallelism" operators into a sequential plan, called *exchange* operators in the Volcano system [Graefe 1990b; Graefe and Davison 1993]. The exchange operator is an iterator like all other operators in the system with *open*, *next*, and *close* procedures; therefore, the other operators are entirely unaffected by the presence of exchange operators in a query evaluation plan. The exchange operator does not contribute to data manipulation; thus, on the logical level, it is a "no-op" that has no place in a logical query algebra such as the relational algebra. On the physical level of algorithms and processes, however, it provides control not provided by any of the normal operators, i.e., process management, data redistribution, and flow control. Therefore, it is a *control operator* or a *meta-operator*. Separation of data manipulation from process control and interprocess communication can be considered an important advantage of the operator model of parallel query processing, because it permits design, implementation, and execution of new data manipulation algorithms such as N-ary hybrid hash join [Graefe 1993a] without regard to the execution environment.

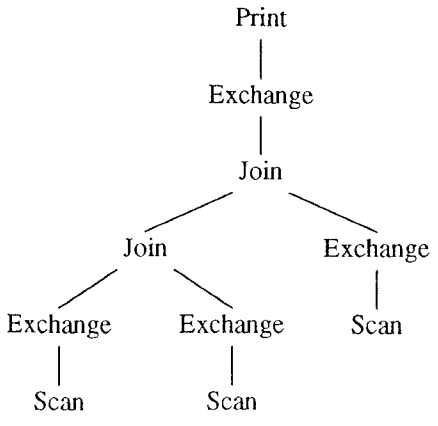


Figure 26. Operator model of parallelization.

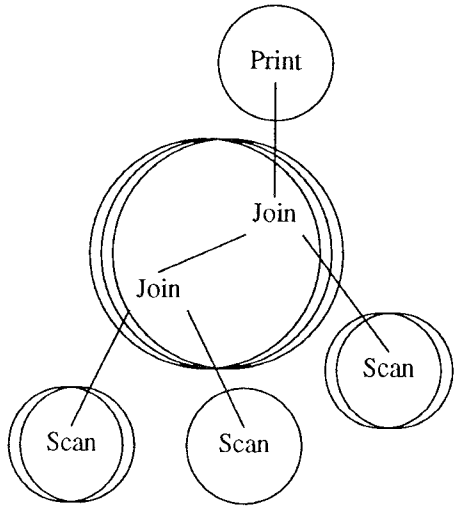


Figure 27. Processes created by exchange operators.

A second issue important to point out is that the exchange operator only provides mechanisms for parallel query processing; it does not determine or presuppose policies for using its mechanisms. Policies for parallel processing such as the degree of parallelism, partitioning functions, and allocation of processes to processors can be set either by a query optimizer or by a human experimenter in the Volcano system as they are still subject to intense research. The design of the exchange operator permits execution of a complex query in a single process (by using a query plan without any exchange operators, which is useful in single-processor environments) or with a number of processes by using one or more exchange operators in the query evaluation plan. The mapping of a sequential plan to a parallel plan by inserting exchange operators permits one process per operator as well as multiple processes for one operator (using data partitioning) or multiple operators per process, which is useful for executing a complex query plan with a moderate number of processes. Earlier parallel query execution engines did not provide this degree of flexibility; the bracket model used in the Gamma design, for example, requires a separate process for each operator [DeWitt et al. 1986].

Figure 27 shows the processes created by the exchange operators in the previous figure, with each circle representing a process. Note that this set of processes is only one possible parallelization, which makes sense if the joins are on the same join attributes. Furthermore, the degrees of data parallelism, i.e., the number of processes in each process group, can be controlled using an argument to the exchange operator.

There is no reason to assume that the two models differ significantly in their performance if implemented with similar care. Both models can be implemented with a minimum of control overhead and can be combined with any partitioning scheme for load balancing. The only difference with respect to performance is that the operator model permits multiple data manipulation operators such as join in a single process, i.e., operator synchronization and data transfer between operators with a single procedure call without operating system involvement. The important advantages of the operator model are that it permits easy parallelization of an existing sequential system as well as development and maintenance of operators and algorithms in a familiar and relatively simple single-process environment [Graefe and Davison 1993].

The bracket and operator models both provide pipelining and partitioning as part of pipelined data transfer between process groups. For most algebraic operators used in database query processing, these two forms of parallelism are sufficient. However, not all operations can be easily supported by these two models. For example, in a transitive closure operator, newly inferred data is equal to input data in its importance and role for creating further data. Thus, to parallelize a single transitive closure operator, the newly created data must also be partitioned like the input data. Neither bracket nor operator model immediately allow for this need. Hence, for transitive closure operators, intraoperator parallelism based on partitioning requires that the processes exchange data among themselves outside of the stream paradigm.

The transitive closure operator is not the only operation for which this restriction holds. Other examples include the complex object assembly operator described by Keller et al. [1991] and operators for numerical optimizations as might be used in scientific databases. Both models, the bracket model and the operator model, could be extended to provide a general and efficient solution to intraoperator data exchange for intraoperator parallelism.

9.4 Load Balancing and Skew

For optimal speedup and scaleup, pieces of the processing load must be assigned carefully to individual processors and disks to ensure equal completion times for all pieces. In interoperator parallelism, operators must be grouped to ensure that no one processor becomes the bottleneck for an entire pipeline. Balanced processing loads are very hard to achieve because intermediate set sizes cannot be anticipated with accuracy and certainty in database query optimization. Thus, no existing or proposed query-processing engine relies solely on interoperator parallelism. In intraoperator parallelism, data sets must be partitioned such that the processing load is nearly equal for each processor. Notice that in particular for binary operations such as join, equal processing loads can be different from equal-sized partitions.

There are several research efforts developing techniques to avoid skew or to limit the effects of skew in parallel query processing, e.g., Baru and Frieder [1989], DeWitt et al. [1991b], Hua and Lee [1991], Kitsuregawa and Ogawa [1990], Lakshmi and Yu [1988; 1990], Omiecinski [1991], Seshadri and Naughton [1992], Walton [1989], Walton et al. [1991], and Wolf et al. [1990; 1991]. However, all of these methods have their drawbacks, for example, additional requirements for local processing to determine quantiles.

Skew management methods can be divided into basically two groups. First, *skew avoidance* methods rely on determining suitable partitioning rules before data is exchanged between processing nodes or processes. For range partitioning, quantiles can be determined or estimated from sampling the data set to be partitioned, from catalog data, e.g., histograms, or from a preprocessing step. Histograms kept on permament base data have only limited use for intermediate query processing results, in particular, if the partitioning attribute or a correlated attribute has been used in a prior selection or matching operation. However, for stored data they may be very beneficial. Sampling implies that the entire population is available for sampling because the first memory load of an intermediate result may be a very poor sample for partitioning decisions. Thus, sampling might imply that the data flow between operators be halted and an entire intermediate result be materialized on disk to ensure proper random sampling and subsequent partitioning. However, if such a halt is required anyway for processing a large set, it can be used for both purposes. For example, while creating and writing initial run files without partitioning in a parallel sort, quantiles can be determined or estimated and used in a combined partitioning and merging step.

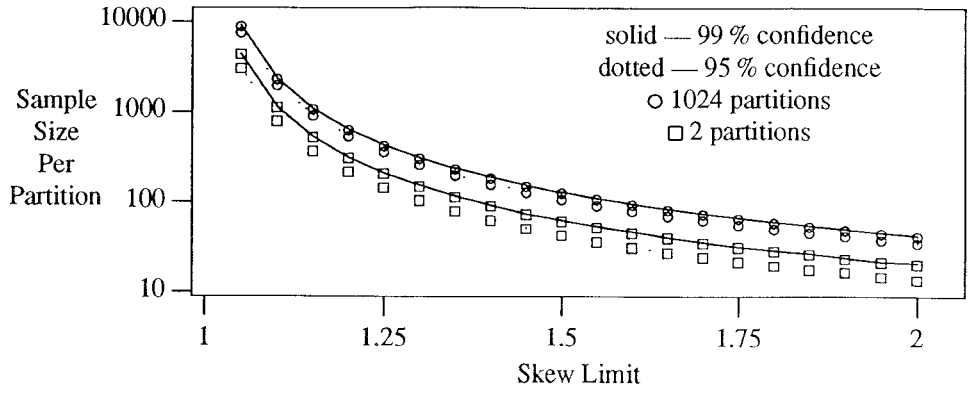


Figure 28. Skew limit, confidence, and sample size per partition.

Second, *skew resolution* repartitions some or all of the data if an initial partitioning has resulted in skewed loads. Repartitioning is relatively easy in shared-memory machines, but can also be done in distributed-memory architectures, albeit at the expense of more network activity. Skew resolution can be based on rehashing in hash partitioning or on quantile adjustment in range partitioning. Since hash partitioning tends to create fairly even loads and since network bandwidth will increase in the near future within distributed-memory machines as well as in local- and wide-area networks, skew resolution is a reasonable method for cases in which a prior processing step cannot be exploited to gather the information necessary for skew avoidance as in the sort example above.

In their recent research into sampling for load balancing, DeWitt et al. [1991b] and Seshadri and Naughton [1992] have shown that stratified random sampling can be used, i.e., samples are selected randomly not from the entire distributed data set but from each local data set at each site, and that even small sets of samples ensure reasonably balanced loads. Their definition of skew is the quotient of sizes of the largest partition and the average partition, i.e., the sum of sizes of all partitions divided by the degree of parallelism. In other words, a skew of 1.0 indicates a perfectly even distribution. Figure 28 shows the required sample sizes per partition for various skew limits, degrees of parallelism, and confidence levels. For example, to ensure a maximal skew of 1.5 among 1,000 partitions with 95% confidence, 110 random samples must be taken at each site. Thus, relatively small samples suffice for reasonably safe skew avoidance and load balancing, making precise methods unnecessary. Typically, only tens of samples per partition are needed, not several hundreds of samples at each site.

For allocation of active processing elements, i.e., CPUs and disks, the bandwidth considerations discussed briefly in the section on sorting can be generalized for parallel processes. In principal, all stages of a pipeline should be sized such that they all have bandwidths proportional to their respective data volumes in order to ensure that no stage in the pipeline becomes a bottleneck and slows the other ones down. The latency almost unavoidable in data transfer between pipeline stages should be hidden by the use of buffer memory equal in size to the product of bandwidth and latency.

9.5 Architectures and Architecture Independence

Many database research projects have investigated hardware architectures for parallelism in database systems. Stone-

braker [1986b] compared shared-nothing (distributed-memory), shared-disk (distributed-memory with multiported disks), and shared-everything (shared-memory) architectures for database use based on a number of issues including scalability, communication overhead, locking overhead, and load balancing. His conclusion at that time was that shared-everything excels in none of the points considered; shared-disk introduces too many locking and buffer coherency problems; and shared-nothing has the significant benefit of scalability to very high degrees of parallelism. Therefore, he concluded that overall shared-nothing is the preferable architecture for database system implementation. (Much of this section has been derived from Graefe et al. [1992] and Graefe and Davison [1993].)

Bhide [1988] and Bhide and Stonebraker [1988] compared architectural alternatives for transaction processing and concluded that a shared-everything (shared-memory) design achieves the best performance, up to its scalability limit. To achieve higher performance, reliability, and scalability, Bhide suggested considering shared-nothing (distributed-memory) machines with shared-everything parallel nodes. The same idea is mentioned in equally general terms by Pirahesh et al. [1990] and Boral et al. [1990], but none of these authors elaborate on the idea's generality or potential. Kitsuregawa and Ogawa's [1990] new database machine SDC uses multiple shared-memory nodes (plus custom hardware such as the Omega network and a hardware sorter), although the effect of the hardware design on operators other than join is not evaluated in their article.

Customized parallel hardware was investigated but largely abandoned after Boral and DeWitt's [1983] influential analysis that compared CPU and I/O speeds and their trends. Their analysis concluded that I/O, not processing, is the most likely bottleneck in future high-performance query execution. Subsequently, both Boral and DeWitt embarked on new database machine projects, Bubba and Gamma, that executed customized software on standard processors with local disks [Boral et al. 1990; DeWitt et al. 1990]. For scalability and availability, both projects used distributed-memory hardware with single-CPU nodes and investigated scaling questions for very large configurations.

The XPRS system, on the other hand, has been based on shared memory [Hong and Stonebraker 1991; Stonebraker et al. 1988a; 1988b]. Its designers believe that modern bus architectures can handle up to 2,000 transactions per second, and that shared-memory architectures provide automatic load balancing and faster communication than shared-nothing machines and are equally reliable and available for most errors, i.e., media failures, software, and operator errors [Gray 1990]. However, we believe that attaching 250 disks to a single machine as necessary for 2,000 transactions per second [Stonebraker et al. 1988b] requires significant special hardware, e.g., channels or I/O processors, and it is quite likely that the investment for such hardware can have greater impact on overall system performance if spent on general-purpose CPUs or disks. Without such special hardware, the performance limit for shared-memory machines is probably much lower than 2,000 transactions per second. Furthermore, there already are applications that require larger storage and access capacities.

Richardson et al. [1987] performed an analytical study of parallel join algorithms on multiple shared-memory "clusters" of CPUs. They assumed a group of clusters connected by a global bus with multiple microprocessors and shared memory in each cluster. Disk drives were attached to the busses within clusters. Their analysis suggested that the best performance is obtained by using only one cluster, i.e., a shared-memory architecture. We contend, however, that their results are due to their parameter settings, in particular small relations (typically 100 pages of 32 KB), slow CPUs (e.g., 5 μsec for a comparison, about 2–5 MIPS), a slow global network (a bus with

typically 100 Mbit/sec), and a modest number of CPUs in the entire system (128). It would be very interesting to see the analysis with larger relations (e.g., 1–10 GB), a faster network, e.g., a modern hypercube or mesh with hardware routing, and consideration of bus load and bus contention in each cluster, which might lead to multiple clusters being the better choice. On the other hand, communication between clusters will remain a significant expense. Wong and Katz [1983] developed the concept of "local sufficiency" that might provide guidance in declustering and replication to reduce data movement between nodes. Other work on declustering and limiting declustering includes Copeland et al. [1988], Fang et al. [1986], Ghandeharizadeh and DeWitt [1990], Hsiao and DeWitt [1990], and Hua and Lee [1990].

Finally, there are several hardware designs that attempt to overcome the shared-memory scaling problem, e.g., the DASH project [Anderson et al. 1988], the Wisconsin Multicube [Goodman and Woest 1988], and the Paradigm project [Cheriton et al. 1991]. However, these designs follow the traditional separation of operating system and application program. They rely on page or cache-line faulting and do not provide typical database concepts such as read-ahead and dataflow. Lacking separation of mechanism and policy in these designs almost makes it imperative to implement dataflow and flow control for database query processing within the query execution engine. At this point, none of these hardware designs has been experimentally tested for database query processing.

New software systems designed to exploit parallel hardware should be able to exploit both the advantages of shared memory, namely efficient communication, synchronization, and load balancing, and of distributed memory, namely scalability to very high degrees of parallelism and reliability and availability through independent failures. Figure 29 shows a general hierarchical architecture, which we believe combines these advantages. The important point is the combination of local busses within shared-memory parallel machines and a global interconnection network among machines. The diagram is only a very general outline of such an architecture; many details are deliberately left out and unspecified. The network could be implemented using a bus such as an ethernet, a ring, a hypercube, a mesh, or a set of point-to-point connections. The local busses may or may not be split into code and data or by address range to obtain less contention and higher bus bandwidth and hence higher scalability limits for the use of shared memory. Design and placement of caches, disk controllers, terminal connections, and local- and wide-area network connections are also left open. Tape drives or other backup devices would be connected to local busses.

Modularity is a very important consideration for such an architecture. For example, it should be possible to replace all CPU boards with upgraded models without having to replace memories or disks. Considering that new components will change communication demands, e.g., faster CPUs might require more local bus bandwidth, it is also important that the allocation of boards to local busses can be changed. For example, it should be easy to reconfigure a machine with 4×16 CPUs into one with 8×8 CPUs.

Beyond the effect of faster communication and synchronization, this architecture can also have a significant effect on control overhead, load balancing, and resulting response time problems. Investigations in the Bubba project at MCC demonstrated that large degrees of parallelism may reduce performance unless load imbalance and overhead for startup, synchronization, and communication can be kept low [Copeland et al. 1988]. For example, when placing 100 CPUs either in 100 nodes or in 10 nodes of 10 CPUs each, it is much faster to distribute query plans to all CPUs and much easier to achieve reasonably balanced loads in the second case than in the first case. Within each shared-memory parallel node, load

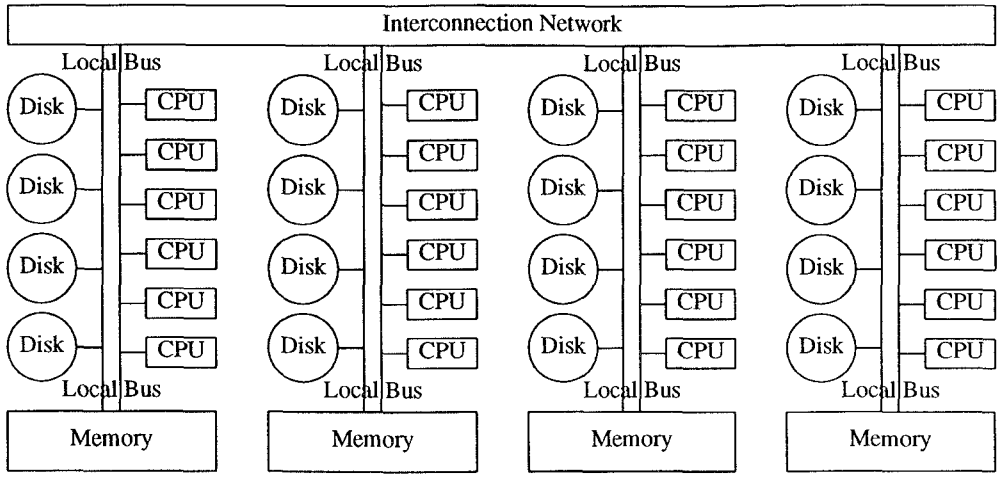


Figure 29. A hierarchical-memory architecture.

imbalance can be dealt with either by compensating allocation of resources, e.g., memory for sorting or hashing, or by relatively efficient reassignment of data to processors.

Many of today's parallel machines are built as one of the two extreme cases of this hierarchical design: a distributed-memory machine uses single-CPU nodes, while a shared-memory machine consists of a single node. Software designed for this hierarchical architecture will run on either conventional design as well as a genuinely hierarchical machine and will allow the exploration of tradeoffs in the range of alternatives in between. The most recent version of Volcano's exchange operator is designed for hierarchical memory, demonstrating that the operator model of parallelization also offers architecture- and topology-independent parallel query evaluation [Graefe and Davison 1993]. In other words, the parallelism operator is the only operator that needs to "understand" the underlying architecture, while all data manipulation operators can be implemented without concern for parallelism, data distribution, and flow control.

10. PARALLEL ALGORITHMS

In the previous section, mechanisms for parallelizing a database query execution engine were discussed. In this section, individual algorithms and their special cases for parallel execution are considered in more detail. Parallel database query processing algorithms are typically based on partitioning an input using range or hash partitioning. Either form of partitioning can be combined with sort- and hash-based query processing algorithms; in other words, the choices of partitioning scheme and local algorithm are almost always entirely orthogonal.

When building a parallel system, there is sometimes a question whether it is better to parallelize a slower sequential algorithm with better speedup behavior or a fast sequential algorithm with inferior speedup behavior. The answer to this question depends on the design goal and the planned degree of parallelism. In the few single-user database systems in use, the goal has been to minimize response time; for this goal, a slow algorithm with linear speedup implemented on highly parallel hardware might be the right choice. In multi-user systems, the goal typically is to minimize resource consumption in order to maximize throughput. For this goal, only the best sequential algorithms should be parallelized. For example, Boral and DeWitt [1983] concluded that parallelism is no substitute for effective and efficient indices. For a new parallel algorithm with impressive

speedup behavior, the question of whether or not the underlying sequential algorithm is the most efficient choice should always be considered.

10.1 Parallel Selections and Updates

Since disk I/O is a performance bottleneck in many systems, it is natural to parallelize it. Typically, either asynchronous I/O or one process per participating I/O device is used, be it a disk or an array of disks under a single controller. If a selection attribute is also the partitioning attribute, fewer than all disks will contain selection results, and the number of processes and activated disks can be limited. Notice that parallel selection can be combined very effectively with local indices, i.e., indices covering the data of a single disk or node. In general, it is most efficient to maintain indices close to the stored data sets, i.e., on the same node in a parallel database system.

For updates of partitioning attributes in a partitioned data set, items may need to move between disks and sites, just as items may move if a clustering attribute is updated. Thus, updates of partitioning attributes may require setting up data transfers from old to new locations of modified items in order to maintain the consistency of the partitioning. The fact that updating partitioning attributes is more expensive is one reason why immutable (or nearly immutable) identifiers or keys are usually used as partitioning attributes.

10.2 Parallel Sorting

Since sorting is the most expensive operation in many of today's database management systems, much research has been dedicated to parallel sorting [Baugsto and Greipsland 1989; Beck et al. 1988; Bitton and Friedland 1982; Graefe 1990a; Iyer and Dias 1990; Kitsuregawa et al. 1989b; Lorie and Young 1989; Menon 1986; Salzberg et al. 1990]. There are two dimensions along which parallel sorting methods can be classified: the number of their parallel inputs (e.g., scan or subplans executed in parallel) and the number of parallel outputs (consumers) [Graefe 1990a]. As sequential input or output restrict the throughput of parallel sorts, we assume a multiple-input multiple-output parallel sort here, and we further assume that the input items are partitioned randomly with respect to the sort attribute and that the output items should be range-partitioned and sorted within each range.

Considering that data exchange is expensive, both in terms of communication and synchronization delays, each data item should be exchanged only once between processes. Thus, most parallel sort algorithms consist of a local sort and a data exchange step. If the data exchange step is done first, quantiles must be known to ensure load balancing during the local sort step. Such quantiles can be obtained from histograms in the catalogs or by sampling. It is not necessary that the quantiles be precise; a reasonable approximation will suffice.

If the local sort is done first, the final local merging should pass data directly into the data exchange step. On each receiving site, multiple sorted streams must be merged during the data exchange step. One of the possible problems is that all producers of sorted streams first produce low key values, limiting performance by the speed of the first (single!) consumer; then all producers switch to the next consumer, etc.

If a different partitioning strategy than range partitioning is used, sorting with subsequent partitioning is not guaranteed to be deadlock free in all situations. Deadlock will occur if (1) multiple consumers feed multiple producers, (2) each producer produces a sorted stream, and each consumer merges multiple sorted streams, (3) some key-based partitioning rule is used other than range partitioning, i.e., hash partitioning, (4) flow control is enabled, and (5) the data distribution is particularly unfortunate.

Figure 30 shows a scenario with two producer and two consumer processes,

i.e., both the producer operators and the consumer operators are executed with a degree of parallelism of two. The circles in Figure 30 indicate processes, and the arrows indicate data paths. Presume that the left sort produces the stream 1, 3, 5, 7, . . . , 999, 1002, 1004, 1006, 1008, . . . , 2000 while the right sort produces 2, 4, 6, 8, . . . , 1000, 1001, 1003, 1005, 1007, . . . , 1999. The merge operations in the consumer processes must receive the first item from each producer process before they can create their first output item and remove additional items from their input buffers. However, the producers will need to produce 500 items each (and insert them into one consumer's input buffer, all 500 for one consumer) before they will send their first item to the other consumer. The data exchange buffer needs to hold 1,000 items at one point of time, 500 on each side of Figure 30. If flow control is enabled and if the exchange buffer (flow control slack) is less than 500 items, deadlock will occur.

The reason deadlock can occur in this situation is that the producer processes need to ship data in the order obtained from their input subplan (the sort in Figure 30) while the consumer processes need to receive data in sorted order as required by the merge. Thus, there are two sides which both require absolute control over the order in which data pass over the process boundary. If the two requirements are incompatible, an unbounded buffer is required to ensure freedom from deadlock.

In order to avoid deadlock, it must be ensured that one of the five conditions listed above is not satisfied. The second condition is the easiest to avoid, and should be focused on. If the receiving processes do not perform a merge, i.e., the individual input streams are not sorted, deadlock cannot occur because the slack given in the flow control must be somewhere, either at some producer or some consumer or several of them, and the process holding the slack can continue to process data, thus preventing deadlock.

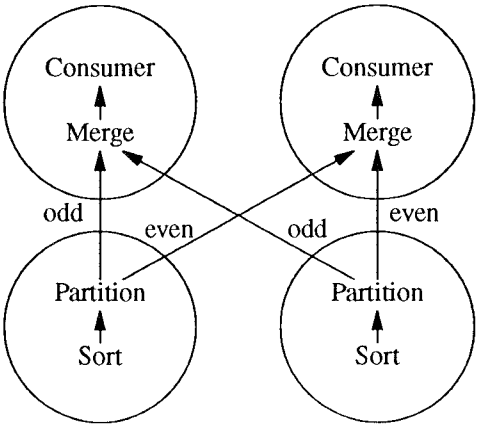


Figure 30. Scenario with possible deadlock.

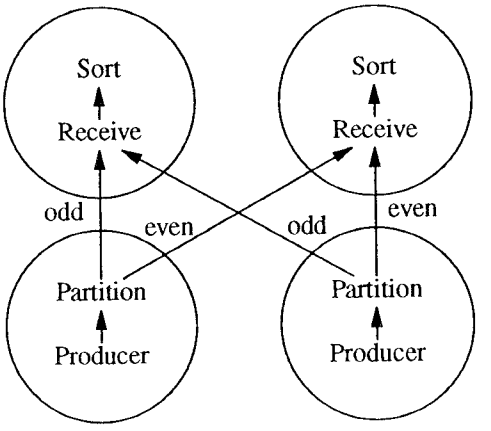


Figure 31. Deadlock-free scenario.

Our recommendation is to avoid the above situation, i.e., to ensure that such query plans are never generated by the optimizer. Consider for which purposes such a query plan would be used. The typical scenario is that multiple processes perform a merge join of two inputs, and each (or at least one) input is sorted by several producer processes. An alternative scenario that avoids the problem is shown in Figure 31. Result data are partitioned and sorted as in the previous scenario. The important difference is that the consumer processes do not merge multiple sorted incoming streams.

One of the conditions for the deadlock problem illustrated in Figure 30 is that

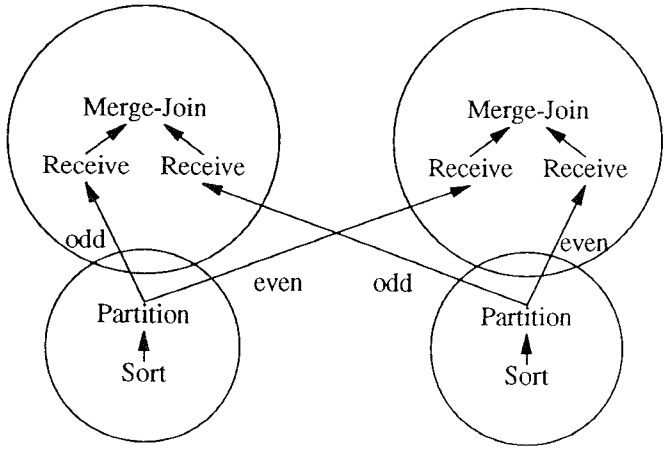


Figure 32. Deadlock danger due to a binary operator in the consumer.

there are multiple producers and multiple consumers of a single logical data stream. However, a very similar deadlock situation can occur with single-process producers if the consumer includes an operation that depends on ordering, typically merge-join. Figure 32 illustrates the problem with a merge-join operation executed in two consumer processes. Notice that the left and right producers in Figure 32 are different inputs of the merge-join, not processes executing the same operators as in Figures 30 and 31. The consumer in Figure 32 is still one operator executed by two processes. Presume that the left sort produces the stream 1, 3, 5, 7, . . . , 999, 1002, 1004, 1006, 1008, . . . , 2000 while the right sort produces 2, 4, 6, 8, . . . , 1000, 1001, 1003, 1005, 1007, . . . , 1999. In this case, the merge-join has precisely the same effect as the merging of two parts of one logical data stream in Figure 30. Again, if the data exchange buffer (flow control slack) is too small, deadlock will occur. Similar to the deadlock avoidance tactic in Figure 31, deadlock in Figure 32 can be avoided by placing the sort operations into the consumer processes rather than into the producers. However, there is an additional solution for the scenario in Figure 32, namely, moving only one of the sort operators, not both, into the consumer processes.

If moving a sort operation into the consumer process is not realistic, e.g., because the data already are sorted when they are retrieved from disk as in a B-tree scan, alternative parallel execution strategies must be found that do not require repartitioning and merging of sorted data between the producers and consumers. There are two possible cases. In the first case, if the input data are not only sorted but also already partitioned systematically, i.e., range or hash partitioned, on the attribute(s) considered by the consumer operator, e.g., the by-list of an aggregate function or the join attribute, the process boundary and data exchange could be removed entirely. This implies that the producer operator, e.g., the B-tree scan, and the consumer, e.g., the merge-join, are executed by the same process group and therefore with the same degree of parallelism.

In the second case, although sorted on the relevant attribute within each partition, the operator's data could be partitioned either round-robin or on a different attribute. For a join, a fragment-and-replicate matching strategy could be used [Epstein et al. 1978; Epstein and Stonebraker 1980; Lohman et al. 1985], i.e., the join should execute within the same threads as the operator producing sorted output while the second input is replicated to all instances of the

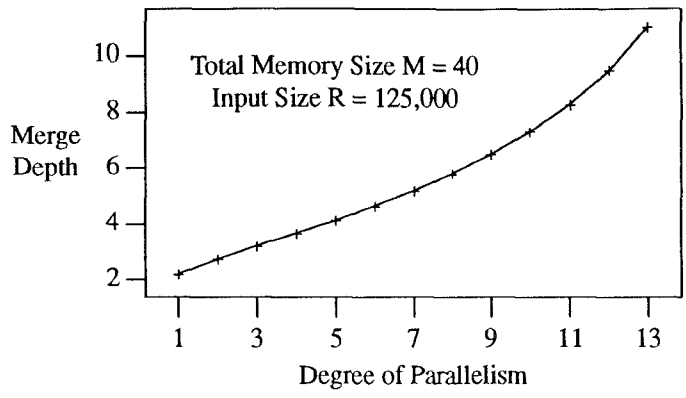


Figure 33. Merge depth as a function of parallelism.

join. Note that fragment-and-replicate methods do not work correctly for semi-join, outer join, difference, and union, i.e., when an item is replicated and is inserted (incorrectly) multiple times into the global output. A second solution that works for all operators, not only joins, is to execute the consumer of the sorted data in a single thread. Recall that multiple consumers are required for a deadlock to occur. A third solution that is correct for all operators is to send dummy items containing the largest key seen so far from a producer to a consumer if no data have been exchanged for a predetermined amount of time (data volume, key range). In the examples above, if a producer must send a key to all consumers at least after every 100 data items processed in the producer, the required buffer space is bounded, and deadlock can be avoided. In some sense, this solution is very simple; however, it requires that not only the data exchange mechanism but also sort-based algorithms such as merge-join must "understand" dummy items. Another solution is to exchange all data without regard to sort order, i.e., to omit merging in the data exchange mechanism, and to sort explicitly after repartitioning is complete. For this sort, replacement selection might be more effective than quicksort for generating initial runs because the runs would probably be much larger than twice the size of memory.

A final remark on deadlock avoidance: Since deadlock can only occur if the consumer process merges, i.e., not only the producer but also the consumer operator try to determine the order in which data cross process boundaries, the deadlock problem only exists in a query execution engine based on sort-based set-processing algorithms. If hash-based algorithms were used for aggregation, duplicate removal, join, semi-join, outer join, intersection, difference, and union, the need for merging and therefore the danger of deadlock would vanish.

An interesting parallel sorting method with balanced communication and without the possibility of deadlock in spite of local sort followed by data exchange (if the data distribution is known a priori) is to sort locally only by the position within the final partition and then exchange data guaranteeing a balanced data flow. This method might be best seen in an example: Consider 10 partitions with key values from 0 to 999 in a uniform distribution. The goal is to have all key values between 0 to 99 sorted on site 0, between 100 and 199 sorted on site 1, etc. First, each partition is sorted locally at its original site, without data exchange, on the last two digits only, ignoring the first digit. Thus, each site has a sequence such as 200, 301, 401, 902, 2, 603, 804, 605, 105, 705,...,999, 399. Now each site sends data to its correct final destination. Notice that each site sends data simulta-

neously to all other sites, creating a balanced data flow among all producers and consumers. While this method seems elegant, its problem is that it requires fairly detailed distribution information to ensure the desired balanced data flow.

In shared-memory machines, memory must be divided over all concurrent sort processes. Thus, the more processes are active, the less memory each one can get. The importance of this memory division is the limitation it puts on the size of initial runs and on the fan-in in each merge process. In other words, large degrees of parallelism may impede performance because they increase the number of merge levels. Figure 33 shows how the number of merge levels grows with increasing degrees of parallelism, i.e., decreasing memory per process and merge fan-in. For input size R, total memory size M, and P parallel processes, the merge depth L is $L = \log_{M/P-1}((R/P)/(M/P)) = \log_{M/P-1}(R/M)$. The optimal degree of parallelism must be determined considering the tradeoff between parallel processing and large fan-ins, somewhat similar to the tradeoff between fan-in and cluster size. Extending this argument using the duality of sorting and hashing, too much parallelism in hash partitioning on shared-memory machines can also be detrimental, both for aggregation and for binary matching [Hong and Stonebraker 1993].

10.3 Parallel Aggregation and Duplicate Removal

Parallel algorithms for aggregation and duplicate removal are best divided into a local step and a global step. First, duplicates are eliminated locally, and then data are partitioned to detect and remove duplicates from different original sites. For aggregation, local and global aggregate functions may differ. For example, to perform a global count, the local aggregation counts while the global aggregation sums local counts into a global count.

For local hash-based aggregation, a special technique might improve performance. Instead of creating overflow files locally to resolve hash table overflow, items can be moved directly to their final site. Hopefully, this site can aggregate them immediately into the local hash table because a similar item already exists. In many recent distributed-memory machines, it is faster to ship an item to another site than to do a local disk I/O. In fact, some distributed-memory vendors attach disk drives not to the primary processing nodes but to special "I/O nodes" because network delay is negligible compared to I/O time, e.g., in Intel's iPSC/2 and its subsequent parallel architectures.

The advantage is that disk I/O is required only when the aggregation output size does not fit into the aggregate memory available on all machines, while the standard local aggregation-exchange-global aggregation scheme requires local disk I/O if any local output size does not fit into a local memory. The difference between the two is determined by the degree to which the original input is already partitioned (usually not at all), making this technique very beneficial.

10.4 Parallel Joins and Other Binary Matching Operations

Binary matching operations such as join, semi-join, outer join, intersection, union, and difference are different than the previous operations exactly because they are binary. For bushy parallelism, i.e., a join for which two subplans create the two inputs independently from one another in parallel, we might consider symmetric hash join algorithms. Instead of differentiating between build and probe inputs, the symmetric hash join uses two hash tables, one for each input. When a data item (or packet of items) arrives, the join algorithm first determines which input it came from and then joins the new data item with the hash table built from the other input as well as inserting the new data item into its hash table such that data items from the other input arriving later can be joined correctly. Such a symmetric hash join algorithm has been used in XPRS, a shared-memory high-performance extensible-

relational database system [Hong and Stonebraker 1991; 1993; Stonebraker et al. 1988a; 1988b] as well as in Prisma/DB, a shared-nothing main-memory database system [Wilschut 1993; Wilschut and Apers 1993]. The advantage of symmetric matching algorithms is that they are independent of the data rates of the inputs; their disadvantage is that they require that both inputs fit in memory, although one hash table can be dropped when one input is exhausted.

For parallelizing a single binary matching operation, there are basically two techniques, called here *symmetric partitioning* and *fragment and replicate*. In both cases, the global result is the union (concatenation) of all local results. Some algorithms exploit the topology of certain architectures, e.g., ring- or cube-based communication networks [Baru and Frieder 1989; Omiecinski and Lin 1989].

In the symmetric partitioning methods, both inputs are partitioned on the attributes relevant to the operation (i.e., the join attribute for joins or all attributes for set operations), and then the operation is performed at each site. Both the Gamma and the Teradata database machines use this method. Notice that the partitioning method (usually hashed) and the local join method are independent of each other; Gamma and Grace use hash joins while Teradata uses merge-join.

In the fragment-and-replicate methods, one input is partitioned, and the other one is broadcast to all sites. Typically, the larger input is partitioned by not moving it at all, i.e., the existing partitions are processed at their locations prior to the binary matching operation. Fragment-and-replicate methods were considered the join algorithms of choice in early distributed database systems such as R*, SDD-1, and distributed Ingres, because communication costs overshadowed local processing costs and because it was cheaper to send a small input to a small number of sites than to partition both a small and a large input. Note that fragment-and-replicate methods do not work correctly for semi-join, outer join, difference, and union, namely, when an item is replicated and is inserted into the output (incorrectly) multiple times.

A technique for reducing network traffic during join processing in distributed database systems uses redundant semi-joins [Bernstein et al. 1981; Chiu and Ho 1980; Gouda and Dayal 1981], an idea that can also be used in distributed-memory parallel systems. For example, consider the join on a common attribute A of relations R and S stored on two different nodes in a network, say r and s. The semi-join method transfers a duplicate-free projection of R on A to s, performs a semi-join there to determine the items in S that actually participate in the join result, and ships these items to r for the actual join. In other words, based on the relational algebra law that

$$R \; JOIN \; S = R \; JOIN \; (S \; SEMIJOIN \; R),$$

cost savings of not shipping all of S were realized at the expense of projecting and shipping the $R.A$-column and executing the semi-join. Of course, this idea can be used symmetrically to reduce R or S or both, and all operations (projection, duplicate removal, semi-join, and final join) can be executed in parallel on both r and s or on more than two nodes using the parallel join strategies discussed earlier in this section. Furthermore, there are probabilistic variants of this idea that use bit vector filtering instead of semi-joins, discussed later in its own section.

Roussopoulos and Kang [1991] recently showed that symmetric semi-joins are particularly useful. Using the equalities (for a join of relations R and S on attribute A)

$$\begin{aligned} & R \; JOIN \; S \\ &= R \; JOIN \; (S \; SEMIJOIN \; \pi_A R) \\ &= (R \; SEMIJOIN \; \pi_A (S \; SEMIJOIN \; \pi_A R)) \\ &\quad JOIN \; (S \; SEMIJOIN(\mathrm{a}) \pi_A R) \qquad (\mathrm{a}) \\ &= (R \; \overline{SEMIJOIN} \; \pi_A (S \; \overline{SEMIJOIN} \; \pi_A R)) \\ &\quad JOIN \; (S \; (SEMIJOIN(\mathrm{b}) \pi_A R), \qquad (\mathrm{b}) \end{aligned}$$

they designed a four-step procedure to compute the join of two relations stored at two sites. First, the first relation's join attribute column $R.A$ is sent duplicate free to the other relation's site, s. Second, the first semi-join is computed at s, and either the matching values (term (a) above) or the nonmatching values (term (b) above) of the join column $S.A$ are sent back to the first site, r. The choice between (a) and (b) is made based on the number of matching and nonmatching[17] values of $S.A$. Third, site r determines which items of R will participate in the join R *JOIN* S, i.e., R *SEMIJOIN* S. Fourth, both input sites send exactly those items that will participate in the join R *JOIN* S to the site that will compute the final result, which may or may not be one of the two input sites. Of course, this two-site algorithm can be used across any number of sites in a parallel query evaluation system.

Typically, each data item is exchanged only once across the interconnection network in a parallel algorithm. However, for parallel systems with small communication overhead, in particular for shared-memory systems, and in parallel processing systems with processors without local disk(s), it may be useful to spread each overflow file over all available nodes and disks in the system. The disadvantage of the scheme may be communication overhead; however, the advantages of load balancing and cumulative bandwidth while reading a partition file have led to the use of this scheme both in the Gamma and SDC database machines, called *bucket spreading* in the SDC design [DeWitt et al. 1990; Kitsuregawa and Ogawa 1990].

For parallel non-equi-joins, a symmetric fragment-and-replicate method has been proposed by Stamos and Young [Stamos and Young 1989]. As shown in Figure 34, processors are organized into rows and columns. One input relation is partitioned over rows, and partitions are replicated within each row, while the other input is partitioned and replicated over columns. Each item from one input "meets" each item from the other input at exactly one site, and the global join result is the concatenation of all local joins.

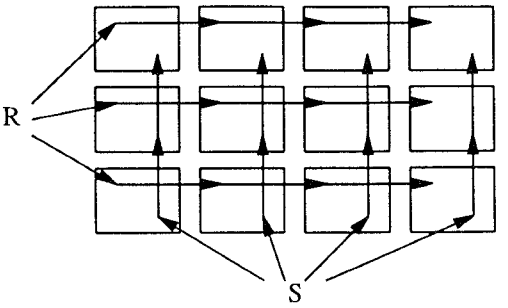


Figure 34. Symmetric fragment-and-replicate join.

Avoiding partitioning as well as broadcasting for many joins can be accomplished with a physical database design that considers frequently performed joins and distributes and replicates data over the nodes of a parallel or distributed system such that many joins already have their input data suitably partitioned. Katz and Wong formalized this notion as *local sufficiency* [Katz and Wong 1983; Wong and Katz 1983]; more recent research on the issue was performed in the Bubba project [Copeland et al. 1988].

For joins in distributed systems, a third class of algorithms, called *fetch-as-needed*, was explored. The idea of these algorithms is that one site performs the join by explicitly requesting (fetching) only those items from the other input needed to perform the join [Daniels and Ng 1982; Williams et al. 1982]. If one input is very small, fetching only the necessary items of the larger input might seem advantageous. However, this algorithm is a particularly poor implementation of a semi-join technique discussed above. Instead of requesting items or values one by one, it seems better to first project all join attribute values, ship (stream) them across the network, perform the semi-join using any local binary matching algorithm, and then stream exactly those items back that will be re-

[17] $\overline{SEMIJOIN}$ stands for the anti-semi-join, which determines those items in the first input that do *not* have a match in the second input.

quired for the join back to the first site. The difference between the semi-join technique and fetch-as-needed is that the semi-join scans the first input twice, once to extract the join values and once to perform the real join, while fetch as needed needs to work on each data item only once.

10.5 Parallel Universal Quantification

In our earlier discussion on sequential universal quantification, we discussed four algorithms for universal quantification or relational division, namely, naive division (a direct, sort-based algorithm), hash-division (direct, hash based), and sort- and hash-based aggregation (indirect) algorithms, which might require semi-joins and duplicate removal in the inputs.

For naive division, pipelining can be used between the two sort operators and the division operator. However, both quotient partitioning and divisor partitioning can be employed as described below for hash-division.

For algorithms based on aggregation, both pipelining and partitioning can be applied immediately using standard techniques for parallel query execution. While partitioning seems to be a promising approach, it has an inherent problem due to the possible need for a semi-join. Recall that in the example for universal quantification using *Transcript* and *Course* relations, the join attribute in the semi-join (*course-no*) is different than the grouping attribute in the subsequent aggregation (*student-id*). Thus, the *Transcript* relation has to be partitioned twice, once for the semi-join and once for the aggregation.

For hash-division, pipelining has only limited promise because the entire division is performed within a single operator. However, both partitioning strategies discussed earlier for hash table overflow can be employed for parallel execution, i.e., quotient partitioning and divisor partitioning [Graefe 1989; Graefe and Cole 1993].

For hash-division with quotient partitioning, the divisor table must be *replicated* in the main memory of all participating processors. After replication, all local hash-division operators work completely independent of each other. Clearly, replication is trivial for shared-memory machines, in particular since a single copy of the divisor table can be shared without synchronization among multiple processes once it is complete.

When using divisor partitioning, the resulting partitions are processed in parallel instead of in phases as discussed for hash table overflow. However, instead of tagging the quotient items with phase numbers, processor network addresses are attached to the data items, and the collection site divides the set of all incoming data items over the set of processor network addresses. In the case that the central collection site is a bottleneck, the collection step can be decentralized using quotient partitioning.

11. NONSTANDARD QUERY PROCESSING ALGORITHMS

In this section, we briefly review the query processing needs of data models and database systems for nonstandard applications. In many cases, the logical operators defined for new data models can use existing algorithms, e.g., for intersection. The reason is that for processing, bulk data types such as array, set, bag (multi-set), or list are represented as sequences similar to the streams used in the query processing techniques discussed earlier, and the algorithms to manipulate these bulk types are equal to the ones used for sets of tuples, i.e., relations. However, some algorithms are genuinely different from the algorithms we have surveyed so far. In this section, we review operators for nested relations, temporal and scientific databases, object-oriented databases, and more meta-operators for additional query processing control.

There are several reasons for integrating these operators into an algebraic query-processing system. First, it permits efficient data transfer from the database to the application embodied in these operators. The interface between

database operators is designed to be as efficient as possible; the same efficient interface should also be used for applications. Second, operator implementors can take advantage of the control provided by the meta-operators. For example, an operator for a scientific application can be implemented in a single-process environment and later parallelized with the exchange operator. Third, query optimization based on algebraic transformation rules can cover all operators, including operations that are normally considered database application code. For example, using algebraic optimization tools such as the EXODUS and Volcano optimizer generators [Graefe and DeWitt 1987; Graefe et al. 1992; Graefe and McKenna 1993], optimization rules that can move an unusual database operator in a query plan are easy to implement. For a sampling operator, a rule might permit transforming an algebra expression to query a sample instead of sampling a query result.

11.1 Nested Relations

Nested relations, or Non-First-Normal-Form (NF^2) relations, permit relation-valued attributes in addition to atomic values such as integers and strings used in the normal or "flat" relational model. For example, in an order-processing application, the set of individual line items on each order could be represented as a nested relation, i.e., as part of an order tuple. Figure 35 shows an NF^2 relation with two tuples with two and three nested tuples and the equivalent normalized relations, which we call the master and detail relations. Nested relations can be used for all one-to-many relationships but are particularly well suited for the representation of "weak entities" in the Entity-Relationship (ER) Model [Chen 1976], i.e., entities whose existence and identification depend on another entity as for order entries in Figure 35. In general, nested subtuples may include relation-valued attributes, with arbitrary nesting depth. The advantages of the NF^2 model are that component relationships can be represented more naturally than in the fully normalized model; many frequent join operations can be avoided, and structural information can be used for physical clustering. Its disadvantage is the added complexity, in particular, in storage management and query processing.

Several algebras for nested relations have been defined, e.g., Deshpande and Larson [1991], Ozsoyoglu et al. [1987], Roth et al. [1988], Schek and Scholl [1986], Tansel and Garnett [1992]. Our discussion here focuses not on the conceptual design of NF^2 algebras but on algorithms to manipulate nested relations.

Two operations required in NF^2 database systems are operations that transform an NF^2 relation into a normalized relation with atomic attributes only, and vice versa. The first operation is frequently called *unnest* or *flatten*; the opposite direction is called the *nest* operation. The unnest operation can be performed in a single scan over the NF^2 relation that includes the nested subtuples; both normalized relations in Figure 35 and their join can be derived readily enough from the NF^2 relation. The nest operation requires grouping of tuples in the detail relation and a join with the master relation. Grouping and join can be implemented using any of the algorithms for aggregate functions and binary matching discussed earlier, i.e., sort- and hash-based sequential and parallel methods. However, in order to ensure that unnest and nest operations are exact inverses of each other, some structural information might have to be preserved in the unnest operation. Ozsoyoglu and Wang [1992] present a recent investigation of "keying methods" for this purpose.

All operations defined for flat relations can also be defined for nested relations, in particular: selection, join, and set operations (union, intersection, difference). For selections, additional power is gained with selection conditions on subtuples and sets of subtuples using set comparisons or existential or universal quantification. In principle, since a nested rela-

Order -No	Customer -No	Date	Items	
			Part-No	Count
110	911	910902	4711	8
			2345	7
112	912	910902	9876	3
			2222	1
			2357	9

Order-No	Customer-No	Date
110	911	910902
112	912	910902

Order-No	Part-No	Quantity
110	4711	8
110	2345	7
112	9876	3
112	2222	1
112	2357	9

Figure 35. Nested relation and equivalent flat relations.

tion is a relation, any relational calculus and algebra expression should be permitted for it. In the example in Figure 35, there may be a selection of orders in which the ordered quantity of all items is more than 100, which is a universal quantification. The algorithms for selections with quantifier are similar to the ones discussed earlier for flat relations, e.g., relational semi-join and division, but are easier to implement because the grouping process built into the flat-relational algorithms is inherent in the nested tuple structure.

For joins, similar considerations apply. Matching algorithms discussed earlier can be used in principle. They may be more complex if the join predicate involves subrelations, and algorithm combinations may be required that are derived from a flat-relation query over flat relations equivalent to the NF^2 query over the nested relations. However, there should be some performance improvements possible if the grouping of values in the nested relations can be exploited, as for example, in the join algorithms described by Rosenthal et al. [1991].

Deshpande and Larson [1992] investigated join algorithms for nested relations because "the purpose of nesting in order to store precomputed joins is defeated if it is unnested every time a join is performed on a subrelation." Their algorithm, (parallel) partitioned nested-hashed-loops, joins one relation's subrelations with a second, flat relation by creating an in-memory hash table with the flat relation. If the flat relation is larger than memory, memory-sized segments are loaded one at a time, and the nested relation is scanned repeatedly. Since an outer tuple of the nested relation might have matches in multiple segments of the flat relation, a final merging pass is required. This join algorithm is reminiscent of hash-division, the flat relation taking the role of the divisor and the nested tuples replacing the quotient table entries with their bit maps.

Sort-based join algorithms for nested relations require either flattening the nested tuples or scanning the sorted flat relation for each nested tuple, somewhat reminiscent of naive division. Neither alternative seems very promising for large

inputs. Sort semantics and appropriate sort algorithms including duplicate removal and grouping have been considered by Saake et al. [1989] and Kuespert et al. [1989]. Other researchers have focused on storage and retrieval methods for nested relations and operations possible with single scans [Dadam et al. 1986; Deppisch et al. 1986; Deshpande and Van Gucht 1988; Hafez and Ozsoyoglu 1988; Ozsoyoglu and Wang 1992; Scholl et al. 1987; Scholl 1988].

11.2 Temporal and Scientific Database Management

For a variety of reasons, management and manipulation of statistical, temporal, and scientific data are gaining interest in the database research community. Most work on temporal databases has focused on semantics and representation in data models and query languages [McKenzie and Snodgrass 1991; Snodgrass 1990]; some work has considered special storage structures, e.g., Ahn and Snodgrass [1988], Lomet and Salzberg [1990b], Rotem and Segev [1987], Severance and Lohman [1976], algebraic operators, e.g., temporal joins [Gunadhi and Segev 1991], and optimization of temporal queries, e.g., Gunadhi and Segev [1990], Leung and Muntz [1990; 1992], Segev and Gunadhi [1989]. While logical query algebras require extensions to accommodate time, only some storage structures and algorithms, e.g., multidimensional indices, differential files, and versioning, and the need for approximate selection and matching (join) predicates are new in the query execution algorithms for temporal databases.

A number of operators can be identified that both add functionality to database systems used to process scientific data and fit into the database query processing paradigm. DeWitt et al. [1991a] considered algorithms for join predicates that express proximity, i.e., join predicates of the form $R.A - c_1 \leq S.B \leq R.A + c_2$ for some constants c_1 and c_2. Such join predicates are very different from the usual use of relational join. They do not reestablish relationships based on identifying keys but match data values that express a dimension in which distance can be defined, in particular, time. Traditionally, such join predicates have been considered non-equi-joins and were evaluated by a variant of nested-loops join. However, such "band joins" can be executed much more efficiently by a variant of merge-join that keeps a "window" of inner relation tuples in memory or by a variant of hash join that uses range partitioning and assigns some build tuples to multiple partition files. A similar partitioning model must be used for parallel execution, requiring multi-cast for some tuples. Clearly, these variants of merge-join and hash join will outperform nested loops for large inputs, unless the band is so wide that the join result approaches the Cartesian product.

For storage and management of the massive amounts of data resulting from scientific experiments, database techniques are very desirable. Operators for processing time series in scientific databases are based on an interpretation of a stream between operators not as a set of items (as in most database applications) but as a sequence in which the order of items in a stream has semantic meaning. For example, data reduction using interpolation as well as extrapolation can be performed within the stream paradigm. Similarly, digital filtering [Hamming 1977] also fits the stream-processing protocol very easily. Interpolation, extrapolation, and digital filtering were implemented in the Volcano system with a single algorithm (physical operator) to verify this fit, including their optimization and parallelization [Graefe and Wolniewicz 1992; Wolniewicz and Graefe 1993]. Another promising candidate is visualization of single-dimensional arrays such as time series.

Problems that do not fit the stream paradigm, e.g., many matrix operations such as transformations used in linear algebra, Laplace or Fast Fourier Transform, and slab (multidimensional subarray) extraction, are not as easy to integrate into database query processing

systems. Some of them seem to fit better into the storage management subsystem rather than the algebraic query execution engine. For example, slab extraction has been integrated into the NetCDF storage and access software [Rew and Davis 1990; Unidata 1991]. However, it is interesting to note that sorting is a suitable algorithm for permuting the linear representation of a multidimensional array, e.g., to modify the hierarchy of dimensions in the linearization (row- vs. column-major linearization). Since the final position of all elements can be predicted from the beginning of the operation, such "sort" algorithms can be based on merging or range partitioning (which is yet another example of the duality of sort- and hash- (partitioning-) based data manipulation algorithms).

11.3 Object-Oriented Database Systems

Research into query processing for extensible and object-oriented systems has been growing rapidly in the last few years. Most proposals or implementations use algebras for query processing, e.g., Albert [1991], Cluet et al. [1989], Graefe and Maier [1988], Guo et al. [1991], Mitschang [1989], Shaw and Zdonik [1989a; 1989b; 1990], Straube and Ozsu [1989], Vandenberg and DeWitt [1991], Yu and Osborn [1991]. These algebras resemble relational algebra in the sense that they focus on bulk data types but are generalized to support operations on arrays, lists, etc., user-defined operations (methods) on instances, heterogeneous bulk types, and inheritance. The use of algebras permits several important conclusions. First, naive execution models that execute programs as if all data were in memory are not the only alternative. Second, data manipulation operators can be designed and implemented that go beyond data retrieval and permit some amount of data reduction, aggregation, and even inference. Third, algebraic execution techniques including the stream paradigm and parallel execution can be used in object-oriented data models and database systems. Fourth, algebraic optimization techniques will continue to be useful.

Associative operations are an important part in all object-oriented algebras because they permit reducing large amounts of data to the interesting subset of the database suitable for further consideration and processing. Thus, set-processing and set-matching algorithms as discussed earlier in this survey will be found in object-oriented systems, implemented in such a way that they can operate on heterogeneous sets. The challenge for query optimization is to map a complex query involving complex behavior and complex object structures to primitives available in a query execution engine. Translating an initial request with abstract data types and encapsulated behavior coded in a computationally complete language into an internal form that both captures the entire query's semantics and allows effective query optimization is still an open research issue [Daniels et al. 1991; Graefe and Maier 1988].

Beyond associative indices discussed earlier, object-oriented systems can also benefit from special relationship indices, i.e., indices that contain condensed information about interobject references. In principle, these index structures are similar to join indices [Valduriez 1987] but can be generalized to support multiple levels of referencing. Examples for indices in object-oriented database systems include the work of Maier and Stein [1986] in the Gemstone object-oriented database system product, Bertino [1990; 1991] and Bertino and Kim [1989], in the Orion project and Kemper et al. [1991] and Kemper and Moerkotte [1990a; 1990b] in the GOM project. At this point, it is too early to decide which index structures will be the most useful because the entire field of query processing in object-oriented systems is still developing rapidly, from query languages to algebra design, algorithm repertoire, and optimization techniques. Other areas of intense current research interest are buffer management and clustering of objects on disk.

One of the big performance penalties in object-oriented database systems is "pointer chasing" (using OID references) which may involve object faults and disk read operations at widely scattered locations, also called "goto's on disk." In order to reduce I/O costs, some systems use what amounts to main-memory databases or map the entire database into virtual memory. For systems with an explicit database on disk and an in-memory buffer, there are various techniques to detect object faults; some commercial object-oriented database systems use hardware mechanisms originally perceived and implemented for virtual-memory systems. While such hardware support makes fault detection faster, it does not address the problem of expensive I/O operations. In order to reduce actual I/O cost, read-ahead and planned buffering must be used. Palmer and Zdonik [1991] recently proposed keeping access patterns or sequences and activating read-ahead if accesses equal or similar to a stored pattern are detected. Another recent proposal for efficient assembly of complex objects uses a window (a small set) of open references and resolves, at any point of time, the most convenient one by fetching this object or component from disk, which has shown dramatic improvements in disk seek times and makes complex object retrieval more efficient and more independent of object clustering [Keller et al. 1991]. Policies and mechanisms for efficient parallel complex object assembly are an important challenge for the developers of next-generation object-oriented database management systems [Maier et al. 1992].

11.4 More Control Operators

The exchange operator used for parallel query processing is not a normal operator in the sense that it does not manipulate, select, or transform data. Instead, the exchange operator provides control of query processing in a way orthogonal to what a query does and what algorithms it uses. Therefore, we call it a meta- or control operator. There are several other control operators that can be used in database query processing, and we survey them briefly in this section.

In situations in which an intermediate result is used repeatedly, e.g., a nested-loops join with a composite inner input, either the intermediate result is derived many times, or it is saved in a temporary file during its first derivation and then retrieved from this file while serving subsequent requests. This situation arises not only with nested-loops join but also with other algorithms, e.g., sort-based universal quantification [Smith and Chang 1975]. Thus, it might be useful to encapsulate this functionality in a new algorithm, which we call the *store-and-scan* operator.

The store-and-scan operator permits three generalizations. First, if the first consumption of the intermediate result might actually not need it entirely, e.g., a nested-loops semi-join which terminates each inner scan after the first match, the operator should be switched to derive only the necessary data items (which implies leaving the input plan ready to produce more data later) or to save the entire intermediate result in the temporary file right away in order to permit release of all resources in the subplan. Second, subsequent scans might permit starting not at the beginning of the temporary file but at some later point. This version is useful if many duplicates exist in the inputs of one-to-one matching algorithms based on merge-join. Third, in some execution strategies for correlated SQL subqueries, the plan corresponding to the inner block is executed once for each tuple in the outer block. The tuples of the outer block provide different correlation values, although each value may occur repeatedly. In order to ensure that the inner plan is executed only once for each outer correlation value, the store-and-scan operator could retain information about which part of its temporary file corresponds to which correlation value and restrict each scan appropriately.

Another use of a temporary file is to support common subexpressions, which

can be executed efficiently with an operator that passes the result of a common subexpression to multiple consumers, as mentioned briefly in the section on the architecture of query execution engines. The problem is that multiple consumers, typically demand-driven and demand-driving their inputs, will request items of the common subexpression result at different times or rates. The two standard solutions are either to execute the common subexpression into a temporary file and let each consumer scan this file at will or to determine which consumer will be the first to require the result of the common subexpression, to execute the common subexpression as part of this consumer and to create a file with the common subexpression result as a by-product of the first consumer's execution. Instead, we suggest a new meta-operator, which we call the *split* operator, to be placed at the top of the common subexpression's plan and which can serve multiple consumers at their own paces. It automatically performs buffering to account for different paces, uses temporary disk space if the discrepancies are too wide, and is suitably parameterized to permit both standard solutions described above.

In query processing systems, data flow is usually paced or driven from the top, the consumer. The leftmost diagram of Figure 36 shows the control flow of normal iterators. (Notice that the arrows in Figure 36 show the *control* flow; the data flow of all diagrams in Figure 36 is assumed to be upward. In data-driven data flow, control and data flows point in the same direction; in demand-driven data flow, their directions oppose each other.) However, in real-time systems that capture data from experiments, this approach may not be realistic because the data source, e.g., a satellite receiver, has to be able to unload data as they arrive. In such systems, data-driven operators, shown in the second diagram of Figure 36, might be more appropriate. To combine the algorithms implemented and used for query processing with such real-time data capture requirements, one could design data *flow translation* control operators. The first such operator which we call the *active scheduler* can be used between a demand-driven producer and a data-driven consumer. In this case, neither operator will schedule the other; therefore, an *active scheduler* that demands items from the producer and forces them onto the consumer will glue these two operators together. An active-scheduler schematic is shown in the third diagram of Figure 36. The opposite case, a data-driven producer and a demand-driven consumer, has two operators, each trying to schedule the other one. A second flow control operator, called the *passive scheduler*, can be built that accepts procedure calls from either neighbor and resumes the other neighbor in a coroutine fashion to ensure that the resumed neighbor will eventually demand the item the scheduler just received. The final diagram of Figure 36 shows the control flow of a passive scheduler. (Notice that this case is similar to the bracket model of parallel operator implementations discussed earlier in which an operating system or networking software layer had to be placed between data manipulation operators and perform buffering and flow control.)

Finally, for very complex queries, it might be useful to break the data flow between operators at some point, for two reasons. First, if too many operators run in parallel, contention for memory or temporary disks might be too intense, and none of the operators will run as efficiently as possible. A long series of hybrid hash joins in a right-deep query plan illustrates this situation. Second, due to the inherent error in selectivity estimation during query optimization [Ioannidis and Christodoulakis 1991; Mannino et al. 1988], it might be worthwhile to execute only a subset of a plan, verify the correctness of the estimation, and then resume query processing with another few steps. After a few processing steps have been performed, their result size and other statistical properties such as minimum and maximum and approximate number of duplicate values can be

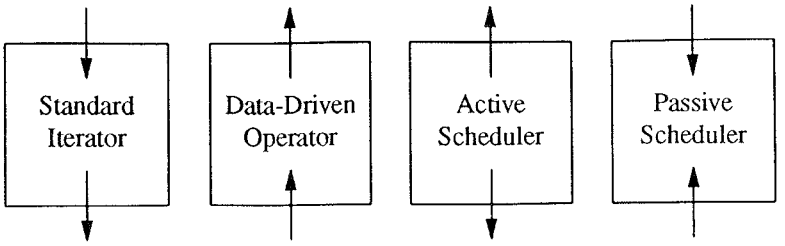


Figure 36. Operators, schedulers, and control flow.

easily determined while saving the result on temporary disk.

In principle, this was done in Ingres' original optimization method called *Decomposition*, except that Ingres performed only one operation at a time before optimizing the remaining query [Wong and Youssefi 1976; Youssefi and Wong 1979]. We propose alternating more slowly between optimization and execution, i.e., to perform a "reasonable" number of steps between optimizations, where reasonable may be three to ten selections and joins depending on errors and error propagation in selectivity estimation. Stopping the data flow and resuming after additional optimization could very well turn out to be the most reliable technique for very large complex queries.

Implementation of this technique could be embodied in another control operator, the *choose-plan* operator first described in Graefe and Ward [1989]. Its current implementation executes zero or more subplans and then invokes a decision function provided by the optimizer that decides which of multiple equivalent plans to execute depending on intermediate result statistics, current system load, and run-time values of query parameters unknown at optimization time. Unfortunately, further research is needed to develop techniques for placing such operators in very complex query plans. One possible purpose of the subplans executed prior to a decision could be to sample the values in the database. A very interesting research direction quantifies the value of sampling by analyzing the resulting improvement in the decision quality [Seppi et al. 1989].

12. ADDITIONAL TECHNIQUES FOR PERFORMANCE IMPROVEMENT

In this section, we consider some additional techniques that have been proposed in the literature or used in real systems and that have not been discussed in earlier sections of this survey. In particular, we consider precomputation, data compression, surrogate processing, bit vector filters, and specialized hardware. Recently proposed techniques that have not been fully developed are not discussed here, e.g., "racing" equivalent plans and terminating the ones that seem not competitive after some small amount of time.

12.1 Precomputation and Derived Data

It is trivial to answer a query for which the answer is already known—therefore, precomputation of frequently requested information is an obvious idea. The problem with keeping preprocessed information in addition to base data is that it is redundant and must be invalidated or maintained on updates to the base data.

Precomputation and derived data such as relational views are duals. Thus, concepts and algorithms designed for one will typically work well for the other. The main difference is the database user's view: precomputed data are typically used after a query optimizer has determined that they can be used to answer a user query against the base data, while derived data are known to the user and can be queried without regard to the fact that they actually must be derived at run-time from stored base data. Not sur-

prisingly, since derived data are likely to be referenced and requested by users and application programs, precomputation of derived data has been investigated both for relational and object-oriented data models.

Indices are the simplest form of precomputed data since they are a redundant and, in a sense, precomputed selection. They represent a compromise between a nonredundant database and one with complex precomputed data because they can be maintained relatively efficiently.

The next more sophisticated form of precomputation are inversions as provided in System R's "0th" prototype [Chamberlin et al. 1981a], view indices as analyzed by Roussopoulos [1991], two-relation *join indices* as proposed by Valduriez [1987], or domain indices as used in the ANDA project (called *VALTREE* there) [Deshpande and Van Gucht 1988] in which all occurrences of one domain (e.g., part number) are indexed together, and each index entry contains a relation identification with each record identifier. With join or domain indices, join queries can be answered very fast, typically faster than using multiple single-relation indices. On the other hand, single-clause selections and updates may be slightly slower if there are more entries for each indexed key.

For binary operators, there is a spectrum of possible levels of precomputations (as suggested by J. A. Blakely), explored predominantly for joins. The simplest form of precomputation in support of binary operations is individual indices, e.g., clustering B-trees that ensure and maintain sorted relations. On the other extreme are completely materialized join results. Intermediate levels are pointer-based joins [Shekita and Carey 1990] (discussed earlier in the section on matching) and join indices [Valduriez 1987]. For each form of precomputed result, the required redundant data structures must be maintained each time the underlying base data are updated, and larger retrieval speedup might be paid for with larger maintenance overhead.

Babb [1982] explored storing only results of outer joins, but not the normalized base relations, in the content-addressable file store (CAFS), and called this encoding join normal form. Blakeley et al. [1989], Blakeley and Martin [1990], Larson and Yang [1985], Medeiros and Tompa [1985], Tompa and Blakeley [1988], and Yang and Larson [1987] investigated storing and maintaining materialized views in relational database systems. Their hope was to speed relational query processing by using derived data, possibly without storing all base data, and ensuring that their maintenance overhead would be less than their benefits in faster query processing. For example, Blakeley and Martin [1990] demonstrated that for a single join there exists a large range of retrieval and update mixes in which materialized views outperform both join indices and hybrid hash join. This investigation should be extended, however, for more complex queries, e.g., joins of three and four inputs, and for queries in object-oriented systems and emerging database applications.

Hanson [1987] compared query modification (i.e., query evaluation from base relations) against the maintenance costs of materialized views and considered in particular the cost of immediate versus deferred updates. His results indicate that for modest update rates, materialized views provide better system performance. Furthermore, for modest selectivities of the view predicate, deferred-view maintenance using differential files [Severance and Lohman 1976] outperforms immediate maintenance of materialized views. However, Hanson also did not include multi-input joins in his study.

Sellis [1987] analyzed caching of results in a query language called Quel+ (which is a subset of Postquel [Stonebraker et al. 1990b]) over a relational database with procedural (QUEL) fields [Sellis 1987]. He also considered the case of limited space on secondary storage used for caching query results, and replacement algorithms for query results in the cache when the space becomes insufficient.

Links between records (pointers of some sort, e.g., record, tuple, or object identifiers) are another form of precomputation. Links are particularly effective for system performance if they are combined with clustering (assignment of records to pages). Database systems for the hierarchical and network models have used physical links and clustering, but supported basically only queries and operations that were "precomputed" in this way. Some researchers tried to overcome this restriction by building relational query engines on top of network systems, e.g., Chen and Kuck [1984], Rosenthal and Reiner [1985], Zaniolo [1979]. However, with performance improvements in the relational world, these efforts seem to have been abandoned. With the advent of extensible and object-oriented database management systems, combining links and ad hoc query processing might become a more interesting topic again. A recent effort for an extensible-relational system are Starburst's pointer-based joins discussed earlier [Haas et al. 1990; Shekita and Carey 1990].

In order to ensure good performance for its extensive rule-processing facilities, Postgres uses precomputation and caching of the action parts of production rules [Stonebraker 1987; Stonebraker et al. 1990a; 1990b]. For automatic maintenance of such derived data, persistent "invalidation locks" are stored for detection of invalid data after updates to the base data.

Finally, the Cactis project focused on maintenance of derived data in object-oriented environments [Hudson and King 1989]. The conclusions of this project include that incremental maintenance coupled with a fairly simple adaptive clustering algorithm is an efficient way to propagate updates to derived data.

One issue that many investigations into materialized views ignore is the fact that many queries do not require views in their entirety. For example, if a relational student information system includes a view that computes each student's grade point average from the enrollment data, most queries using this view will select only a single student, not all students at the school. Thus, if the view definition is merged into the query before query optimization, as discussed in the introduction, only one student's grade point average, not the entire view, will be computed for each query. Obviously, the treatment of this difference will affect an analysis of costs and benefits of materialized views.

12.2 Data Compression

A number of researchers have investigated the effect of compression on database systems and their performance [Graefe and Shapiro 1991; Lynch and Brownrigg 1981; Ruth and Keutzer 1972; Severance 1983]. There are two types of compression in database systems. First, the amount of redundancy can be reduced by prefix and suffix truncation, in particular, in indices, and by use of encoding tables (e.g., color combination "9" means "red car with black interior"). Second, compression schemes can be applied to attribute values, e.g., adaptive Huffman coding or Ziv-Lempel methods [Bell et al. 1989; Lelewer and Hirschberg 1987]. This type of compression can be exploited most effectively in database query processing if all attributes of the same domain use the same encoding, e.g., the "Part-No" attributes of data sets representing parts, orders, shipments, etc., because common encodings permit comparisons without decompression.

Most obviously, compression can reduce the amount of disk space required for a given data set. Disk space savings has a number of ramifications on I/O performance. First, the reduced data space fits into a smaller physical disk area; therefore, the seek distances and seek times are reduced. Second, more data fit into each disk page, track, and cylinder, allowing more intelligent clustering of related objects into physically near locations. Third, the unused disk space can be used for disk shadowing to increase reliability, availability, and I/O performance [Bitton and Gray 1988]. Fourth, compressed data can be trans-

ferred faster to and from disk. In other words, data compression is an effective means to increase disk bandwidth (not by increasing physical transfer rates but by increasing the information density of transferred data) and to relieve the I/O bottleneck found in many high-performance database management systems [Boral and DeWitt 1983]. Fifth, in distributed database systems and in client-server situations, compressed data can be transferred faster across the network than uncompressed data. Uncompressed data require either more network time or a separate compression step. Finally, retaining data in compressed form in the I/O buffer allows more records to remain in the buffer, thus increasing the buffer hit rate and reducing the number of I/Os. The last three points are actually more general. They apply to the entire storage hierarchy of tape, disk, controller caches, local and remote main memories, and CPU caches.

For query processing, compression can be exploited far beyond improved I/O performance because decompression can often be delayed until a relatively small data set is presented to the user or an application program. First, exact-match comparisons can be performed on compressed data. Second, projection and duplicate removal can be performed without decompressing data. The situation for aggregation is a little more complex since the attribute on which arithmetic is performed typically must be decompressed. Third, neither the join attributes nor other attributes need to be decompressed for most joins. Since keys and foreign keys are from the same domain, and if compression schemes are fixed for each domain, a join on compressed key values will give the same results as a join on normal uncompressed key values. It might seem unusual to perform a merge-join in the order of compressed values, but it nonetheless is possible and will produce correct results.

There are a number of benefits from processing compressed data. First, materializing output records is faster because records are shorter, i.e., less copying is required. Second, for inputs larger than memory, more records fit into memory. In hybrid hash join and duplicate removal, for instance, the fraction of the file that can be retained in the hash table and thus be joined without any I/O is larger. During sorting, the number of records in memory and thus per run is larger, leading to fewer runs and possibly fewer merge levels. Third, and very interestingly, skew is less likely to be a problem. The goal of compression is to represent the information with as few bits as possible. Therefore, each bit in the output of a good compression scheme has close to maximal information content, and bit columns seen over the entire file are unlikely to be skewed. Furthermore, bit columns will not be correlated. Thus, the compressed key values can be used to create a hash value distribution that is almost guaranteed to be uniform, i.e., optimal for hashing in memory and partitioning to overflow files as well as to multiple processors in parallel join algorithms.

We believe that data compression is undervalued in current query processing research, mostly because it was not realized that many operations can often be performed faster on compressed data than on uncompressed data, and we hope that future database management systems make extensive use of data compression. Considering the current growth rates in CPU and I/O performance, it might even make sense to exploit data compression on the fly for hash table overflow resolution.

12.3 Surrogate Processing

Another very useful technique in query processing is the use of surrogates for intermediate results. A surrogate is a reference to a data item, be it a logical object identifier (OID) used in object-oriented systems or a physical record identifier (RID) or location. Instead of keeping a complete record in memory, only the fields that are used immediately are kept, and the remainder replaced by

a surrogate, which has in principle the same effect as compression. While this technique has traditionally been used to reduce main-memory requirements, it can also be employed to improve board- and CPU-level caching [Nyberg et al. 1993].

The simplest case in which surrogate processing can be exploited is in avoiding copying. Consider a relational join; when two items satisfy the join predicate, a new tuple is created from the two original ones. Instead of copying the data fields, it is possible to create only a pair of RIDs or pointers to the original records if they are kept in memory. If a record is 50 times larger than an RID, e.g., 8 vs. 400 bytes, the effort spent on copying bytes is reduced by that factor.

Copying is already a major part of the CPU time spent in many query processing systems, but it is becoming more expensive for two reasons. First, many modern CPU designs and implementations are optimized for an impressive number of instructions per second but do not provide the performance improvements in mundane tasks such as moving bytes from one memory location to another [Ousterhout 1990]. Second, many modern computer architectures employ multiple CPUs accessing shared memory over one bus because this design permits fast and inexpensive parallelism. Although alleviated by local caches, bus contention is the major bottleneck and limitation to scalability in shared-memory parallel machines. Therefore, reductions in memory-to-memory copying in database query execution engines permit higher useful degrees of parallelism in shared-memory machines.

A second example for surrogate processing was mentioned earlier in connection with indices. To evaluate a conjunction with multiple clauses, each of which is supported by an index, it might be useful to perform an intersection of RID-lists to reduce the number of records needed before actual data are accessed.

A third case is the use of indices and RIDs to evaluate joins, for example, in the query processing techniques used in Ingres [Kooi 1980; Kooi and Frankforth 1982] and IBM's hybrid join [Cheng et al. 1991] discussed in the section on binary matching.

Surrogate processing has also been used in parallel systems, in particular, distributed-memory implementations, to reduce network traffic. For example, Lorie and Young [1989] used RIDs to reduce the communication time in parallel sorting by sending (sort key, RID) pairs to a central site, which determines each record's global rank, and then repartitioning and merging records very quickly by their rank alone without further data comparisons.

Another form of surrogates are encodings with lossy compressions, such as superimposed coding used for efficient access methods [Bloom 1970; Faloutsos 1985; Sacks-Davis and Ramamohanarao 1983; Sacks-Davis et al. 1987]. Berra et al. [1987] and Chung and Berra [1988] considered indexing and retrieval organizations for very large (relational) knowledge bases and databases. They employed three techniques, concatenated code words (CCWs), superimposed code words (SCWs), and transformed inverted lists (TILs). TILs are normal index structures for all attributes of a relation that permit answering conjunctive queries by bitwise *and*ing. CCWs and SCWs use hash values of all attributes of a tuple and either concatenate such hash values or bitwise *or* them together. The resulting code words are then used as keys in indices. In their particular architecture, Berra et al. and Chung and Berra consider associative memory and optical computing to search efficiently through such indices, although conventional software techniques could be used as well.

12.4 Bit Vector Filtering

In parallel systems, bit vector filters have been used very effectively for what we call here "probabilistic semi-joins." Consider a relational join to be executed on a distributed-memory machine with repartitioning of both input relations on the join attribute. It is clear that communica-

tion effort could be reduced if only the tuples that actually contribute to the join result, i.e., those with a match in the other relation, needed to be shipped across the network. To accomplish this, distributed database systems were designed to make extensive use of semi-joins, e.g., SDD-1 [Bernstein et al. 1981].

A faster alternative to semi-joins, which, as discussed earlier, requires basically the same computational effort as natural joins, is the use of bit vector filters [Babb 1979], also called Bloom-filters [Bloom 1970]. A bit vector filter with N bits is initialized with zeroes; and all items in the first (preferably the smaller) input are hashed on their join key to $0, \ldots, N-1$. For each item, one bit in the bit vector filter is set to one; hash collisions are ignored. After the first join input has been exhausted, the bit vector filter is used to filter the second input. Data items of the second input are hashed on their join key value, and only items for which the bit is set to one can possibly participate in the join. There is some chance for false passes in the case of collisions, i.e., items of the second input pass the bit vector filter although they actually do not participate in the join, but if the bit vector filter is sufficiently large, the number of false passes is very small.

In general, if the number of bits is about twice the number of items in the first input, bit vector filters are very effective. If many more bits are available, the bit vector filter can be split into multiple subvectors, or multiple bits can be set for each item using multiple hash functions, reducing the number of false passes. Babb [1979] analyzed the use of multiple bit vector filters in detail.

The Gamma relational database machine demonstrated the effectiveness of bit vector filtering in relational join processing on distributed-memory hardware [DeWitt et al. 1986; 1988; 1990; Gerber 1986]. When scanning and redistributing the build input of a join, the Gamma machine creates a bit vector filter that is then distributed to the scanning sites of the probe input. Based on the bit vector filter, a large fraction of the probe tuples can often be discarded before incurring network costs. The decision whether to create one bit vector filter for the entire build input or to create a bit vector filter for each of the join sites depends on the space available for bit vector filters and the communication costs for bit arrays.

Mullin [1990] generalized bit vector filtering to sending bit vector filters back and forth between sites. In his words, "the central notion is to send small but optimally information-dense Bloom filters between sites as long as these filters serve to reduce the volume of tuples which need to be transmitted by more than their own size." While this procedure achieves very low communication costs, it ignores the I/O cost at each site if the reduced relations must be scanned from disk in each step. Qadah [1988] discussed a limited form of this idea using only two bit vector filters and augmenting it with bit vector filter compression.

While bit vector filtering is typically used only for joins, it is equally applicable to all other one-to-one match operators, including semi-join, outer join, intersection, union, and difference. For operators that include nonmatching items in their output, e.g., outer joins and unions, part of the result can be obtained before network transfer, based solely on the bit vector filter. For one-to-one match operations other than join, e.g., outer join and union, bit vector filters can also be used, but the algorithm must be modified to ensure that items that do not pass the bit vector filter are properly included in the operation's output stream. For parallel relational division (universal quantification), bit vector filtering can be used on the divisor attributes to eliminate most of the dividend items that do not pertain to any divisor item. Thus, our earlier assessment that universal quantification can be performed as fast as existential quantification (a semi-join of dividend and divisor relations) even extends to special techniques used to boost join performance.

Bit vector filtering can also be exploited in sequential systems. Consider a

merge-join with sort operations on both inputs. If the bit vector filter is built based on the input of the first sort, i.e., the bit vector filter is completed when all data have reached the first sort operator. This bit vector filter can then be used to reduce the input into the second sort operator on the (presumably larger) second input. Depending on how the sort operation is organized into phases, it might even be possible to create a second bit vector filter from the second merge-join input and use it to reduce the first join input while it is being merged.

For sequential hash joins, bit vector filters can be used in two ways. First, they can be used to filter items of the probe input using a bit vector filter created from items of the build input. This use of bit vector filters is analogous to bit vector filter usage in parallel systems and for merge-join. In Rdb/VMS and DB2, bit vector filters are used when intersecting large RID lists obtained from multiple indices on the same table [Antoshenkov 1993; Mohan et al. 1990]. Second, new bit vector filters can be created and used for each partition in each recursion level. In the Volcano query-processing system, the operator implementing hash join, intersection, etc. uses the space used as anchor for each bucket's linked list for a small bit vector filter after the bucket has been spilled to an overflow file. Only those items from the probe input that pass the bit vector filter are written to the probe overflow file. This technique is used in each recursion level of overflow resolution. Thus, during recursive partitioning, relatively small bit vector filters can be used repeatedly and at increasingly finer granularity to remove items from the probe input that do not contribute to the join result. Bit vectors could also be used to remove items from the build input using bit vector filters created from the probe input; however, since the probe input is presumed the larger input and hash collisions in the bit vector filter would make the filter less effective, it may or may not be an effective technique.

With some modifications of the standard algorithm, bit vector filters can also be used in hash-based duplicate removal. Since bit vector filters can only determine safely which item has not been seen yet, but not which item has been seen yet (due to possible hash collisions), bit vector filters cannot be used in the most direct way in hash-based duplicate removal. However, hash-based duplicate removal can be modified to become similar to a hash join or actually a hash-based set intersection. Consider a large file R and a partitioning fan-out F. First, R is partitioned into $F/2$ partitions. For each partition, two files are created; thus, this step uses the entire fan-out to create a total of F files. Within each partition, a bit vector filter is used to determine whether an item belongs into the first or the second file of that partition. If an item is guaranteed to be unique, i.e., there is no earlier item indicated in the bit vector filter, the item is assigned to the first file, and a bit in the bit vector filter is set. Otherwise, the item is assigned into the partition's second file. At the end of this partitioning step, there are F files, half of them guaranteed to be free of duplicate data items. The possible size of the duplicate-free files is limited by the size of the bit vector filters; therefore, this step should use the largest bit vector filters possible. After the first partitioning step, each partition's pair of files is intersected using the duplicate-free file as probe input. Recall that duplicate removal for a join's build input can be accomplished easily and inexpensively while building the in-memory hash table. Remaining duplicates with one copy in the duplicate-free (probe) file and another copy in the other file (the build input) in the hash table are found when the probe input is matched against the hash table. This algorithm performs very well if many output items depend on only one input item and if the bit vectors are quire large. In that case, the duplicate-free partition files are very large, and the smaller partition file with duplicates can be processed very efficiently.

In order to find and exploit a dual in the realm of sorting and merge-join to bit vector filtering in each recursion level of recursive hash join, sorting of multiple

inputs must be divided into individual merge levels. In other words, for a merge-join of inputs R and S, the sort activity should switch back and forth between R and S, level by level, creating and using a new bit vector filter in each merge level. Unfortunately, even with a sophisticated sort implementation that supports this use of bit vector filters in each merge level, recursive hybrid hash join will make more effective use of bit vector filters because the inputs are partitioned, thus reducing the number of distinct values in each partition in each recursion level.

12.5 Specialized Hardware

Specialized hardware was considered by a number of researchers, e.g., in the forms of hardware sorters and logic-per-track selection. A relatively recent survey of database machine research is given by Su [1988]. Most of this research was abandoned after Boral and DeWitt's [1983] influential analysis that compared CPU and I/O speeds and their trends. They concluded that I/O is most likely the bottleneck in future high-performance query execution, not processing. Therefore, they recommended moving from research on custom processors to techniques for overcoming the I/O bottleneck, e.g., by use of parallel readout disks, disk caching and read-ahead, and indexing to reduce the amount of data to be read for a query. Other investigations also came to the conclusion that parallelism is no substitute for effective storage structures and query execution algorithms [DeWitt and Hawthorn 1981; Neches 1984]. An additional very strong argument against custom VLSI processors is that microprocessor speed is currently improving so rapidly that it is likely that, by the time a special hardware component has been designed, fabricated, tested, and integrated into a larger hardware and software system, the next generation of general-purpose CPUs will be available and will be able to execute database functions programmed in a high-level language at the same speed as the specialized hardware component. Furthermore, it is not clear what specialized hardware would be most beneficial to design, in particular, in light of today's directions toward extensible database systems and emerging database application domains. Therefore, we do not favor specialized database hardware modules beyond general-purpose processing, storage, and communication hardware dedicated to executing database software.

SUMMARY AND OUTLOOK

Database management systems provide three essential groups of services. First, they maintain both data and associated metadata in order to make databases self-contained and self-explanatory, at least to some extent, and to provide data independence. Second, they support safe data sharing among multiple users as well as prevention and recovery of failures and data loss. Third, they raise the level of abstraction for data manipulation above the primitive access commands provided by file systems with more or less sophisticated matching and inference mechanisms, commonly called the query language or query-processing facility. We have surveyed execution algorithms and software architectures used in providing this third essential service.

Query processing has been explored extensively in the last 20 years in the context of relational database management systems and is slowly gaining interest in the research community for extensible and object-oriented systems. This is a very encouraging development, because if these new systems have increased modeling power over previous data models and database management systems but cannot execute even simple requests efficiently, they will never gain widespread use and acceptance. Databases will continue to manage massive amounts of data; therefore, efficient query and request execution will continue to represent both an important research direction and an important criterion in investment decisions in the "real world." In other words, new database management systems should provide greater modeling power (this is widely

accepted and intensely pursued), but also competitive or better performance than previous systems. We hope that this survey will contribute to the use of efficient and parallel algorithms for query processing tasks in new database management systems.

A large set of query processing algorithms has been developed for relational systems. Sort- and hash-based techniques have been used for physical-storage design, for associative index structures, for algorithms for unary and binary matching operations such as aggregation, duplicate removal, join, intersection, and division, and for parallel query processing using hash- or range-partitioning. Additional techniques such as precomputation and compression have been shown to provide substantial performance benefits when manipulating large volumes of data. Many of the existing algorithms will continue to be useful for extensible and object-oriented systems, and many can easily be generalized from sets of tuples to more general pattern-matching functions. Some emerging database applications will require new operators, however, both for translation between alternative data representations and for actual data manipulation.

The most promising aspect of current research into database query processing for new application domains is that the concept of a fixed number of parameterized operators, each performing a part of the required data manipulation and each passing an intermediate result to the next operator, is versatile enough to meet the new challenges. This concept permits specification of database queries and requests in a logical algebra as well as concise representation of database programs in a physical algebra. Furthermore, it allows algebraic optimizations of requests, i.e., optimizing transformations of algebra expressions and cost-sensitive translations of logical into physical expressions. Finally, it permits pipelining between operators to exploit parallel computer architectures and partitioning of stored data and intermediate results for most operators, in particular, for operators on sets but also for other bulk types such as arrays, lists, and time series.

We can hope that much of the existing relational technology for query optimization and parallel execution will remain relevant and that research into extensible optimization and parallelization will have a significant impact on future database applications such as scientific data. For database management systems to become acceptable for new application domains, their performance must at least match that of the file systems currently in use. Automatic optimization and parallelization may be crucial contributions to achieving this goal, in addition to the query execution techniques surveyed here.

ACKNOWLEDGMENTS

José A Blakeley, Cathy Brand, Rick Cole, Diane Davison, David Helman, Ann Linville, Bill McKenna, Gail Mitchell, Shengsong Ni, Barb Peters, Leonard Shapiro, the students of "Readings in Database Systems" at the University of Colorado at Boulder (Fall 1991) and "Database Implementation Techniques" at Portland State University (Winter 1993), David Maier's weekly reading group at the Oregon Graduate Institute (Winter 1992), the anonymous referees, and the *Computing Surveys* editors Shamkant Navathe and Dick Muntz gave many valuable comments on earlier drafts of this survey, which have improved the paper very much. This paper is based on research partially supported by the National Science Foundation with grants IRI-8996270, IRI-8912618, IRI-9006348, IRI-9116547, IRI-9119446, and ASC-9217394, ARPA with contract DAAB 07-91-C-Q518, Texas Instruments, Digital Equipment Corp., Intel Supercomputer Systems Division, Sequent Computer Systems, ADP, and the Oregon Advanced Computing Institute (OACIS)

REFERENCES

ADAM, N. R., AND WORTMANN, J. C. 1989. Security-control methods for statistical databases: A comparative study. *ACM Comput. Surv. 21*, 4 (Dec. 1989), 515.

AHN, I., AND SNODGRASS, R. 1988. Partitioned storage for temporal databases. *Inf. Syst. 13*, 4, 369.

ALBERT, J. 1991. Algebraic properties of bag data types. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 211.

ANALYTI, A., AND PRAMANIK, S. 1992. Fast search in main memory databases. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 215.

ANDERSON, D. P., TZOU, S. Y., AND GRAHAM, G. S. 1988. The DASH virtual memory system. Tech. Rep. 88/461, Univ. of California—Berkeley, CS Division, Berkeley, Calif.

ANTOSHENKOV, G. 1993. Dynamic query optimization in Rdb/VMS. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York.

ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, I. L., WADE, B. W., AND WATSON, V. 1976. System R: A relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June), 97.

ASTRAHAN, M. M., SCHKOLNICK, M., AND WHANG, K. Y. 1987. Approximating the number of unique values of an attribute without sorting, *Inf. Syst. 12*, 1, 11.

ATKINSON, M. P., AND BUNEMANN, O. P. 1987. Types and persistence in database programming languages. *ACM Comput. Surv. 19*, 2 (June), 105.

BABB, E. 1982. Joined Normal Form: A storage encoding for relational databases. *ACM Trans. Database Syst. 7*, 4 (Dec.), 588.

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

BAEZA-YATES, R. A., AND LARSON, P. A. 1989. Performance of B+ -trees with partial expansions. *IEEE Trans. Knowledge Data Eng. 1*, 2 (June), 248.

BANCILHON, F., AND RAMAKRISHNAN, R. 1986. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 16.

BARGHOUTI, N. S., AND KAISER, G. E. 1991. Concurrency control in advanced database applications. *ACM Comput. Surv. 23*, 3 (Sept.), 269.

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

BATINI, C., LENZERINI, M., AND NAVATHE, S. B. 1986. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv. 18*, 4 (Dec.), 323.

BATORY, D. S., BARNETT, J. R., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B. C., AND WISE, T. E. 1988a. GENESIS: An extensible database management system, *IEEE Trans. Softw. Eng. 14*, 11 (Nov.), 1711.

BATORY, D. S., LEUNG, T. Y., AND WISE, T. E. 1988b. Implementation concepts for an extensible data model and data language. *ACM Trans. Database Syst. 13*, 3 (Sept.), 231.

BAUGSTO, B., AND GREIPSLAND, J. 1989. Parallel sorting methods for large data volumes on a hypercube database computer. In *Proceedings of the 6th International Workshop on Database Machines* (Deauville, France, June 19–21).

BAYER, R., AND MCCREIGHTON, E. 1972. Organisation and maintenance of large ordered indices. *Acta Informatica 1*, 3, 173.

BECK, M., BITTON, D., AND WILKINSON, W. K. 1988. Sorting large files on a backend multiprocessor. *IEEE Trans. Comput. 37*, 7 (July), 769.

BECKER, B., SIX, H. W., AND WIDMAYER, P. 1991. Spatial priority search: An access technique for scaleless maps. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 128.

BECKMANN, N., KRIEGEL, H. P., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 322.

BELL, T., WITTEN, I. H., AND CLEARY, J. G. 1989. Modelling for text compression. *ACM Comput. Surv. 21*, 4 (Dec.), 557.

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

BERNSTEIN, P. A., AND GOODMAN, N. 1981. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June), 185.

BERNSTEIN, P. A., GOODMAN, N., WONG, E., REEVE, C. L., AND ROTHNIE, J. B. 1981. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec.), 602.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass.

BERRA, P. B., CHUNG, S. M., AND HACHEM, N. I. 1987. Computer architecture for a surrogate file to a very large data/knowledge base. *IEEE Comput. 20*, 3 (Mar), 25.

BERTINO, E. 1991. An indexing technique for object-oriented databases. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 160.

BERTINO, E. 1990. Optimization of queries using nested indices. In *Lecture Notes in Computer Science*, vol. 416. Springer-Verlag, New York.

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

BHIDE, A. 1988. An analysis of three transaction processing architectures. In *Proceedings of the International Conference on Very Large Data Bases* (Los Angeles, Aug.). VLDB Endowment, 339.

BHIDE, A., AND STONEBRAKER, M. 1988. A performance comparison of two architectures for fast transaction processing. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 536.

BITTON, D., AND DEWITT, D. J. 1983. Duplicate record elimination in large data files. *ACM Trans. Database Syst. 8*, 2 (June), 255.

BITTON-FRIEDLAND, D. 1982. Design, analysis, and implementation of parallel external sorting algorithms Ph D. Thesis, Univ. of Wisconsin—Madison.

BITTON, D , AND GRAY, J 1988. Disk shadowing. In *Proceedings of the International Conference on Very Large Data Bases*. (Los Angeles, Aug.). VLDB Endowment, 331

BITTON, D., DEWITT, D J., HSIAO, D. K. AND MENON, J. 1984 A taxonomy of parallel sorting. *ACM Comput. Surv. 16*, 3 (Sept.), 287.

BITTON, D., HANRAHAN, M. B., AND TURBYFILL, C. 1987. Performance of complex queries in main memory database systems. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York.

BLAKELEY, J. A., AND MARTIN, N. L. 1990 Join index, materialized view, and hybrid hash-join: A performance analysis In *Proceedings of the IEEE Conference on Data Engineering* IEEE, New York.

BLAKELEY, J. A , COBURN, N., AND LARSON, P. A. 1989. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans Database Syst 14*, 3 (Sept.), 369

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

BLASGEN, M., AND ESWARAN, K. 1976. On the evaluation of queries in a relational database system IBM Res. Rep RJ 1745, IBM, San Jose, Calif.

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

BORAL, H. 1988. Parallelism in Bubba. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems* (Austin, Tex., Dec.), 68.

BORAL, H., AND DEWITT, D. J. 1983. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Proceedings of the International Workshop on Database Machines*. Reprinted in *Parallel Architectures for Database Systems*. IEEE Computer Society Press, Washington, D.C., 1989.

BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. 1990. Prototyping Bubba, A Highly Parallel Database System. *IEEE Trans. Knowledge Data Eng. 2*, 1 (Mar.), 4.

BRATBERGSENGEN, K. 1984. Hashing methods and relational algebra operations. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 323.

BROWN, K. P., CAREY, M. J., DEWITT, D. J., MEHTA, M., AND NAUGHTON, J. F. 1992. Scheduling issues for complex database workloads. Computer Science Tech. Rep. 1095, Univ. of Wisconsin—Madison.

BUCHERAL, P., THEVERIN, J. M., AND VALDURIEZ, P. 1990. Efficient main memory data management using the DBGraph storage model. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 683.

BUNEMAN, P., AND FRANKEL, R. E. 1979. FQL—A Functional Query Language. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 52.

BUNEMAN, P., FRANKEL, R. E., AND NIKHIL, R. 1982 An implementation technique for database query languages. *ACM Trans. Database Syst. 7*, 2 (June), 164.

CACACE, F., CERI, S., AND HOUTSMA, M A. W. 1992. A survey of parallel execution strategies for transitive closures and logic programs To appear in *Distr. Parall. Databases*.

CAREY, M. J., DEWITT, D. J., RICHARDSON, J. E., AND SHEKITA, E. J. 1986. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 91.

CARLIS, J. V. 1986. HAS: A relational algebra operator, or divided is not enough to conquer. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 254.

CARTER, J. L., AND WEGMAN, M. N. 1979. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2, 143.

CHAMBERLIN, D. D., ASTRAHAN, M M., BLASGEN, M. W., GRAY, J. N., KING, W. F., LINDSAY, B. G., LORIE, R., MEHL, J. W., PRICE, T. G., PUTZOLO, F , SELINGER, P. G., SCHKOLNIK, M., SLUTZ, D. R., TRAIGER, I. L , WADE, B. W., AND YOST, R. A. 1981a. A history and evaluation of System R. *Commun ACM 24*, 10 (Oct.), 632.

CHAMBERLIN, D. D., ASTRAHAN, M. M., KING, W F., LORIE, R. A., MEHL, J. W., PRICE, T. G., SCHKOLNIK, M., SELINGER, P G., SLUTZ, D. R., WADE, B. W., AND YOST, R. A. 1981b. Support for repetitive transactions and ad hoc queries in System R. *ACM Trans. Database Syst. 6*, 1 (Mar), 70.

CHEN, P. P. 1976. The entity relationship model —Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar.), 9.

CHEN, H , AND KUCK, S. M. 1984. Combining relational and network retrieval methods. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 131.

CHEN, M. S., LO, M. L., YU, P. S., AND YOUNG, H C 1992. Using segmented right-deep trees for

the execution of pipelined hash joins. In *Proceedings of the International Conference on Very Large Data Bases* (Vancouver, BC, Canada). VLDB Endowment, 15.

CHENG, J., HADERLE, D., HEDGES, R., IYER, B. R., MESSINGER, T., MOHAN, C., AND WANG, Y. 1991. An efficient hybrid join algorithm: A DB2 prototype. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 171.

CHERITON, D. R., GOOSEN, H. A., AND BOYLE, P. D. 1991 Paradigm: A highly scalable shared-memory multicomputer. *IEEE Comput. 24*, 2 (Feb.), 33.

CHIU, D. M., AND HO, Y. C. 1980. A methodology for interpreting tree queries into optimal semi-join expressions. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 169.

CHOU, H. T. 1985. Buffer management of database systems. Ph.D. thesis, Univ. of Wisconsin—Madison.

CHOU, H. T., AND DEWITT, D. J. 1985. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the International Conference on Very Large Data Bases* (Stockholm, Sweden, Aug.). VLDB Endowment, 127. Reprinted in *Readings in Database Systems*. Morgan-Kaufman, San Mateo, Calif., 1988.

CHRISTODOULAKIS, S. 1984. Implications of certain assumptions in database performance evaluation. *ACM Trans. Database Syst. 9*, 2 (June), 163.

CHUNG, S. M., AND BERRA, P. B. 1988. A comparison of concatenated and superimposed code word surrogate files for very large data/knowledge bases. In *Lecture Notes in Computer Science*, vol. 303. Springer-Verlag, New York, 364.

CLUET, S., DELOBEL, C., LECLUSE, C., AND RICHARD, P. 1989. Reloops, an algebra based query language for an object-oriented database system. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases* (Kyoto, Japan, Dec. 4–6).

COMER, D. 1979. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June), 121.

COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. 1988. Data placement in Bubba. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 99.

DADAM, P., KUESPERT, K., ANDERSON, F., BLANKEN, H., ERBE, R., GUENAUER, J., LUM, V., PISTOR, P., AND WALCH, G. 1986. A database management prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 356.

DANIELS, D., AND NG, P. 1982. Distributed query compilation and processing in R*. *IEEE Database Eng. 5*, 3 (Sept.).

DANIELS, S., GRAEFE, G., KELLER, T., MAIER, D., SCHMIDT, D., AND VANCE, B. 1991. Query optimization in revelation, an overview. *IEEE Database Eng. 14*, 2 (June).

DAVIDSON, S. B., GARCIA-MOLINA, H., AND SKEEN, D. 1985. Consistency in partitioned networks. *ACM Comput. Surv. 17*, 3 (Sept.), 341.

DAVIS, D. D. 1992. Oracle's parallel punch for OLTP. *Datamation* (Aug. 1), 67.

DAVISON, W. 1992. Parallel index building in Informix OnLine 6.0. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 103.

DEPPISCH, U., PAUL, H. B., AND SCHEK, H. J. 1986. A storage system for complex objects. In *Proceedings of the International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept.), 183.

DESHPANDE, V., AND LARSON, P. A. 1992. The design and implementation of a parallel join algorithm for nested relations on shared-memory multiprocessors. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 68.

DESHPANDE, V., AND LARSON, P. A. 1991. An algebra for nested relations with support for nulls and aggregates. Computer Science Dept., Univ. of Waterloo, Waterloo, Ontario, Canada.

DESHPANDE, A., AND VAN GUCHT, D. 1988. An implementation for nested relational databases. In *Proceedings of the International Conference on Very Large Data Bases* (Los Angeles, Calif., Aug.) VLDB Endowment, 76.

DEWITT, D. J. 1991. The Wisconsin benchmark: Past, present, and future. In *Database and Transaction Processing System Performance Handbook*. Morgan-Kaufman, San Mateo, Calif.

DEWITT, D. J., AND GERBER, R. H. 1985. Multiprocessor hash-based join algorithms. In *Proceedings of the International Conference on Very Large Data Bases* (Stockholm, Sweden, Aug.). VLDB Endowment, 151.

DEWITT, D. J., AND GRAY, J. 1992. Parallel database systems: The future of high-performance database systems. *Commun. ACM 35*, 6 (June), 85.

DEWITT, D. J., AND HAWTHORN, P. B. 1981. A performance evaluation of database machine architectures. In *Proceedings of the International Conference on Very Large Data Bases* (Cannes, France, Sept.). VLDB Endowment, 199.

DEWITT, D. J., GERBER, R. H., GRAEFE, G., HEYTENS, M. L., KUMAR, K. B., AND MURALIKRISHNA, M. 1986. GAMMA—A high performance dataflow database machine. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 228. Reprinted in *Readings in Database Systems*. Morgan-Kaufman, San Mateo, Calif., 1988.

DEWITT, D. J., GHANDEHARIZADEH, S., AND SCHNEIDER, D. 1988. A performance analysis of the GAMMA database machine. In *Proceedings of*

ACM SIGMOD Conference. ACM, New York, 350

DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H. I., AND RASMUSSEN, R. 1990. The Gamma database machine project. *IEEE Trans. Knowledge Data Eng. 2*, 1 (Mar.), 44.

DEWITT, D. J., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. 1984. Implementation techniques for main memory database systems In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 1.

DEWITT, D., NAUGHTON, J., AND BURGER, J. 1993. Nested loops revisited. In *Proceedings of Parallel and Distributed Information Systems* (San Diego, Calif., Jan.).

DEWITT, D. J., NAUGHTON, J. E., AND SCHNEIDER, D. A. 1991a. An evaluation of non-equijoin algorithms. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain). VLDB Endowment, 443.

DEWITT, D., NAUGHTON, J., AND SCHNEIDER, D. 1991b Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systems* (Miami Beach, Fla , Dec.)

DOZIER, J. 1992. Access to data in NASA's Earth observing systems. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 1.

EFFELSBERG, W., AND HAERDER, T. 1984. Principles of database buffer management. *ACM Trans. Database Syst. 9*, 4 (Dec), 560.

ENBODY, R. J., AND DU, H. C. 1988. Dynamic hashing schemes *ACM Comput. Surv. 20*, 2 (June), 85.

ENGLERT, S., GRAY, J., KOCHER, R., AND SHAH, P. 1989. A benchmark of nonstop SQL release 2 demonstrating near-linear speedup and scaleup on large databases. Tandem Computers Tech. Rep. 89.4, Tandem Corp., Cupertino, Calif.

EPSTEIN, R. 1979. Techniques for processing of aggregates in relational database systems UCB/ERL Memo. M79/8, Univ. of California, Berkeley, Calif.

EPSTEIN, R., AND STONEBRAKER, M. 1980. Analysis of distributed data base processing strategies. In *Proceedings of the International Conference on Very Large Data Bases* (Montreal, Canada, Oct.). VLDB Endowment, 92.

EPSTEIN, R , STONEBRAKER, M., AND WONG, E. 1978 Distributed query processing in a relational database system. In *Proceedings of ACM SIGMOD Conference*. ACM, New York.

FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. 1979. Extendible hashing: A fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (Sept.), 315.

FALOUTSOS, C 1985. Access methods for text. *ACM Comput. Surv. 17*, 1 (Mar.), 49.

FALOUTSOS, C., NG, R., AND SELLIS, T. 1991. Predictive load control for flexible buffer allocation. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain). VLDB Endowment, 265.

FANG, M. T., LEE, R. C. T., AND CHANG, C. C. 1986. The idea of declustering and its applications. In *Proceedings of the International Conference on Very Large Data Bases* (Kyoto, Japan, Aug.). VLDB Endowment, 181.

FINKEL, R. A., AND BENTLEY, J. L. 1974. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica 4*, 1, 1.

FREYTAG, J. C., AND GOODMAN, N. 1989 On the translation of relational queries into iterative programs. *ACM Trans. Database Syst. 14*, 1 (Mar.), 1.

FUSHIMI, S., KITSUREGAWA, M., AND TANAKA, H. 1986. An overview of the system software of a parallel relational database machine GRACE. In *Proceedings of the International Conference on Very Large Data Bases* (Kyoto, Japan, Aug.). ACM, New York, 209.

GALLAIRE, H., MINKER, J., AND NICOLAS, J M. 1984. Logic and databases A deductive approach. *ACM Comput. Surv. 16*, 2 (June), 153

GERBER, R. H. 1986. Dataflow query processing using multiprocessor hash-partitioned algorithms. Ph.D. thesis, Univ. of Wisconsin—Madison.

GHANDEHARIZADEH, S., AND DEWITT, D. J. 1990. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the International Conference on Very Large Data Bases* (Brisbane, Australia). VLDB Endowment, 481.

GOODMAN, J. R., AND WOEST, P. J 1988. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. Computer Science Tech Rep. 766, Univ. of Wisconsin—Madison

GOUDA, M. G., AND DAYAL, U. 1981. Optimal semijoin schedules for query processing in local distributed database systems. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 164.

GRAEFE, G. 1993a. Volcano, An extensible and parallel dataflow query processing system. *IEEE Trans. Knowledge Data Eng.* To be published.

GRAEFE, G. 1993b. Performance enhancements for hybrid hash join Available as Computer Science Tech. Rep. 606, Univ. of Colorado, Boulder.

GRAEFE, G. 1993c Sort-merge-join: An idea whose time has passed? Revised in Portland State Univ. Computer Science Tech. Rep. 93-4.

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

GRAEFE, G. 1990a. Parallel external sorting in Volcano. Computer Science Tech. Rep. 459, Univ. of Colorado, Boulder.

GRAEFE, G. 1990b. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 102.

GRAEFE, G. 1989. Relational division: Four algorithms and their performance. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 94.

GRAEFE, G., AND COLE, R. L. 1993. Fast algorithms for universal quantification in large databases. Portland State Univ. and Univ. of Colorado at Boulder.

GRAEFE, G., AND DAVISON, D. L. 1993. Encapsulation of parallelism and architecture-independence in extensible database query processing. *IEEE Trans. Softw. Eng. 19*, 7 (July).

GRAEFE, G., AND DEWITT, D. J. 1987. The EXODUS optimizer generator. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 160.

GRAEFE, G., AND MAIER, D. 1988. Query optimization in object-oriented database systems: A prospectus. In *Advances in Object-Oriented Database Systems*, vol. 334. Springer-Verlag, New York, 358.

GRAEFE, G., AND MCKENNA, W. J. 1993. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York.

GRAEFE, G., AND SHAPIRO, L. D. 1991. Data compression and database performance. In *Proceedings of the ACM/IEEE-Computer Science Symposium on Applied Computing*. ACM/IEEE, New York.

GRAEFE, G., AND WARD, K. 1989. Dynamic query evaluation plans. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 358.

GRAEFE, G., AND WOLNIEWICZ, R. H. 1992. Algebraic optimization and parallel execution of computations over scientific databases. In *Proceedings of the Workshop on Metadata Management in Scientific Databases* (Salt Lake City, Utah, Nov. 3–5).

GRAEFE, G., COLE, R. L., DAVISON, D. L., MCKENNA, W. J., AND WOLNIEWICZ, R. H. 1992. Extensible query optimization and parallel execution in Volcano. In *Query Processing for Advanced Database Applications*. Morgan-Kaufman, San Mateo, Calif.

GRAEFE, G., LINVILLE, A., AND SHAPIRO, L. D. 1993. Sort versus hash revisited. *IEEE Trans. Knowledge Data Eng.* To be published.

GRAY, J. 1990. A census of Tandem system availability between 1985 and 1990. Tandem Computers Tech. Rep. 90.1, Tandem Corp., Cupertino, Calif.

GRAY, J., AND PUTZOLO, F. 1987. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 395.

GRAY, J., AND REUTER, A. 1991. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufman, San Mateo, Calif.

GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLO, F., AND TRAIGER, I. 1981. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2 (June), 223.

GRUENWALD, L., AND EICH, M. H. 1991. MMDB reload algorithms. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 397.

GUENTHER, O., AND BILMES, J. 1991. Tree-based access methods for spatial databases: Implementation and performance evaluation. *IEEE Trans. Knowledge Data Eng. 3*, 3 (Sept.), 342.

GUIBAS, L., AND SEDGEWICK, R. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th Symposium on the Foundations of Computer Science*.

GUNADHI, H., AND SEGEV, A. 1991. Query processing algorithms for temporal intersection joins. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 336.

GUNADHI, H., AND SEGEV, A. 1990. A framework for query optimization in temporal databases. In *Proceedings of the 5th International Conference on Statistical and Scientific Database Management*.

GUNTHER, O. 1989. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 598.

GUNTHER, O., AND WONG, E. 1987 A dual space representation for geometric data. In *Proceedings of the International Conference on Very Large Data Bases* (Brighton, England, Aug.). VLDB Endowment, 501.

GUO, M., SU, S. Y. W., AND LAM, H. 1991. An association algebra for processing object-oriented databases. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 23.

GUTTMAN, A. 1984. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 47. Reprinted in *Readings in Database Systems*. Morgan-Kaufman, San Mateo, Calif., 1988.

HAAS, L., CHANG, W., LOHMAN, G., MCPHERSON, J., WILMS, P. F., LAPIS, G., LINDSAY, B., PIRAHESH, H., CAREY, M. J., AND SHEKITA, E. 1990. Starburst mid-flight: As the dust clears. *IEEE Trans. Knowledge Data Eng. 2*, 1 (Mar.), 143.

HAAS, L., FREYTAG, J. C., LOHMAN, G., AND PIRAHESH, H. 1989. Extensible query processing in Starburst. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 377.

HAAS, L. M., SELINGER, P. G., BERTINO, E., DANIELS, D., LINDSAY, B., LOHMAN, G., MASUNAGA, Y., MOHAN, C., NG, P., WILMS, P., AND YOST, R.

1982. R*: A research project on distributed relational database management. IBM Res. Division, San Jose, Calif.

HAERDER, T., AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (Dec.).

HAFEZ, A., AND OZSOYOGLU, G. 1988. Storage structures for nested relations. *IEEE Database Eng. 11*, 3 (Sept.), 31.

HAGMANN, R. B. 1986. An observation on database buffering performance metrics. In *Proceedings of the International Conference on Very Large Data Bases* (Kyoto, Japan, Aug.). VLDB Endowment, 289.

HAMMING, R. W. 1977. *Digital Filters*. Prentice-Hall, Englewood Cliffs, N.J.

HANSON, E. N. 1987. A performance analysis of view materialization strategies. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 440.

HENRICH, A., SIX, H. W., AND WIDMAYER, P. 1989. The LSD tree: Spatial access to multidimensional point and nonpoint objects. In *Proceedings of the International Conference on Very Large Data Bases* (Amsterdam, The Netherlands). VLDB Endowment, 45.

HOEL, E. G., AND SAMET, H. 1992. A qualitative comparison study of data structures for large linear segment databases. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 205.

HONG, W., AND STONEBRAKER, M. 1993. Optimization of parallel query execution plans in XPRS. *Distrib. Parall. Databases 1*, 1 (Jan.), 9.

HONG, W., AND STONEBRAKER, M. 1991. Optimization of parallel query execution plans in XPRS. In *Proceedings of the International Conference on Parallel and Distributed Information Systems* (Miami Beach, Fla., Dec.).

HOU, W. C., AND OZSOYOGLU, G. 1993. Processing time-constrained aggregation queries in CASE-DB. *ACM Trans. Database Syst.* To be published.

HOU, W. C., AND OZSOYOGLU, G. 1991. Statistical estimators for aggregate relational algebra queries. *ACM Trans. Database Syst. 16*, 4 (Dec.), 600.

HOU, W. C., OZSOYOGLU, G., AND DOGDU, E. 1991. Error-constrained COUNT query evaluation in relational databases. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 278.

HSIAO, H. I., AND DEWITT, D. J. 1990. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 456.

HUA, K. A., AND LEE, C. 1991. Handling data skew in multicomputer database computers using partition tuning. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain). VLDB Endowment, 525.

HUA, K. A., AND LEE, C. 1990. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of the International Conference on Very Large Data Bases* (Brisbane, Australia). VLDB Endowment, 493.

HUDSON, S. E., AND KING, R. 1989. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. Database Syst. 14*, 3 (Sept.), 291.

HULL, R., AND KING, R. 1987. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv. 19*, 3 (Sept.), 201.

HUTFLESZ, A., SIX, H. W., AND WIDMAYER, P. 1990. The R-File: An efficient access structure for proximity queries. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 372.

HUTFLESZ, A., SIX, H. W., AND WIDMAYER, P. 1988a. Twin grid files: Space optimizing access schemes. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 183.

HUTFLESZ, A., SIX, H. W., AND WIDMAYER, P. 1988b. The twin grid file: A nearly space optimal index structure. In *Lecture Notes in Computer Science*, vol. 303. Springer-Verlag, New York, 352.

IOANNIDIS, Y. E., AND CHRISTODOULAKIS, S. 1991. On the propagation of errors in the size of join results. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 268.

IYER, B. R., AND DIAS, D. M. 1990. System issues in parallel sorting for database systems. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 246.

JAGADISH, H. V. 1991. A retrieval technique for similar shapes. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 208.

JARKE, M., AND KOCH, J. 1984. Query optimization in database systems. *ACM Comput. Surv. 16*, 2 (June), 111.

JARKE, M., AND VASSILIOU, Y. 1985. A framework for choosing a database query language. *ACM Comput. Surv. 17*, 3 (Sept.), 313.

KATZ, R. H. 1990. Towards a unified framework for version modeling in engineering databases. *ACM Comput. Surv. 22*, 3 (Dec.), 375.

KATZ, R. H., AND WONG, E. 1983. Resolving conflicts in global storage design through replication. *ACM Trans. Database Syst. 8*, 1 (Mar.), 110.

KELLER, T., GRAEFE, G., AND MAIER, D. 1991. Efficient assembly of complex objects. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 148.

KEMPER, A., AND MOERKOTTE G. 1990a. Access support in object bases. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 364.

KEMPER, A., AND MOERKOTTE, G. 1990b. Advanced query processing in object bases using access support relations. In *Proceedings of the International Conference on Very Large Data*

Bases (Brisbane, Australia). VLDB Endowment, 290.

KEMPER, A., AND WALLRATH, M. 1987. An analysis of geometric modeling in database systems. *ACM Comput. Surv. 19*, 1 (Mar.), 47.

KEMPER, A., KILGER, C., AND MOERKOTTE, G. 1991. Function materialization in object bases. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 258.

KERNIGHAN, B. W., AND RITCHIE, D. M. 1978. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J.

KIM, W. 1984. Highly available systems for database applications. *ACM Comput. Surv. 16*, 1 (Mar.), 71.

KIM, W. 1980. A new way to compute the product and join of relations. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 179.

KITSUREGAWA, M., AND OGAWA, Y. 1990. Bucket spreading parallel hash: A new, robust, parallel hash join method for skew in the super database computer (SDC). In *Proceedings of the International Conference on Very Large Data Bases* (Brisbane, Australia). VLDB Endowment, 210.

KITSUREGAWA, M., NAKAYAMA, M., AND TAKAGI, M. 1989a. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proceedings of the International Conference on Very Large Data Bases* (Amsterdam, The Netherlands). VLDB Endowment, 257.

KITSUREGAWA, M., TANAKA, H., AND MOTOOKA, T. 1983. Application of hash to data base machine and its architecture. *New Gener. Comput. 1*, 1, 63.

KITSUREGAWA, M., YANG, W., AND FUSHIMI, S. 1989b. Evaluation of 18-stage pipeline hardware sorter. In *Proceedings of the 6th International Workshop on Database Machines* (Deauville, France, June 19–21).

KLUG, A. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM 29*, 3 (July), 699.

KNAPP, E. 1987. Deadlock detection in distributed databases. *ACM Comput. Surv. 19*, 4 (Dec.), 303.

KNUTH, D. 1973. *The Art of Computer Programming*. Vol. III, *Sorting and Searching*. Addison-Wesley, Reading, Mass.

KOLOVSON, C. P., AND STONEBRAKER M. 1991. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 138.

KOOI, R. P. 1980. The optimization of queries in relational databases. Ph.D. thesis, Case Western Reserve Univ., Cleveland, Ohio.

KOOI, R. P., AND FRANKFORTH, D. 1982. Query optimization in Ingres. *IEEE Database Eng. 5*, 3 (Sept.), 2.

KRIEGEL, H. P., AND SEEGER, B. 1988. PLOP-Hashing: A grid file without directory. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 369.

KRIEGEL, H. P., AND SEEGER, B. 1987. Multidimensional dynamic hashing is very efficient for nonuniform record distributions. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 10.

KRISHNAMURTHY, R., BORAL, H., AND ZANIOLO, C. 1986. Optimization of nonrecursive queries. In *Proceedings of the International Conference on Very Large Data Bases* (Kyoto, Japan, Aug.). VLDB Endowment, 128.

KUESPERT, K., SAAKE, G., AND WEGNER, L. 1989. Duplicate detection and deletion in the extended NF2 data model. In *Proceedings of the 3rd International Conference on the Foundations of Data Organization and Algorithms* (Paris, France, June).

KUMAR, V., AND BURGER, A. 1991. Performance measurement of some main memory database recovery algorithms. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 436.

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

LAKSHMI, M. S., AND YU, P. S. 1988. Effect of skew on join performance in parallel architectures. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems* (Austin, Tex., Dec.), 107.

LANKA, S., AND MAYS, E. 1991. Fully persistent B+-trees. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 426.

LARSON, P. A. 1981. Analysis of index-sequential files with overflow chaining. *ACM Trans. Database Syst. 6*, 4 (Dec.), 671.

LARSON, P., AND YANG, H. 1985. Computing queries from derived relations. In *Proceedings of the International Conference on Very Large Data Bases* (Stockholm, Sweden, Aug.). VLDB Endowment, 259.

LEHMAN, T. J., AND CAREY, M. J. 1986. Query processing in main memory database systems. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 239.

LELEWER, D. A., AND HIRSCHBERG, D. S. 1987. Data compression. *ACM Comput. Surv. 19*, 3 (Sept.), 261.

LEUNG, T. Y. C., AND MUNTZ, R. R. 1992. Temporal query processing and optimization in multiprocessor database machines. In *Proceedings of the International Conference on Very Large Data Bases* (Vancouver, BC, Canada). VLDB Endowment, 383.

LEUNG, T. Y. C., AND MUNTZ, R. R. 1990. Query processing in temporal databases. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 200.

LI, K., AND NAUGHTON, J. 1988. Multiprocessor main memory transaction processing. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems* (Austin, Tex., Dec.), 177.

LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the International Conference on Very Large Data Bases* (Montreal, Canada, Oct.). VLDB Endowment, 212. Reprinted in *Readings in Database Systems*. Morgan-Kaufman, San Mateo, Calif

LITWIN, W., MARK, L., AND ROUSSOPOULOS, N. 1990. Interoperability of multiple autonomous databases. *ACM Comput. Surv. 22*, 3 (Sept.), 267.

LOHMAN, G., MOHAN, C., HAAS, L., DANIELS, D., LINDSAY, B., SELINGER, P., AND WILMS, P. 1985. Query processing in R*. In *Query Processing in Database Systems*. Springer, Berlin, 31.

LOMET, D. 1992. A review of recent work on multi-attribute access methods. *ACM SIGMOD Rec. 21*, 3 (Sept.), 56.

LOMET, D., AND SALZBERG, B 1990a The performance of a multiversion access method. In *Proceedings of ACM SIGMOD Conference* ACM, New York, 353

LOMET, D. B , AND SALZBERG, B. 1990b. The hB-tree A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst. 15*, 4 (Dec.), 625.

LORIE, R. A., AND NILSSON, J. F. 1979. An access specification language for a relational database management system *IBM J. Res. Devel. 23*, 3 (May), 286

LORIE, R. A., AND YOUNG, H. C. 1989. A low communication sort algorithm for a parallel database machine. In *Proceedings of the International Conference on Very Large Data Bases* (Amsterdam, The Netherlands). VLDB Endowment, 125.

LYNCH, C A., AND BROWNRIGG, E. B. 1981. Application of data compression to a large bibliographic data base In *Proceedings of the International Conference on Very Large Data Bases* (Cannes, France, Sept.). VLDB Endowment, 435

LYYTINEN, K. 1987. Different perspectives on information systems: Problems and solutions. *ACM Comput. Surv. 19*, 1 (Mar.), 5.

MACKERT, L. F., AND LOHMAN, G. M. 1989 Index scans using a finite LRU buffer: A validated I/O model. *ACM Trans Database Syst. 14*, 3 (Sept.), 401.

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

MAIER, D., AND STEIN, J. 1986 Indexing in an object-oriented database management. In *Proceedings of the International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif , Sept), 171

MAIER, D., GRAEFE, G., SHAPIRO, L., DANIELS, S., KELLER, T., AND VANCE, B. 1992 Issues in distributed complex object assembly In *Proceedings of the Workshop on Distributed Object Management* (Edmonton, BC, Canada, Aug.).

MANNINO, M. V., CHU, P., AND SAGER, T. 1988. Statistical profile estimation in database systems. *ACM Comput. Surv. 20*, 3 (Sept.).

MCKENZIE, L. E., AND SNODGRASS, R. T. 1991. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Comput. Surv. 23*, 4 (Dec.).

MEDEIROS, C., AND TOMPA, F. 1985. Understanding the implications of view update policies. In *Proceedings of the International Conference on Very Large Data Bases* (Stockholm, Sweden, Aug.). VLDB Endowment, 316.

MENON, J. 1986. A study of sort algorithms for multiprocessor database machines. In *Proceedings of the International Conference on Very Large Data bases* (Kyoto, Japan, Aug) VLDB Endowment, 197

MISHRA, P., AND EICH, M. H. 1992. Join processing in relational databases. *ACM Comput. Surv. 24*, 1 (Mar.), 63

MITSCHANG, B. 1989. Extending the relational algebra to capture complex objects. In *Proceedings of the International Conference on Very Large Data Bases* (Amsterdam, The Netherlands). VLDB Endowment, 297.

MOHAN, C., HADERLE, D., WANG, Y., AND CHENG, J. 1990. Single table access using multiple indexes: Optimization, execution and concurrency control techniques. In *Lecture Notes in Computer Science*, vol. 416. Springer-Verlag, New York, 29.

MOTRO, A. 1989. An access authorization model for relational databases based on algebraic manipulation of view definitions In *Proceedings of the IEEE Conference on Data Engineering* IEEE, New York, 339

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

NAKAYAMA, M., KITSUREGAWA, M., AND TAKAGI, M. 1988. Hash-partitioned join method using dynamic destaging strategy. In *Proceedings of the International Conference on Very Large Data Bases* (Los Angeles, Aug.). VLDB Endowment, 468.

NECHES, P M. 1988. The Ynet: An interconnect structure for a highly concurrent data base computer system. In *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation* (Fairfax, Virginia, Oct.).

NECHES, P M. 1984. Hardware support for advanced data management systems. *IEEE Comput. 17*, 11 (Nov.), 29.

NEUGEBAUER, L. 1991 Optimization and evaluation of database queries including embedded interpolation procedures. In *Proceedings of*

ACM SIGMOD Conference. ACM, New York, 118.

NG, R., FALOUTSOS, C., AND SELLIS, T. 1991. Flexible buffer allocation based on marginal gains. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 387.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst. 9*, 1 (Mar.), 38.

NYBERG, C., BERCLAY, T., CVETANOVIC, Z., GRAY. J., AND LOMET, D. 1993. AlphaSort: A RISC machine sort. Tech. Rep. 93.2. DEC San Francisco Systems Center. Digital Equipment Corp., San Francisco.

OMIECINSKI, E. 1991. Performance analysis of a load balancing relational hash-join algorithm for a shared-memory multiprocessor. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain). VLDB Endowment, 375.

OMIECINSKI, E. 1985. Incremental file reorganization schemes. In *Proceedings of the International Conference on Very Large Data Bases* (Stockholm, Sweden, Aug.). VLDB Endowment, 346.

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

ONO, K., AND LOHMAN, G. M. 1990. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the International Conference on Very Large Data Bases* (Brisbane, Australia). VLDB Endowment, 314.

OUSTERHOUT, J. 1990. Why aren't operating systems getting faster as fast as hardware. In *USENIX Summer Conference* (Anaheim, Calif., June). USENIX.

OZSOYOGLU, Z. M., AND WANG, J. 1992. A keying method for a nested relational database management system. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 438.

OZSOYOGLU, G., OZSOYOGLU, Z. M., AND MATOS, V. 1987. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. Database Syst. 12*, 4 (Dec.), 566.

OZSU, M. T., AND VALDURIEZ, P. 1991a. Distributed database systems: Where are we now. *IEEE Comput. 24*, 8 (Aug.), 68.

OZSU, M. T., AND VALDURIEZ, P. 1991b. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, N.J.

PALMER, M., AND ZDONIK, S. B. 1991. FIDO: A cache that learns to fetch. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain). VLDB Endowment, 255.

PECKHAM, J., AND MARYANSKI, F. 1988. Semantic data models. *ACM Comput. Surv. 20*, 3 (Sept.), 153.

PIRAHESH, H., MOHAN, C., CHENG, J., LIU, T. S., AND SELINGER, P. 1990. Parallelism in relational data base systems: Architectural issues and design approaches. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems* (Dublin, Ireland, July).

QADAH, G. Z. 1988. Filter-based join algorithms on uniprocessor and distributed-memory multiprocessor database machines. In *Lecture Notes in Computer Science*, vol. 303. Springer-Verlag, New York, 388.

REW, R. K., AND DAVIS, G. P. 1990. The Unidata NetCDF: Software for scientific data access. In the *6th International Conference on Interactive Information and Processing Systems for Meterology, Oceanography, and Hydrology* (Anaheim, Calif.).

RICHARDSON, J. E., AND CAREY, M. J. 1987. Programming constructs for database system implementation in EXODUS. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 208.

RICHARDSON, J. P., LU, H., AND MIKKILINENI, K. 1987. Design and evaluation of parallel pipelined join algorithms. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 399.

ROBINSON, J. T. 1981. The K-D-B-Tree: A search structure for large multidimensional dynamic indices. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 10.

ROSENTHAL, A., AND REINER, D. S. 1985. Querying relational views of networks. In *Query Processing in Database Systems*. Springer, Berlin, 109.

ROSENTHAL, A., RICH, C., AND SCHOLL, M. 1991. Reducing duplicate work in relational join(s): A modular approach using nested relations. ETH Tech. Rep., Zurich, Switzerland.

ROTEM, D., AND SEGEV, A. 1987. Physical organization of temporal data. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 547.

ROTH, M. A., KORTH, H. F., AND SILBERSCHATZ, A. 1988. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst. 13*, 4 (Dec.), 389.

ROTHNIE, J. B., BERNSTEIN, P. A., FOX, S., GOODMAN, N., HAMMER, M., LANDERS, T. A., REEVE, C., SHIPMAN, D. W., AND WONG, E. 1980. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar.), 1.

ROUSSOPOULOS, N. 1991. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM Trans. Database Syst. 16*, 3 (Sept.), 535.

ROUSSOPOULOS, N., AND KANG, H. 1991. A pipeline N-way join algorithm based on the

2-way semijoin program. *IEEE Trans Knowledge Data Eng. 3*, 4 (Dec.), 486.

RUTH, S. S., AND KEUTZER, P. J. 1972. Data compression for business files. *Datamation 18* (Sept.), 62.

SAAKE, G., LINNEMANN, V., PISTOR, P., AND WEGNER, L. 1989. Sorting, grouping and duplicate elimination in the advanced information management prototype. In *Proceedings of the International Conference on Very Large Data Bases* VLDB Endowment, 307. Extended version in IBM Sci. Ctr. Heidelberg Tech. Rep. 89.03.008, March 1989.

SACCO, G. 1987. Index access with a finite buffer. In *Proceedings of the International Conference on Very Large Data Bases* (Brighton, England, Aug.) VLDB Endowment, 301.

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

SACCO, G. M., AND SCHKOLNIK, M. 1982. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the International Conference on Very Large Data Bases* (Mexico City, Mexico, Sept.). VLDB Endowment, 257.

SACKS-DAVIS, R., AND RAMAMOHANARAO, K. 1983. A two-level superimposed coding scheme for partial match retrieval. *Inf. Syst. 8*, 4, 273.

SACKS-DAVIS, R., KENT, A., AND RAMAMOHANARAO, K. 1987. Multikey access methods based on superimposed coding techniques. *ACM Trans Database Syst. 12*, 4 (Dec.), 655.

SALZBERG, B. 1990. Merging sorted runs using large main memory. *Acta Informatica 27*, 195.

SALZBERG, B. 1988. *File Structures: An Analytic Approach*. Prentice-Hall, Englewood Cliffs, N.J.

SALZBERG, B., TSUKERMAN, A., GRAY, J., STEWART, M., UREN, S., AND VAUGHAN, B. 1990. FastSort: A distributed single-input single-output external sort. In *Proceedings of ACM SIGMOD Conference* ACM, New York, 94.

SAMET, H. 1984. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2 (June), 187.

SCHEK, H. J., AND SCHOLL, M. H. 1986. The relational model with relation-valued attributes. *Inf. Syst. 11*, 2, 137.

SCHNEIDER, D. A. 1991. Bit filtering and multiway join query processing. Hewlett-Packard Labs, Palo Alto, Calif. Unpublished Ms.

SCHNEIDER, D. A. 1990. Complex query processing in multiprocessor database machines. Ph.D. thesis, Univ. of Wisconsin—Madison.

SCHNEIDER, D. A., AND DEWITT, D. J. 1990. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the International Conference on Very Large Data Bases* (Brisbane, Australia). VLDB Endowment, 469.

SCHNEIDER, D., AND DEWITT, D. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of ACM SIGMOD Conference* ACM, New York, 110.

SCHOLL, M. H. 1988. The nested relational model—Efficient support for a relational database interface. Ph.D. thesis, Technical Univ. Darmstadt. In German.

SCHOLL, M., PAUL, H. B., AND SCHEK, H. J. 1987. Supporting flat relations by a nested relational kernel. In *Proceedings of the International Conference on Very Large Data Bases* (Brighton, England, Aug.) VLDB Endowment, 137.

SEEGER, B., AND LARSON, P. A. 1991. Multi-disk B-trees. In *Proceedings of ACM SIGMOD Conference* ACM, New York, 436.

SEGEV, A., AND GUNADHI, H. 1989. Event-join optimization in temporal relational databases. In *Proceedings of the International Conference on Very Large Data Bases* (Amsterdam, The Netherlands). VLDB Endowment, 205.

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 system. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 23. Reprinted in *Readings in Database Systems* Morgan-Kaufman, San Mateo, Calif., 1988.

SELLIS, T. K. 1987. Efficiently supporting procedures in relational database systems. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 278.

SEPPI, K., BARNES, J., AND MORRIS, C. 1989. A Bayesian approach to query optimization in large scale data bases. The Univ. of Texas at Austin ORP 89-19, Austin.

SERLIN, O. 1991. The TPC benchmarks. In *Database and Transaction Processing System Performance Handbook*. Morgan-Kaufman, San Mateo, Calif.

SESHADRI, S., AND NAUGHTON, J. F. 1992. Sampling issues in parallel database systems. In *Proceedings of the International Conference on Extending Database Technology* (Vienna, Austria, Mar.).

SEVERANCE, D. G. 1983. A practitioner's guide to data base compression. *Inf. Syst. 8*, 1, 51.

SEVERANCE, D., AND LOHMAN, G. 1976. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst. 1*, 3 (Sept.).

SEVERANCE, C., PRAMANIK, S., AND WOLBERG, P. 1990. Distributed linear hashing and parallel projection in main memory databases. In *Proceedings of the International Conference on Very Large Data Bases* (Brisbane, Australia) VLDB Endowment, 674.

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

SHAW, G. M., AND ZDONIK, S. B. 1990. A query

algebra for object-oriented databases. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 154.

SHAW, G., AND ZDONIK, S. 1989a. An object-oriented query algebra. *IEEE Database Eng. 12*, 3 (Sept.), 29.

SHAW, G. M., AND ZDONIK, S. B. 1989b. An object-oriented query algebra. In *Proceedings of the 2nd International Workshop on Database Programming Languages*. Morgan-Kaufmann, San Mateo, Calif., 103.

SHEKITA, E. J., AND CAREY, M. J. 1990. A performance evaluation of pointer-based joins. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 300.

SHERMAN, S. W., AND BRICE, R. S. 1976. Performance of a database manager in a virtual memory system. *ACM Trans. Database Syst. 1*, 4 (Dec.), 317.

SHETH, A. P., AND LARSON, J. A. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv. 22*, 3 (Sept.), 183.

SHIPMAN, D. W. 1981. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (Mar.), 140.

SIKELER, A. 1988. VAR-PAGE-LRU: A buffer replacement algorithm supporting different page sizes. In *Lecture Notes in Computer Science*, vol. 303. Springer-Verlag, New York, 336.

SILBERSCHATZ, A., STONEBRAKER, M., AND ULLMAN, J. 1991. Database systems: Achievements and opportunities. *Commun. ACM 34*, 10 (Oct.), 110.

SIX, H. W., AND WIDMAYER, P. 1988. Spatial searching in geometric databases. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 496.

SMITH, J. M., AND CHANG, P. Y. T. 1975. Optimizing the performance of a relational algebra database interface. *Commun. ACM 18*, 10 (Oct.), 568.

SNODGRASS, R. 1990. Temporal databases: Status and research directions. *ACM SIGMOD Rec. 19*, 4 (Dec.), 83.

SOCKUT, G. H., AND GOLDBERG, R. P. 1979. Database reorganization—Principles and practice. *ACM Comput. Surv. 11*, 4 (Dec.), 371.

SRINIVASAN, V., AND CAREY, M. J. 1992. Performance of on-line index construction algorithms. In *Proceedings of the International Conference on Extending Database Technology* (Vienna, Austria, Mar.).

SRINIVASAN, V., AND CAREY, M. J. 1991. Performance of B-tree concurrency control algorithms. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 416.

STAMOS, J. W., AND YOUNG, H. C. 1989. A symmetric fragment and replicate algorithm for distributed joins. Tech. Rep. RJ7188, IBM Research Labs, San Jose, Calif.

STONEBRAKER, M. 1991. Managing persistent objects in a multi-level store. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 2.

STONEBRAKER, M. 1987. The design of the POSTGRES storage system. In *Proceedings of the International Conference on Very Large Data Bases* (Brighton, England, Aug.). VLDB Endowment, 289. Reprinted in *Readings in Database Systems*. Morgan-Kaufman, San Mateo, Calif., 1988.

STONEBRAKER, M. 1986a. The case for shared-nothing. *IEEE Database Eng. 9*, 1 (Mar.).

STONEBRAKER, M. 1986b. The design and implementation of distributed INGRES. In *The INGRES Papers*. Addison-Wesley, Reading, Mass., 187.

STONEBRAKER, M. 1981. Operating system support for database management. *Commun. ACM 24*, 7 (July), 412.

STONEBRAKER, M. 1975. Implementation of integrity constraints and views by query modification. In *Proceedings of ACM SIGMOD Conference* ACM, New York.

STONEBRAKER, M., AOKI, P., AND SELTZER, M. 1988a. Parallelism in XPRS. UCB/ERL Memorandum M89/16, Univ. of California, Berkeley.

STONEBRAKER, M., JHINGRAN, A., GOH, J., AND POTAMIANOS, S. 1990a. On rules, procedures, caching and views in data base systems In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 281

STONEBRAKER, M., KATZ, R., PATTERSON, D., AND OUSTERHOUT, J. 1988b. The design of XPRS. In *Proceedings of the International Conference on Very Large Data Bases* (Los Angeles, Aug.). VLDB Endowment, 318.

STONEBRAKER, M., ROWE, L. A., AND HIROHAMA, M. 1990b. The implementation of Postgres. *IEEE Trans. Knowledge Data Eng. 2*, 1 (Mar.), 125.

STRAUBE, D. D., AND OZSU, M. T. 1989. Query transformation rules for an object algebra. Dept. of Computing Sciences Tech. Rep. 89-23, Univ. of Alberta, Alberta, Canada.

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

TANSEL, A. U., AND GARNETT, L. 1992. On Roth, Korth, and Silberschatz's extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst. 17*, 2 (June), 374.

TEOROY, T. J., YANG, D., AND FRY, J. P. 1986. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv. 18*, 2 (June), 197.

TERADATA. 1983. *DBC/1012 Data Base Computer, Concepts and Facilities*. Teradata Corporation, Los Angeles.

THOMAS, G., THOMPSON, G. R., CHUNG, C. W., BARKMEYER, E., CARTER, F., TEMPLETON, M.,

FOX, S., AND HARTMAN, B. 1990. Heterogeneous distributed database systems for production use. *ACM Comput. Surv. 22*, 3 (Sept.), 237.

TOMPA, F. W., AND BLAKELEY, J A. 1988. Maintaining materialized views without accessing base data. *Inf Syst. 13*, 4, 393.

TRAIGER, I. L. 1982. Virtual memory management for data base systems *ACM Oper. Syst. Rev. 16*, 4 (Oct.), 26.

TRIAGER, I. L., GRAY, J., GALTIERI, C A., AND LINDSAY, B. G. 1982. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst. 7*, 3 (Sept.), 323.

TSUR, S., AND ZANIOLO, C. 1984 An implementation of GEM—Supporting a semantic data model on relational back-end. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 286.

TUKEY, J. W. 1977. *Exploratory Data Analysis*. Addison-Wesley, Reading, Mass.

UNIDATA 1991. *NetCDF User's Guide, An Interface for Data Access, Version I.II.* NCAR Tech Note TS-334 + 1A, Boulder, Colo

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

VANDENBERG, S. L., AND DEWITT, D. J. 1991. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 158.

WALTON, C B. 1989 Investigating skew and scalability in parallel joins. Computer Science Tech. Rep. 89-39, Univ. of Texas, Austin.

WALTON, C. B., DALE, A. G., AND JENEVEIN, R. M. 1991. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain). VLDB Endowment, 537.

WHANG, K. Y., AND KRISHNAMURTHY, R. 1990. Query optimization in a memory-resident domain relational calculus database system. *ACM Trans. Database Syst. 15*, 1 (Mar.), 67.

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*. Springer, Berlin, 297.

WHANG, K. Y., WIEDERHOLD, G., AND SAGLOWICZ, D. 1984. Separability—An approach to physical database design. *IEEE Trans. Comput. 33*, 3 (Mar.), 209.

WILLIAMS, P., DANIELS, D., HAAS, L., LAPIS, G., LINDSAY, B., NG, P., OBERMARCK, R., SELINGER, P., WALKER, A., WILMS, P., AND YOST, R. 1982 R*: An overview of the architecture. In *Improving Database Usability and Responsiveness*. Academic Press, New York. Reprinted in *Readings in Database Systems*. Morgan-Kaufman, San Mateo, Calif., 1988

WILSCHUT, A. N. 1993. Parallel query execution in a main memory database system. Ph.D. thesis, Univ. of Tweuk, The Netherlands.

WILSCHUT, A. N., AND APERS, P. M. G. 1993. Dataflow query execution in a parallel main-memory environment. *Distrib. Parall. Databases 1*, 1 (Jan.), 103.

WOLF, J. L , DIAS, D. M , AND YU, P. S. 1990. An effective algorithm for parallelizing sort merge in the presence of data skew In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems* (Dublin, Ireland, July)

WOLF, J. L., DIAS, D M., YU, P. S , AND TUREK, J. 1991. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, New York, 200

WOLNIEWICZ, R. H., AND GRAEFE, G. 1993 Algebraic optimization of computations over scientific databases. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment.

WONG, E., AND KATZ, R. H. 1983. Distributing a database for parallelism. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 23.

WONG, E., AND YOUSSEFI, K. 1976 Decomposition —A strategy for query processing. *ACM Trans Database Syst 1*, 3 (Sept.), 223.

YANG, H., AND LARSON, P. A. 1987. Query transformation for PSJ-queries. In *Proceedings of the International Conference on Very Large Data Bases* (Brighton, England, Aug.) VLDB Endowment, 245.

YOUSSEFI, K , AND WONG, E 1979. Query processing in a relational database management system. In *Proceedings of the International Conference on Very Large Data Bases* (Rio de Janeiro, Oct). VLDB Endowment, 409.

YU, C. T., AND CHANG, C. C. 1984. Distributed query processing. *ACM Comput. Surv. 16*, 4 (Dec.), 399.

YU, L., AND OSBORN, S. L. 1991. An evaluation framework for algebraic object-oriented query models. In *Proceedings of the IEEE Conference on Data Engineering*. VLDB Endowment, 670.

ZANIOLO, C. 1983 The database language Gem. In *Proceedings of ACM SIGMOD Conference*. ACM, New York, 207. Reprinted in *Readings in Database Systems*. Morgan-Kaufman, San Mateo, Calif., 1988.

ZANIOLO, C. 1979 Design of relational views over network schemas. In *Proceedings of ACM SIGMOD Conference* ACM, New York, 179

ZELLER, H. 1990. Parallel query execution in NonStop SQL. In *Digest of Papers, 35th CompCon Conference*. San Francisco.

ZELLER H., AND GRAY, J. 1990 An adaptive hash join algorithm for multiuser environments. In *Proceedings of the International Conference on Very Large Data Bases* (Brisbane, Australia). VLDB Endowment, 186.

Received January 1992, final revision accepted February 1993