Query Evaluation Techniques for Large Databases

Goetz Graefe
Portland State University, Computer Science Department
P.O. Box 751, Portland, Oregon 97207−0751
(503) 725−5544, graefe@cs.pdx.edu

Based on a survey with the same title published in
ACM Computing Surveys 25(2), June 1993, p. 73-170.
Revision of November 10, 1993.

Abstract

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 it: 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 post-relational 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, retrieval, searching, sorting; H.2.4 [Database Management]: query processing, query evaluation algorithms, parallelism, systems architecture.

General Terms: Software, architecture, algorithms, parallelism, performance.

Additional Keywords and Phrases: Relational, Extensible, and Object-Oriented Database Systems; Iterators; Complex Query Evaluation Plans; Set Matching Algorithms; Sort-Hash Duality; Dynamic Query Evaluation Plans; Operator Model of Parallelization; Parallel Algorithms; Emerging Database Application Domains.

Acknowledgements

José A. Blakeley, Cathy Brand, Rick Cole, Diane Davison, David Helman, Ann Linville, Bill McKenna, Gail Mitchell, Shengsong Ni, Barb Peters, Leonard D. 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 Sham 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).

1. 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 volumes 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 non-traditional database application domains and extensible database management systems 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, Stonebraker, and Ullman 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 vs. hashing, parallelism, resource allocation and scheduling issues in complex queries, special

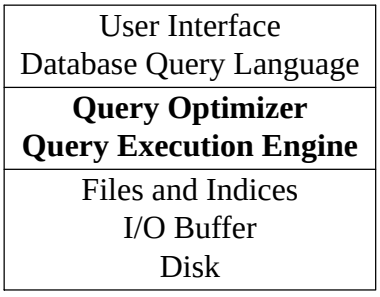


Figure 1. Query Processing in a Database System.

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.

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), the time-space-product of locked database items and their lock duration, 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

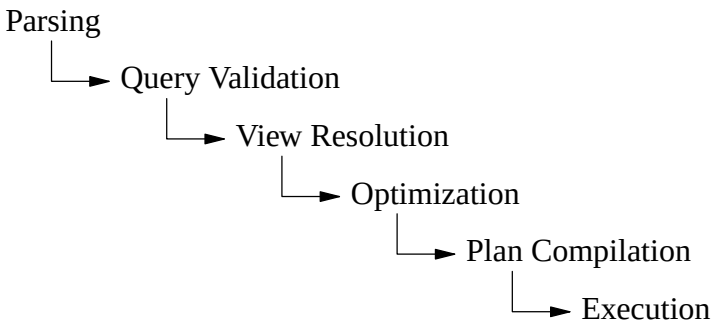


Figure 2. Query Processing Steps.

internal form. Next, the query is validated against the meta-data (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 except in a brief section toward the end; 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 system's query execution engine; the result of this translation can be compiled machine code or a semi-compiled 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 which database objects are to be modified. Standard query optimization and execution techniques apply to this search; the actual update procedure can be either applied 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, Hadzilacos, and Goodman 1987; Gray and Reuter 1991; 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 requires 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]. Query processing, on the other hand, the subject of this survey, focuses on extracting information from a large amount of data without actually changing the database. For example, printing reports for each branch office with

---

[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.

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 [Bancilhon and Ramakrishnan 1986; Cacace, Ceri, and Houtsma 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, Minker, and Nicolas 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, Witten, and Cleary 1989; Lelewer and Hirschberg 1987], distributed and heterogeneous systems [Batini, Lenzerini, and Navathe 1986; Litwin, Mark, and Roussopoulos 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, Garcia-Molina, and Skeen 1985; Kim 1984], query optimization [Jarke and Koch 1984; Mannino, Chu, and Sager 1988; Yu and Chang 1984] and a variety of other database-related topics [Adam and Wortmann 1989; Atkinson and Buneman 1987; Katz 1990; Kemper and Wallrath 1987; Lyytinen 1987; Teoroy, Yang, and Fry 1986]. Bitton et al. have discussed a number of parallel sorting techniques [Bitton et al. 1984], only a few of which are really used in database systems. Mishra and Eich's recent survey of relational join algorithms [Mishra and Eich 1992] compares their behavior using diagrams derived from one by Kitsuregawa et al. [Kitsuregawa, Tanaka, and Motooka 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 non-traditional data models, and auxiliary techniques such as compression.

Section 2 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 3. Section 4 focuses on accessing large data sets on disk. Section 5 begins the discussion of actual data manipulation methods with algorithms for aggregation and duplicate removal, continued in Section 6 with binary matching operations such as join and intersection and in Section 7 with operations for universal quantification. Section 8 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 non-trivial plan shapes is discussed in Section 9. Section 10 is devoted to mechanisms for parallel execution, including architectural issues and load balancing, and Section 11 discusses specific parallel algorithms. Section 12 outlines some non-standard operators for emerging database applications such as statistical and scientific database management systems. Section 13 is a potpourri of additional techniques that enhance the performance of many algorithms, e.g., compression, precomputation, and specialized hardware. To put the topics of this survey into a larger perspective, the relationship between query execution and query optimization is considered again in Section 14 and translated into some general guidelines for database tuning in Section 15. Section 16 outlines current directions in query processing research and development. 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.

2. 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 techniques 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.

---

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

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 Section 6 on binary matching operators and algorithms.) Moreover, the physical algebra may include utility operations such as database loading and unloading, index creation, consistency checking (for physical storage structures as well as for logical integrity constraints), statistics gathering (for query optimization), log and data back-up and restore operations, and data replication. Implementing these operations as operators of the physical algebra makes the query execution infrastructure available to these utilities, in particular mechanisms for pipelining and for parallelism. (Parallelism is discussed in detail later, in Section 10 and Section 11.)

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 algorithms choices and because logical and physical operators frequently do not map directly into one another, as shown in the following 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. Concretely, 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. For example, 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,

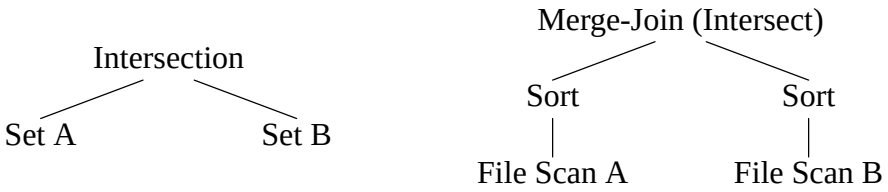


Figure 3. Logical and Physical Algebra Expressions.

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, inter-object reference, etc. Each logical type constructor is complemented by 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.

The mapping process is guided by the meta-data, i.e., the schema information that describes the data in the database. The meta-data are held in catalog files and usually describe the data on the logical and the physical levels rather than the mapping per se, because the mapping options are typically quite obvious once both the logical level and the physical level are described. Once the mapping from logical query to physical plan is complete, the query evaluation plan should not require any further schema information. In other words, the query evaluation plan should be completely bound in order to permit the fastest possible execution. The only issue verified at plan start-up time should be the existence and state of relevant database object.

If a query plan is bound at compile-time, there might be changes in the database (logical or physical) between compile-time and run-time that makes a particular mapping incorrect or suboptimal. For example, if an index is dropped, a query evaluation plan that uses that index is incorrect. Inversely, if an index is created, a query evaluation plan may become suboptimal because a faster, index-based plan has become feasible. To identify plans that have become infeasible since compile-time, query plan start-up typically includes a validation step to ensure that all database objects required for the plan do indeed still exist [Chamberlin et al. 1981b]. Identifying plans that have become suboptimal is much harder; some systems use approximations based on version numbers of schema information.

Synchronization and data transfer between operators are the other main issues 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 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 [Freytag and Goodman 1989]. 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 Section 6).

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 procedure call, much cheaper than inter-process 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. 1988; Batory, Leung, and Wise 1988], and Starburst [Haas et al. 1989; Haas et al. 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 house-keeping. 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

---

[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 iteration granule except for the inputs of merge-join (see later in Section 6), for which it uses "value packets," i.e., groups of records with equal join attribute values.

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

Table 1. Examples of Iterator Functions.

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 non-trivial 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 10.

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, Boral, and Zaniolo 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 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 is 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

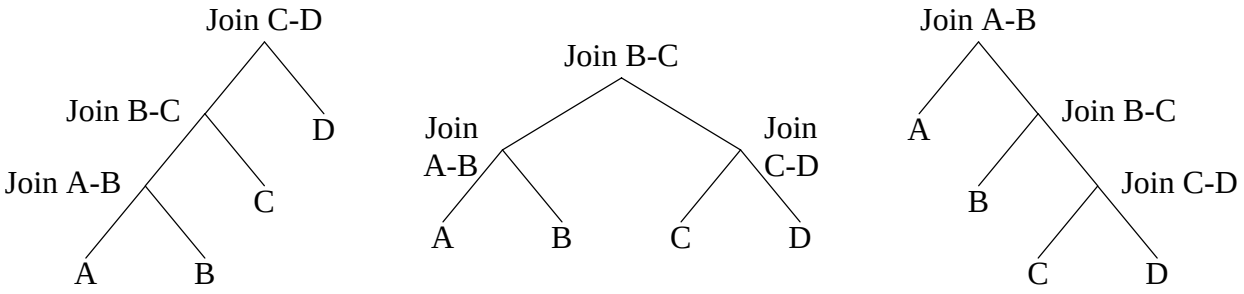


Figure 4. Left-Deep, Bushy, and Right-Deep Plans.

the optimal plan for some queries[4] [Selinger et al. 1979]. 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; DeWitt et al. 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 1993c]. 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].

In addition to the minimal iterator interface, the *open*, *next*, and *close* procedures, most systems can benefit from some further iterator functions. We only discuss them briefly here because we will come back to them later at appropriate points. These functions should be included in the iterator interface because they are tied to physical algorithms in the sense that a new one must be written each time an algorithm is added to the query execution engine's repertoire. First, associated with each physical algorithm is a *cost function*. This function is required by the query optimizer for plan comparisons, but it can also be used very effectively during query plan activation if the query execution engine permits run-time decisions among multiple alternative plans (e.g., file scan and index scan). Second, a subplan must sometimes be executed many times, each time with a different set of bindings for some parameters. Two typical examples are nested subqueries in SQL and nested loops joins. It typically is not optimal to *close* and then re-*open* the subplan; instead, a *re-bind* operation could be included in the iterator interface to accomodate these cases. For example, *re-bind*ing an index scan sets new start- and stop-points, while *re-bind*ing a file scan returns to the start of the file. The operation required to *re-bind* operations other than scans depends on the semantics of the operation, the algorithm used, and the parameters being re-bound. Third, in order to permit linearizing a plan into a data structure that can be stored on disk or shipped across a network, two iterator functions *pack* and *unpack* could be included in the standard iterator interface. Storing a plan on disk is required if a plan is to be used repetitively over an extended period of time, and shipping a plan across a network is required for parallel query evaluation on distributed-memory architectures.

Having discussed synchronization of operators in detail, let us consider mechanisms for data transfer between a producer and a consumer iterator. Each call to a *next* procedure produces one data granule, which is typically one record or object. There are three means where this data

---

[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."

item can actually reside, each of them has been used in some real system. First, a fixed data transfer area can be associated with each producer-consumer pair; the producer's *next* procedure copies a data item into that memory location, and the consumer empties it. This method has the obvious drawbacks that it works only for fixed maximal record sizes and that it requires excessive amount of copying, which is particularly undesirable in shared-memory parallel machines. Nonetheless, this method is used in several commercial products. Second, the data item can reside in temporary space allocated from the heap by either the producer or the consumer. However, memory allocation and deallocation is very expensive in systems that support user aborts without terminating the program, because all allocated memory chunks must be kept track of so they can be identified and deallocated in an abort. Third, each *next* call produces a pointer into a global buffer area. If that buffer area is actually the system's I/O buffer, copying is required only when a new file is created, e.g., a run file in external sorting or a partitioning file in hash-based query processing algorithms. Moreover, work space used for in-memory sorting and for hash tables can actually be part of the I/O buffer, thus making a division of memory into I/O buffer and work space unnecessary. This method is used, for example, in the Volcano query evaluation system [Graefe 1993c].

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 algorithms and 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 database 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

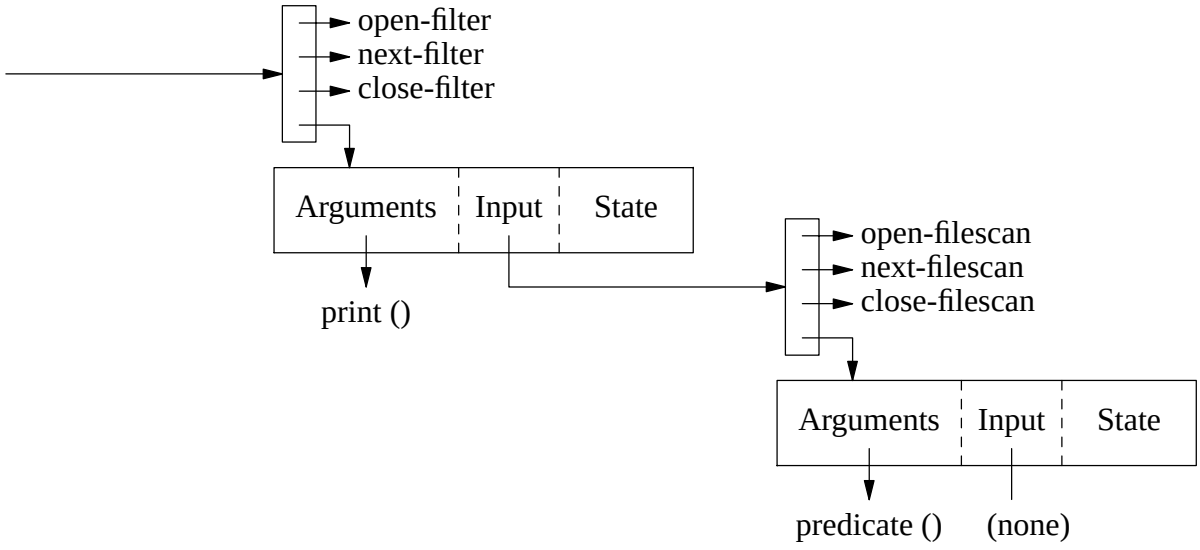


Figure 5. Two Operators in a Volcano Query Plan.

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 pointers 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 compiled-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 the each invocation of *next* and the invocation of *close*. Thus, state records do not reside on the stack but in heap space.

For each iterator, there is a specific state records type. This type is different from other iterator's state records as it contains iterator-specific arguments and local variables (state) while the iterator is suspended, e.g., currently not active between invocations of the operators *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 selected records from a file. The purpose and capabilities of the *filter* operator in Volcano includes 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. 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.

3. 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 for such "value-based" operations, one using pre-computed data structures such as indices and the other using large amounts of memory. The memory-based algorithms can be divided further into those based on sorting and those based on hashing. This pair, sorting and hashing, permeates many aspects of query processing, from indexing and clustering over aggregation and join algorithms to methods for parallelizing database operations. Therefore, we first discuss these approaches first in general terms, without regard to specific algorithms. The subsequent sections survey specific algorithms for unary (grouping, aggregation, duplicate removal) and binary (join, semi-join, intersection, division, etc.) matching, followed by a review of the duality between sort- and hash-based query processing algorithms.

Both sort- and hash-based query processing algorithms are memory-intensive. Their memory allocation should exceed the size of their input data for best performance. Moreover, the allocated memory should be physical memory, not virtual memory that can be swapped out by the operating system. While virtual memory is heavily relied on in most computer systems because it offers an easy way to run algorithms on large inputs, it typically does not provide optimal database performance. All memory-intensive algorithms have variants that use temporary disk files to cope with limited amounts of memory, i.e., less memory than the size of the input data. Therefore, a choice must be made for large inputs whether to run an algorithm's in-memory version in virtual memory or to run an algorithm's version that uses explicit I/O. Since operating systems and virtual memory implementations are general-purpose software that must function reasonably well for all applications and memory access patterns, not only for database query processing, algorithm variants using explicit I/O operations on temporary files typically outperform their in-memory equivalents using virtual memory. One of the specific reasons is that explicit I/O can be planned, which permits moving larger amounts of useful data in a single operation between memory and disk and enables read-ahead for overlap of CPU processing and I/O. Virtual memory can write data to disk asynchronously, but it must rely on page faults to detect which pages to read back.

The needs of efficient algorithms for very large inputs and the virtual-memory abstraction provided by operating systems are a typical example for the discrepancy between system services required by a database system and those offered by a general-purpose operating system. Other examples include processor scheduling, synchronization primitives, file structures, and protection mechanisms for files and address spaces. A number of papers have been written about this interface; suffice it here to point to the most well-known one [Stonebraker 1981].

3.1. 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 merging, 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 [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; Graefe 1993c]. 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 upon in the section on processing surrogates.

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 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 sub-algorithms, 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 sub-algorithms. 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.

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)

Table 2. A Taxonomy of Database Sorting Algorithms.

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)

Table 3. Variables, Their Meaning and Units.

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 for cost calculations in this survey.) Second, runs can be produced using replacement selection. 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 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 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 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 was provided by Knuth [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, in particular for shared-memory parallel machines) 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 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 $\lfloor M / (2 \times C) - 2 \rfloor \le F \le \lfloor M / C - 1 \rfloor$. 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 controller. 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

---

[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 exists in this context.

$F = \lfloor\ 160 / 16\ \rfloor - 3 = 7$. 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, i.e., 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.

Let us explain this idea with the example shown in Figure 6. Consider a sort with a maximal fan-in $F = 5$ and an input file that requires $W = 10$ initial runs. Instead of merging only runs of the same level, merging is delayed until the end of the input has been reached and all run

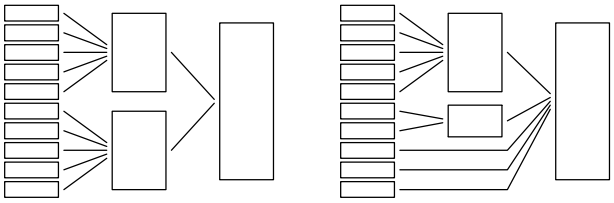


Figure 6. Naive and Optimized Merging.

files have been created. In the first merge step, only 2 of the 10 runs are combined. The second merge step uses the maximal fan-in, combining 5 runs into one. Finally, the five remaining runs (which are of three different sizes), are merged to form the entire sorted output file. The I/O cost (measured by the number of initial runs that must be written to any of the runs created) for the first strategy is $2 \times 10 = 20$, while for the second strategy, it is $2 + 5 + 10 = 17$. Thus, the first strategy requires about 20% more I/O to temporary files than the second one. While 20% savings are significant, even larger savings can be achieved, depending on the relative sizes of the input and of memory. In the comparison shown in Figure 7, 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 operation to produce output into a very slow operator, e.g., a display operator that allows

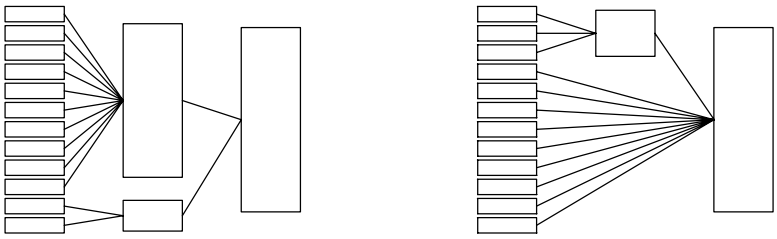


Figure 7. Stronger Effect of Optimized Merging.

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. The first merge should use a fan-in of $F_0 = ((W - f - 1)$ *modulo* $(F - 1)) + 2$. In the example of Figure 7, $(12 - 10 - 1)$ *modulo* $(10 - 1) + 2$ results in a fan-in for the initial merge of $F_0 = 3$. If the sort of Figure 7 were the input into a merge-join and a final fan-in of 5 were desired, the initial merge should proceed with a fan-in of $F_0 = (12 - 5 - 1)$ *modulo* $(10 - 1) + 2 = 8$.

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 size of the two inputs. For example, if two merge-join inputs are 1 MB and 9 MB, and 20 clusters are available for inputs into the two final merges, 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 subsection), 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., 1¼ times the size of memory. Hybrid hashing determines dynamically how much input data truly must be written to temporary disk files. In the example, only slightly more than ¼ 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 (1¼ 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

---

[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.

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

Table 4. Effect of Cluster Size Optimizations.

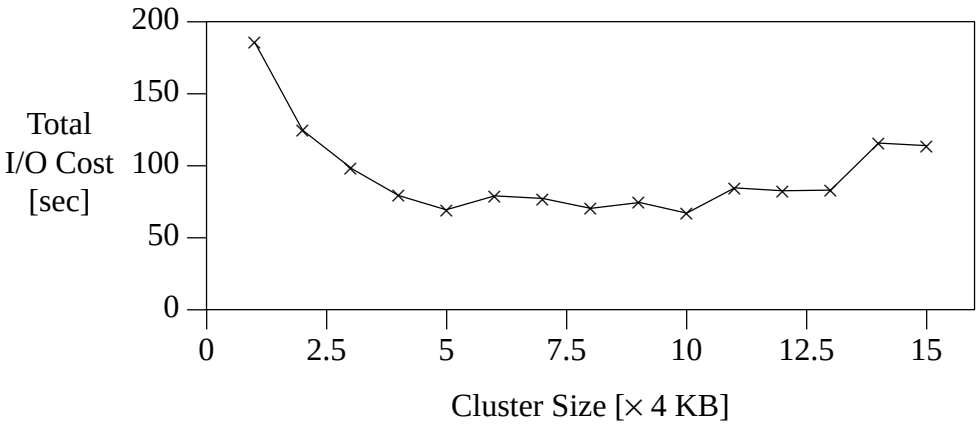


Figure 8. Effect of Cluster Size Optimizations.

exception of read-ahead and forecasting.[7] We used a sort space of forty pages (160 KB) within a fifty-page (200 KB) I/O buffer, varying the cluster size from one page (4 KB) to fifteen 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

[7] This experiment and a similar one were described in more detail earlier [Graefe 1990a; Graefe, Linville, and Shapiro 1994].

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 8, 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 (one 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 under the (unrealistic) assumption that the cost of an I/O operation is independent of the amount of data being transferred [Bratbergsengen 1984].

3.2. 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 $O(N \log N)$ 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 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-partitioning 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, Nakayama, and Takagi 1989; Nakayama, Kitsuregawa, and Takagi 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][8]. 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 9 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., $M < R \leq F \times M$.

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

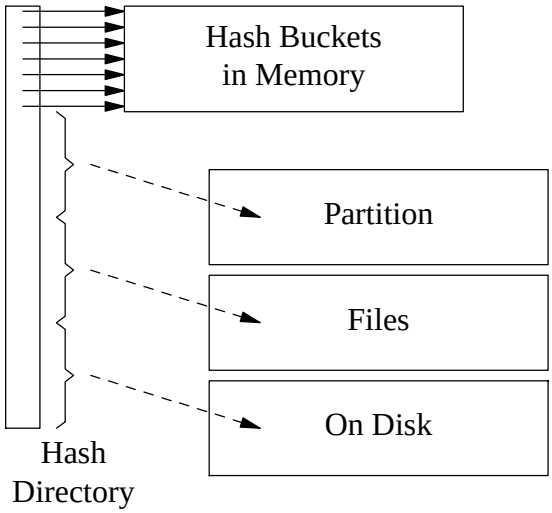


Figure 9. Hybrid Hashing.

[8] Although invented for relational join and known as hybrid hash join, hybrid hashing is equally applicable to all hash-based query processing algorithms.

during query optimization), the number of required partition files on disk must be determined. Call this number K, which must satisfy $0 \le K \le F$. Presuming that the assignment of buckets to partitions is optimal and 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 $K \times M + (M - (K + 1) \times C) \ge R$. 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 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 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 9, 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. Once hash table overflow occurred, all input were 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[9] can be 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]. 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, Nakayama, and Takagi 1989; Nakayama, Kitsuregawa, and Takagi 1988], a large number of small partition files is created and then

[9] Bucket in [DeWitt and Gerber 1985] means what is called an output partition in this survey.

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 non-uniform and 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 has no advantage. Another disadvantage 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; Graefe 1994].

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 10 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

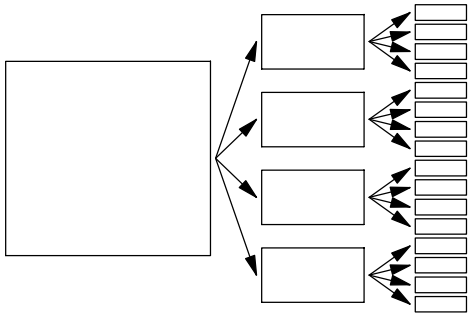


Figure 10. Recursive Partitioning.

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 $L = \lfloor \log_F (R / M) \rfloor$. 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

$$2 \times R \times L + 2 \times F^L \times \left(R' - (M - K \times C) \right)$$

$$= 2 \times \left(R \times (L + 1) - F^L \times (M - K \times C) \right)$$

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

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 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 (meta-data), from sampling the input data, 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 (e.g., histograms) gathered in earlier recursion levels to ensure that each partitioning step creates even partitions, i.e., that the data is partitioned with maximal effectiveness [Graefe 1993a; Graefe 1994]. As a final resort, if skew cannot be managed otherwise or if not distribution skew but duplicates are the problem, 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; Zeller 1991].

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, Linville, and Shapiro 1994]. 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.

Excursus 1: Reducing Memory-to-Memory Copying

In order to reduce volume and cost of copying build input items while building the hash table and writing build overflow files, some hybrid hashing implementations use the following technique [Graefe 1993c; Zeller and Gray 1990; Zeller 1991], which is particularly important in shared-memory parallel machines. Before the hash table is built, hash buckets are assigned to partitions, and a build overflow file is initialized for each partition. Thus, the hash function maps each data item not only to a hash bucket but also implicitly to a partition file. These partitions are called *prepared partitions*, and the number of prepared partitions is fixed once it is set.

When a build item is inserted into the hash table, the item is also copied into an output buffer allocated for its partition file. The clusters (pages) of possible overflow files are assembled at the same time the hash table is constructed. Very importantly, however, the clusters of these overflow files are not yet unfixed in the buffer, i.e., no I/O is performed as yet. Therefore, the partitions are called *resident partitions*. When the number of clusters with items in the hash table reaches the memory allotment for the hybrid hash operation, one of the resident partitions is unfixed and "spilled" to disk. All full clusters of this file are written to disk and the count of clusters holding items in the hash table is reduced accordingly. Only the last cluster of a spilled partition remains in memory to serve as output buffer. Thus, when overflow occurs, a resident partition becomes a *spilled partition*, freeing memory in the process. By spilling the largest resident partition, the algorithm frees the largest amount of memory. For binary operations such as hybrid hash join (discussed in detail later), spilling the largest resident partition also ensures optimal performance except in the case that the probe input is even more skewed than but in exactly the same way as the build input.

As more build input items are consumed, some of them belong to spilled partitions and are written to disk while others belong to resident partitions and are inserted into the hash table, possibly requiring more clusters to be allocated to partition files prepared but not yet spilled. When the number of clusters holding items in the hash table reaches the spilling threshold again, the next partition is unfixed and spilled, etc. For very large build inputs, all partitions will be spilled and the algorithm has transformed itself dynamically from an in-memory hashing operation first to a hybrid hashing operation and then to the first partitioning step in a recursive hashing operation. This implementation technique for hybrid hash join using dynamic transitions from resident to spilled partitions has several important effects, namely that (i) this hybrid hash join algorithm does not depend on accurate knowledge of its input size (or sizes for

hybrid hash join) as it spills partitions dynamically, (ii) build input items are packed densely in the available memory, thus permitting the maximal number of build items in the hash table, (iii) build input items are copied only once, when the partitions are prepared and the partition file clusters are assembled, (iv) the algorithm copes with hash value skew in the build input by choosing to spill the largest resident partition, and (v) the hash directory space freed when partitions are spilled can be used for bit vector filters to reduce probe overflow or for histograms to ensure effective partitioning in deeper recursion levels, should those be necessary [Graefe 1993a; Graefe 1993b].

Excursus 2: Hash Table Organization in Volcano

[10]Volcano's one-to-one match operator, which is based on hybrid hashing, uses the technique just discussed as well as another option that can eliminate all copying in some situations. Items can be inserted into the hash table without copying, i.e., the hash table points directly to records in the buffer as produced by one-to-one match's build input stream. If input items are not densely packed, however, the available buffer memory can fill up very quickly. Therefore, the one-to-one match operator has an argument called the *packing threshold*. When the number of items in the hash table reaches this threshold, items are packed densely into memory, i.e., output partition buffers. When the number of items in the hash table reaches a second threshold called the *spilling threshold* is the first partition file written to disk and the count of items in the hash table accordingly reduced. When this number reaches the spilling threshold again, the next partition is written, etc. If necessary, partitioning is performed recursively, with automatically adjusted packing and spilling thresholds. The unused portions of the hash table, i.e., the portions corresponding to spilled buckets, are used for bit vector filtering to save I/O to probe overflow files.

The fan-out of the first partitioning step is determined by the total available memory minus the memory required to reach the packing threshold. By choosing the packing and spilling thresholds, a query optimizer can avoid record copying entirely for small build inputs, specify overflow avoidance (and the maximum fan-out) for very large build inputs, or determine packing and spilling thresholds based on the expected build input size. In fact, because the input sizes cannot be estimated precisely if the inputs are produced by moderately complex expressions, the optimizer can adjust packing and spilling thresholds based on the estimated probability distributions of input sizes. For example, if overflow is very unlikely, it might be best to set the packing threshold quite high such that, with high probability, the operation can proceed without copying. On the other hand, if overflow is more likely, the packing threshold should be set lower to obtain a larger partitioning fan-out.

The initial packing and spilling thresholds can be set to zero; in that case, Volcano's one-to-one match performs overflow avoidance very similar to the join algorithm used in the Grace

[10] This excursus is adapted from [Graefe 1993c].

database machine. Beyond this parameterization of overflow avoidance and resolution, Volcano's hash-based one-to-one match algorithm also supports bit vector filtering as well as optimizations of cluster size and recursion depth similar to the ones used for sorting [Bratbergsengen 1984; Graefe 1990a]. Future reimplementations will also support histogram-based techniques to conquer and often even exploit non-uniform hash value distributions [Graefe 1993a; Graefe 1993b].

Excursus 3: Larger Units of I/O through Dynamic Allocation

While large units of I/O increase the sustainable I/O bandwidth, they also limit the number of buffers that can be kept in memory at any point of time. For the partitioning algorithms discussed here, this implies a reduced partitioning fan-out. However, there is a technique that permits a fan-out or cluster size about twice those calculated by the normal formula given above, i.e., $F = \lfloor M / C - 1 \rfloor$. Instead of dividing memory into F output buffers, each of size C pages, we divide memory into M pages. Each page is allocated individually. At the start of a partitioning process, each partition is allotted one page. The remaining pages are considered the pool of free pages.

When a partition's page fills up during partitioning of input items into output partitions, a new page is allocated from the pool of free pages. This may be done repeatedly, and there is no limit on how many pages may be allocated to a single partition. When the pool of free pages is exhausted, i.e., each of the M pages is allocated to some partition, the partition with the most allocated pages is determined and all pages but the last of that partition will be written *in a single I/O operation*. Notice that this requires a write system call capable of gathering data from multiple memory locations, such as the UNIX system call *writev*. After writing, the memory pages written to disk are returned to the pool of free pages and may then be used by any partition.

The first write operation will probably write slightly more than $M / (F + 1)$ pages. However, subsequent write operations will have benefited from pages being freed after earlier writes. Thus, their data volume will be larger than $M / (F + 1)$. In fact, for large input sets, the average write size will approach $2 \times M / F$. Thus, by exploiting the gather-write system call, this technique about doubles the unit of I/O for a given fan-out. Similarly, it can double the fan-out for an expected unit of I/O.

This technique has one drawback, but also an important advantage. On the one hand, fragmentation might increase if records must fit into individual pages, not large clusters. On the other hand, this technique permits extensions towards dynamic memory allocation and re-allocation. When memory contention in a system increases, memory from the pool of free pages can be free by the hash algorithm. If the pool of free pages does not hold a sufficient number of pages, it can be grown for that purpose using the standard method. The effect is that the average unit of I/O will shrink; however, it will do so gracefully with the degree of memory contention. When memory contention diminuishes and more memory becomes available, this technique can make use of it immediately and effectively. Finally, this technique can be combined with hybrid hashing by either restricting the dynamic allocation to the spilled partitions or by having all

partitions compete for memory equally and then applying bucket tuning as appropriate.

Excursus 4: Temporary Files and Complex Objects

Finally, there is an interesting issue of great importance for object-oriented database systems that support large data volumes, set operations and value-based matching using hybrid hash join and other algorithms using temporary files. Consider a complex object with multiple component records. While in memory, such a complex object is typically linked together by memory-to-memory pointers. If such an object is written to disk, e.g., a run file in an external sort or a partition file in hashing, four methods for dealing with the subcomponent records present themselves.

First, only the root component is actually written to disk, while the other components remain in main memory and are not moved. The memory pointers in the root components can be written to disk and later read back without change. This methods works correctly only if the subcomponents can indeed remain in memory without being moved. Thus, if the root components require the largest amount of space and all subcomponents are guaranteed to fit in memory, this method may indeed work very efficiently. It is particularly attractive if many complex objects are known to share subcomponents. Otherwise, the space dedicated to the subcomponents may severely reduce the memory space available for manipulating root components.

Second, each complex object can be assembled into a single large record, which is written to the temporary file and later retrieved to permit reconstruction of the complex object structure. Unfortunately, this technique requires further refinement to retain information about shared subcomponents. Moreover, if the subcomponents are large, this technique might require substantial volumes of I/O, including redundant I/O for shared subcomponents.

Third, in order to remove subcomponents from main memory as well as to save on repetitive I/O for large subcomponents during multi-level merging or partitioning, the subcomponents might be saved in a special temporary file, from which they are retrieved when needed, in particular at the end when the matching operation is to produce complex objects assembled in memory. While this technique seems to overcome the shortcomings of the previous two methods, it also suggests further improvements as it does not fully exploit the capabilities of inter-object references that are so fundamental in object-oriented database systems.

Fourth, since the root components refer to the subcomponents not only by means of memory addresses but also by object identifiers or record locations, the temporary files can be written with root components only and the subcomponents be ignored, to be retrieved again later when needed. If the operation at hand requires attribute values from subcomponents, those can be attached to the root components in the temporary files. Of course, this technique may require that each complex object be assembled twice in main memory. Thus, it should be combined with a cost-based query optimizer that carefully considers at which point in a query evaluation plan to assemble subcomponents of complex objects and when to perform set operations and value-based matching.

4. 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 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, [Carey, Haas, and Livny 1993; Stonebraker 1991]. Research into database systems supporting and exploiting deep storage hierarchies is still in its infancy.

On the other hand, motivated both by the desire for faster transaction and query processing performance and by the decreasing cost of semi-conductor memory, some researchers have considered in-memory or main memory databases, e.g., [Analyti and Pramanik 1992; Bitton, Hanrahan, and Turbyfill 1987; Bucheral, Theverin, and Valduriez 1990; DeWitt et al. 1984; Gruenwald and Eich 1991; Kumar and Burger 1991; Lehman and Carey 1986; Li and Naughton 1988; Severance, Pramanik, and Wolberg 1990; Whang and Krishnamurthy 1990]. However, for most appications, an analysis by Gray and Putzolo demonstrated that main memory is cost-effective only for the most frequently accessed data [Gray and Putzolo 1987]. The time interval between accesses with equal disk and memory costs was five minutes for their values of memory and disk prices. Only data accessed at least every five minutes should be kept in memory, whereas other data should reside on disks. Although this threshold time interval is expected to grow as main memory prices decrease faster than disk prices, there will always be substantial data volumes that are accessed less frequently than the threshold but that must be queried efficiently. Therefore, for the purposes of this survey, we presume a disk-based storage architecture and consider disk I/O one of the major costs of query evaluation over large databases.

4.1. File Scans

The first operator to access base data is the file scan, which is the typical operator at the leaves of physical algebra expressions. Its role is to make a file available as iterator, i.e., with *open*, *next*, and *close* procedures. There are only a few points to be made here about file scans. First, file scans should evaluate simple selection predicates, i.e., predicates that can be evaluated based on a single record. In order to minimize the number of software levels an item is handled by before it is eliminated by a simple Boolean expression, the expression evaluator should be invoked as early as possible [Astrahan et al. 1976], certainly before a record is copied from the data page in the buffer. Moreover, for complex Boolean expressions, expression evaluation should be "short-circuited," e.g., "A and B" should be evaluated as "if A then B else false," and the individual clauses should be ordered based on their probablity of eliminating a record from further consideration and their cost (e.g., integer comparison vs. string matching) [Hanani 1977].

Second, file scans 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. For example, 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 multi-page read-ahead.

4.2. 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 fan-out 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, e.g., [Becker, Six, and Widmayer 1991; Beckmann et al. 1990; Bentley 1975; Finkel and Bentley 1974; Guenther and Bilmes 1991; Gunther and Wong 1987; Gunther 1989; Guttman 1984; Henrich, Six, and Widmayer 1989; Hoel and Samet 1992; Hutflesz, Six, and Widmayer 1988a; Hutflesz, Six, and Widmayer 1988b; Hutflesz, Six, and Widmayer 1990; Jagadish 1991; Kemper and Wallrath 1987; Kolovson and Stonebraker 1991; Kriegel and Seeger 1987; Kriegel and Seeger 1988; Lomet and Salzberg 1990b; Lomet 1992; Neugebauer 1991; Robinson 1981; Samet 1984; Six and Widmayer 1988]. One of the few multi-dimensional index structures actually implemented

in a complete database management system are R-trees in Postgres [Guttman 1984; Stonebraker, Rowe, and Hirohama 1990].

The large variety of index types can be described by the following characteristics. First, does the index support range retrievals and ordered scans, or only exact-match equality retrievals? This issue is the main difference between sort-based indices such as B-trees and hash-based indices. Indices that support ordered key domains tend to have logarithmic insertion, deletion, and search costs, while index and storage structures based on hashing typically have constant average maintenance complexity. The exception to this rule are hash-based indices using an order-preserving hash function.

Second, is the index structure static (e.g., ISAM) or dynamic (e.g., B-tree)? Either the index structure allocates a fixed number of "buckets" when it is first created and resorts to overflow pages if buckets cannot hold all data items that logically belong in them, or it reorganizes itself incrementally as items are inserted and deleted. While dynamic structures such as B-trees are frequently preferred because they adapt gracefully to growing data volumes, static structures may have distinct advantages with respect to concurrency control and recovery since their internal nodes do not change except during reorganization.

Third, an index structure can be programmed to permit only single-attribute search or it can support multiple attributes in a hierarchical fashion. Such index implementations supporting composite keys, e.g., last name and first name, are still single-dimensional indices because the components of the composite key are ordered hierarchically into a major and a minor key. Note that this is an implementation detail; logically, multiple attributes are concatenated into one search key.

Fourth, does the index support only single-dimensional data or also data representing multiple dimensions? True multi-dimensional indices support all dimensions as equals, for

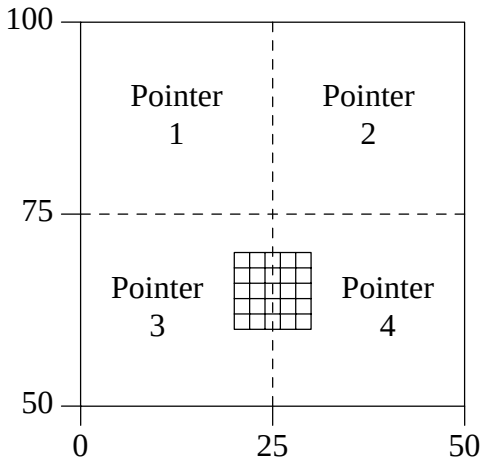


Figure 11. A Node in a Quadtree.

example the x- and y-axes in a geometric application. Figure 11 shows a node in a quadtree [Beckley, Evans, and Raman 1985; Finkel and Bentley 1974; Samet 1984; Unnikrishnan, Shankar, and Venkatesh 1988], the simplest multi-dimensional index structure. The region represented by this node is divided along both dimensions, and each of the four subregions is represented by its own node. Quad-trees can trivially generalized from 2 to N dimensions. Another simple tree structure to represent points in N-dimensions is the kD-tree, which splits each node in only one of the k dimensions rather than in all dimensions [Bentley 1975].

The use of multi-dimensional indices for record-keeping applications has been largely ignored, despite their effectiveness for conjunctive queries. Consider a search for employees within both a certain age and salary range, e.g., $20 \leq$ employee.age ≤ 30 and $60 \leq$ employee.salary ≤ 70. In a system with single-dimensional indices only, either only one index can be utilized or two pointer lists must be intersected. Using only one index results in more data accesses since all employee records satisfying one clause must be inspected to evaluate the other clause. Intersection of two lists (or sets) can be quite expensive as will be seen in the subsequent section on binary matching, which includes relational join and set intersection. A two-dimensional index, on the other hand, permits more direct access to only the required employees because it supports both restriction clauses simultaneously. Presuming that the two dimensions represent age and salary, the relevant search region is shaded in Figure 11. It is clear from the figure that more than one pointer from one node may need to be followed for some range queries, possibly at each index level. However, a multi-dimensional index promises to be still faster for large data sets than any method using single-dimensional indices. Of course, multi-dimensional indices can also be used for disjunctive queries, although their performance advantage is not as obvious for disjunctive as for conjunctive queries. A possibly overwhelming argument against the use of multi-dimensional indices in record-keeping applications is the update performance of many of these structures.

Fifth, do the indices support point data or range data? Range data have two data points in each dimension; the standard example is the case of two-dimensional rectangles. One method to support N-dimensional range data is to use an index structure for point data in $2 \times N$ dimensions. For example, the region shaded in Figure 11 could be represented in a four-dimensional index for point data as $x_1 = 20$, $x_2 = 30$, $x_3 = 60$, and $x_4 = 70$. One of the problems with this solution is that pairs of dimensions (the start and end value in the original dimensions) will likely be correlated, and the data structure may or may not include space- and search-efficient balancing mechanisms.

Sixth, most index implementations can be switched to accept or reject duplicate keys. Thus, indices are used to enforce uniqueness constraints, particularly for identifying keys in database systems.

Finally, some database systems permit that an index cover only part of an underlying data set, e.g., only those data items satisfying a frequently used predicate [Stonebraker 1989]. While not discussed much in the database research literature, such *conditional* or *partial indices* have been implemented in Unisys' DMS II, IBM's AS/400, Wang's Pace systems, and probably others. Conversely, indices are sometimes used to provide associativity for more than one

underlying data set. The best-known examples are the two-relation *join indices* proposed by Valduriez [Valduriez 1987], view indices analyzed by Roussopoulos [Roussopoulos 1991], and 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.

Upon closer inspection, partial indices and multi-set indices belong to a different conceptual level than the earlier characteristics. The first six criteria pertain to the physical structure, the layout and use of pages on disk, whereas these last issues pertain to the intended use of files. For example, join indices can be implemented with a variety of physical storage structures, including heap files, two-dimensional indices, or pairs of B-trees. Thus, join indices are a form of precomputation, not a physical file format as discussed in this section.

Table 5 shows some example index structures classified according to four of these characteristics. We omitted hierarchical concatenation of attributes and uniqueness, because all index structures can be implemented to support these. The indication "no range data" for multi-dimensional 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, multi-disk and multi-processor

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-trees	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 1990b]
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, Hinterberger, and Sevcik 1984]

Table 5. Classification of Some Index Structures.

implementations, page placement on disk, concurrency control, recovery, order-preserving 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 multi-dimensional index structures.

Storage structures typically thought of as index structures may be used as primary structures to store actual data or as redundant structure ("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 storage structure.

If the primary file is a standard record file, i.e., a file with no inherent structure or ordering according to some field or attribute values, redundant indices can be used to cluster the actual data items, i.e., to assign data items to locations in the file such that the order or organization of items in the data file corresponds to the order of the index entries. Such indices are called *clustering* indices; other indices are called *non-clustering* 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*. Non-clustering indices must always be *dense*, i.e., the number of entries in the index is equal to the number of indexed 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 retrieval to the index scan.

There are several advantages to separating index scan and record retrieval. 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/O's are required (consider the number of I/O's for retrieving N successive index entries and those to retrieve N index entries plus N full records, in particular if the index is non-clustering [Mackert and Lohman 1989], and (ii) the remaining I/O operations are basically sequential along the leaves of the index (at least for a 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 RID's to obtain all attributes required for a query (join algorithms are discussed below in the section on binary matching). For example, by matching entries in indices on salaries and on names by equal RID's, 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 efficient to compute the union or intersection of two RID lists obtained from two index scans than to use 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 record will be retrieved that truly contribute to the join result [Kooi 1980]. Two variants on this method are the asymmetric version which joins one data set with another data set's index and a complex join of *N* indices preceeding the record retrievals from any of the underlying *N* data sets. Fifth, for non-clustering indices, sets of RID's 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. A sorted RID list is particularly useful when combined with effective multi-page read-ahead controlled by a list of page numbers. Obviously, several of these techniques can be combined. In addition, some systems such as Rdb/VMS and DB2 use a very sophisticated implementations of multi-index 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]. For systems in which the primary data structure is a search structure and secondary indices map to a search key of the primary structure (as in Tandem's NonStop system, as described above), many of the ideas listed above can easily be adapted; for example, a scan of the secondary index yields primary keys that can be used in value-based operations such as joins before the primary structure is employed to obtain complete records [Toyoma 1993].

Record access performance for non-clustering indices can be addressed without performing the entire index scan first (as required if all RID's are to be sorted) by using a "window" of RID's. Instead of obtaining one RID from the index scan, retrieving the record, getting the next RID from the index scan, etc., the retrieval operator (sometimes called "functional join") could load *N* RID's, 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, as discussed in the next subsection.

4.3. Unclustered Index Lookup and Complex Object Assembly

[11]The main costs of a random I/O operation are disk arm seek and rotational latency. Thus, these

[11] This section is a summary of [Keller, Graefe, and Maier 1991].

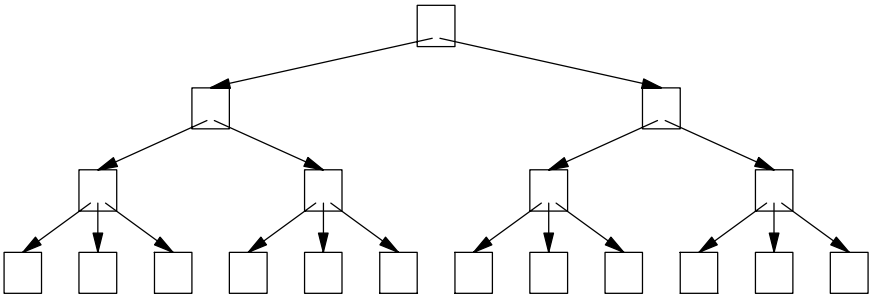


Figure 12. A Complex Object with Multiple Levels of Subcomponents.

costs must be addressed in order to speed retrieving records pointed to by an unclustered index or assembling a set of complex objects from disk into memory using references from one object component to another. One technique is sorting all references on their physical location, which has some restrictions. For unclustered index retrievals, the entire index scan must be complete before any record retrieval can commence. For assembly of complex objects with multiple levels of subcomponents, only one level of object assembly can be accomplished at a time, because further references are not known until the previous levels of subcomponents have been retrieved. If objects are clustered such that all components of an object are located closely together on disk, this method is clearly a poor choice. Instead, the most desirable algorithm is one that adapts to any clustering of objects on disk.

From an algorithmic perspective, record retrieval after index lookup is a special case of complex object assembly. The principal idea of the complex object assembly operator of [Keller, Graefe, and Maier 1991] is to strike a balance between resolving references one at-a-time and consuming an entire set of references before resolving any of them. A window of W objects is maintained during assembly, indicating that W objects are being assembled in an concurrent or interleaved way. Given the partially assembled objects in the current window, the assembly algorithm maintains a list of known but as yet unresolved references. These references are called the "open" references. The list is actually a priority queue, ordered by physical disk location. Using this queue, the algorithm request data pages in ascending and descending order, similar to the "scan" policy known in operating systems. If the number of open references is sufficiently large, substantial savings in disk seek operations can be expected. Moreover, shared components can be detected and retrieved only once. The same is true for components located on the same disk page, resulting in even larger I/O savings.

After each I/O operation, some object may be completed and can be passed from the assembly operator to its consumer. Alternatively, the algorithm can be modified to ensure that objects are passed to the consumer operator in the same order as object references were obtained from the producer operator. Another interesting modification to the basic algorithm is inclusion of selection predicates and their impact on scheduling. If a certain subcomponent may disqualify an entire object, it may be worthwhile to retrieve this component early, possibly making a number of other disk accesses obsolete. Given multiple choices, the component with the highest probability of disqualifying a complex object should be retrieved first [Hanani 1977], which, of

course, may create a conflict with the "scan" scheduling policy.

The assembly operator creates a spectrum of choices, of which previously only the end points had been considered. In the extreme cases, a window size of $W = 1$ results in one-at-a-time complex object assembly, while $W = infinity$ represents complete input consumption followed by index retrieval or complex object assembly. Obviously, the window size W is an important tuning parameter. More importantly, the assembly operator's flexibility isolates a query processing engine from the effects of object clustering. Most clustering strategies presume that there is a dominant access pattern that must be optimized [Chang and Katz 1989; Cheng and Hurson 1991; Ghandeharizadeh and DeWitt 1990b; Harada et al. 1990; Omiecinski and Scheuermann 1990; Snodgrass and Shannon 1990; Tsangaris and Naughton 1991; Tsangaris and Naughton 1992]; the complex object assembly algorithm permits efficient retrieval of object components somewhat independently of their locations on disk. If objects are clustered suitably on disk, such locality is exploited. However, if they are not, the assembly algorithm uses a set of "open" references and the "scan" policy to improve the locality of actual disk accesses.

4.4. Faster Storage Techniques

Both for file scans and for index retrieval, the raw performance of the underlying storage (disk) system is crucial. Numerous ideas for faster disk access have been proposed, including RAM disks, i.e., simulation of disk drives using semi-conductor memory, and improvements in disk hardware speed, including physically smaller disks for faster seeks, faster disk rotation for shorter rotational latencies, and improved channel transfer rates. Other commonly used techniques are dual- or even quadruple-ported memory for concurrent transfer to or from multiple disks and processors, multiple paths from disks arms to I/O channels, and the insertion of RAM caches at various points in the data staging hierarchy, e.g., the disk controller or the drive. Write-only disk caches are an interesting proposal for cache use because most disk writes can be delayed in safe RAM and later piggy-backed transparently onto disk reads to the right cylinder or track, thus giving the illusion of instantaneous writes [Solworth and Orji 1990].

Recently, the idea of using multiple disk devices as a single, more powerful device has received considerable attention, and is now commonly known as Redundant Array of Inexpensive Disks (RAID) [Patterson, Gibson, and Katz 1988], the basis of which had been proposed earlier as disk mirroring [Bitton and Gray 1988] and as disk striping [Salem and Garcia-Molina 1986]. Put simply, by distributing the data and blocks of a file over multiple disk devices, higher transfer rates can be achieved. Furthermore, by using controlled redundancy, the mean time to failure as well as the mean time to repair can be improved substantially [Bitton and Gray 1988; Chen and Patterson 1990; Copeland et al. 1988; Garcia-Molina and Salem 1988; Gibson et al. 1989; Muntz and Lui 1990]. Reliability and availability can be further improved by dual-redundancy and the use of "hot spares," i.e., disk drives that are unused until a broken disk must be replaced quickly. These ideas can be further generalized from multiple disks to multiple nodes in distributed database and file systems [Stonebraker and Schloss 1990].

Several RAID levels have been defined [Patterson, Gibson, and Katz 1988]. RAID 1 is simple disk mirroring. RAID 2 stripes each data byte plus a parity bit over 9 disks. RAID 3

defines "array pages" (the same page location on all disk spindles in the array), which include a parity page located on a designated parity drive within the disk array and derived by exclusive bit-wise *or*-ing all data pages. RAID 4 permits reading not only entire array pages as in level 3 but also individual data pages for faster random access, although each updates still require updating a parity page. The new parity page content can be derived by exclusive bit-wise *or*-ing the old parity page content with both the old and new data page content. RAID 5 alleviates the higher I/O load for the parity drive in update-intensive applications to rotating (among the disk spindles) the assignment of parity pages (and therefore the drive) within array pages.

One problem with all disk array designs is ensuring proper placement of data on the disks to obtain the benefits of parallel I/O without incurring the additional overhead of controlling multiple devices for relatively simple and small requests [Stonebraker 1987; Stonebraker et al. 1988; Weikum, Zabback, and Scheuermann 1991]. Redundant disk arrays might be attractive for transaction logs because they provide reliability and bandwidth and therefore can provide the notion of "stable storage" for large transaction volumes, but they also introduce the problem of increased waiting time (latency) until enough log records are collected to fill an entire array page, a transaction's commit record is written to stable storage, and the commit can be acknowledged to the user. Without solving these problems, it will be difficult to maximize disk array benefits; nonetheless, several vendors are making disk arrays commercially available, including implementations of RAID control logic in device controllers to permit replacing a disk drive by a disk array.

While RAID-style disk arrays provide increased capacity, reliability, availability, and bandwidth for permanent data, they might not be the optimal I/O architecture for temporary data. The first reason is that writing parity blocks will be unnecessary (if a disk fails, most probably the entire transaction will abort) and reduce processing performance (two writes instead of one). Second, important operations using multiple temporary files might not make optimal use of multiple disk drives if they are controlled as a single unit in a disk array. Consider sorting[12] a very large file using a disk array, with each run file striped across all disk drives. Merging is faster using a disk array than using a single disk because many disk arms perform the required number of disk seeks. If each merge input file were stored contiguously on its own disk, however, merging could proceed without any disk seeks at all. Recall that our earlier discussion on cluster size and fan-in justifies and even recommends fairly small fan-ins; therefore, a modest disk array will permit one input file per disk. Thus, for run files in a sort operation, disk striping as built into disk array controllers is not optimal. It would be desirable if disk array controllers provided disk striping and parity blocks for permanent files but at the same time permitted sophisticated application software such as database management systems to set their own disk space allocation strategies. Current disk array controllers are examples of useful mechanisms

[12] The same concern about disk arrays applies to hash-based query processing algorithms, because merging and partitioning are quite similar, as discussed later in the section on the duality of sorting and hashing.

packaged with fixed strategies deplored in all papers on the interface between operating systems and database systems, e.g. [Stonebraker 1981]. Space allocation for temporary files on multiple disk drives for sorting and partitioning in database query processing is an interesting issue that is not completely resolved at this time.

Other interesting proposals for I/O performance improvement are safe RAM, database caches, differential files, and log-structured files. Safe RAM is normal semi-conductor main memory that is dual-powered by the normal power supply and by batteries [Copeland et al. 1989]. The batteries are used to provide safety against data loss in the event of a power failure, either by retaining the data until normal power is restored or until the data have been written to a disk, which also must be battery-powered during this short time of writing. If properly implemented, safe RAM can be considered "stable storage" necessary for the implementation of transaction- or ACID-semantics (which were briefly discussed in the introduction).

The database cache does not require specialized hardware [Elhard and Bayer 1984]. It consists of (volatile) cache in main memory and a "safe" to back up the cache. Updates to the database are performed in place, but only after the updating transaction has been committed. In other words, the database never contains dirty, uncommitted data. The safe contains those portions of the cache that are relevant to a restart after a crash. Since writing to the safe is sequential (append-only), it is faster than random I/O for updates-in-place in the database. Moreover, the database cache alleviates "hot spots" from the database by retaining hot pages in the cache. Elhard and Bayer observed significant performance improvements due to the database cache in their experimental evaluation [Elhard and Bayer 1984]. While safe RAM and database caches increase transaction processing throughput, they have no effect on query performance.

The concept of differential files splits a file into a "main file" and an update log [Severance and Lohman 1976]. The log or differential file contains all the updates to a main file, which is never modified. Each record retrieval must first determine whether or not the record has been updated, i.e., whether the valid version of the record is in the main file or in the differential file. This decision can be made efficient for most records by using bit vector filters or Bloom filters [Gremillion 1982; Mullin 1983].

Recently, differential files have been generalized to log-structured file systems [Ousterhout and Douglis 1989; Rosenblum and Ousterhout 1991; Rosenblum and Ousterhout 1992], In effect, all file content is in the differential files; therefore, the "main file" of the original differential file idea is not required in log-structured file systems. Writing logs can be made particularly efficient by writing large physical units at a time, e.g., entire disk cylinders. However, this requires that an asynchronous service process identifies and sweeps cylinders with few remaining valid data blocks. Log-structured file systems have also been called write-optimized file systems because updates-in-place at random disk locations are completely eliminated and replaced by sequentially appending all new data to the end of the log whereas collecting all items of a logical file might require many random reads. Since data modifications in log-structured files are always appended at the end of the file (since it is a log!), they can be used very effectively together with densely packed compressed records (because a record modification might change the compression ratio and therefore the record size) and with RAIDs (where updating a single block in-place requires

multiple I/Os for the parity block).

4.5. 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 surveys many of the buffer management issues, including those pertaining to issues of recovery, e.g., write-ahead logging [Effelsberg and Haerder 1984]. In his survey paper on the interactions of operating systems and database management systems, Stonebraker pointed out that the "standard" buffer replacement policy, LRU (least recently used), is wrong for many database situations [Stonebraker 1981]. 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 focused on the non-linear performance effects of buffer allocation to many relational algorithms, e.g., nested loops join [Sacco and Schkolnik 1982; Sacco and Schkolnik 1986]. Chou and DeWitt 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 [Chou 1985; Chou and DeWitt 1985]. A recent study into buffer allocation is the study by Faloutsos et al. on using marginal gain for buffer allocation [Faloutsos, Ng, and Sellis 1991; Ng, Faloutsos, and Sellis 1991]. A very promising research direction for buffer management in object-oriented database systems is the work by Palmer and Zdonik on saving reference patterns and using them to predict future object faults and to prevent them by prefetching the required pages [Palmer and Zdonik 1991].

The interactions of index retrieval and buffer management were studied by Sacco as well as Mackert and Lohman [Mackert and Lohman 1989; Sacco 1987], while several authors studied database buffer management and virtual memory provided by the operating system, e.g., [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 the next iterator (e.g., if a predicate succeeds). Because the iterator control and interaction of operators ensures that items are never produced and fixed before they are required, the iterator protocol is very efficient in its buffer usage.

Some implementors, however, 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.

4.6. Physical Database Design

In order to minimize the I/O costs in a database system, it is important that (i) the data structures on disk permit efficient retrieval of only relevant data through effective access paths, and (ii) data be arranged and placed on disk such that the I/O cost for relevant data is minimized. Both of these concerns are addressed in physical database design. Because physical database design is a wide area in which there are many studies on individual techniques but no comprehensive set of rules or guidelines on how to consider all of them in combination, we only list a number of choices to be made in physical database design and a few selected references. These options represent design choices and implementation effort for the DBMS implementor and vendor; their implemented subset then exists as actual physical database design choices for the DBMS user and database administrator. There is no existing database system at this point that supports all of these choices, although all of them could have a significant performance impact. These choices are not all orthogonal; the goal at this point is to outline the scope of physical database design and its concerns. For the reader's convenience, we have assigned issues into groups, although these groups are somewhat arbitrary. Table 6 summarizes the issues discussed in this section.

Physical representation types for **abstract data types** is only slowly gaining research attention for object-oriented database systems but will likely become a very important tuning option. Examples include sets represented as bit maps, arrays, or lists and matrices represented densely or sparsely, by row or by column or as tiles, e.g. [Maier and Vance 1993]. The goal is to bring physical data independence to object-oriented and scientific databases and their applications.

Physical **pointers**, references, or object identifiers to represent relationships support "navigation" through a database, which is very good for single-instance retrievals and often improves set matching, but also creates a new type of updates, structural updates, which may increase the complexity of concurrency control and recovery [Carey et al. 1990; Chen and Kuck 1984; Rosenthal and Reiner 1985; Shekita and Carey 1990].

Vertical partitioning of large records into multiple files, each with some of the attributes, allows faster retrieval of the most commonly used attributes; the problem is that joins or join-like operations are required when attributes from multiple files are needed [Cornell and Yu 1987; Cornell and Yu 1990; Hammer and Niamir 1979; Muthuraj et al. 1993; Navathe et al. 1984; Navathe and Ra 1989].

Data **compression** is undervalued in most database implementations but will increase in popularity with the introduction of database systems that process compressed data to improve

Item Representation	Abstract data types Physical pointers Vertical partitioning Compression
File Formats	File format Free space for logically organized data
Associative Search	Index selection
Other Redundant Data	Replication Derived information, materialized views
File Access Performance	Clustering (co-locating related items) Declustering/striping Horizontal partitioning Partitioning of indices
Update Management	Differential & log-structured files Versions Special CC&R versions
Archival Storage	Archival storage Staging between storage levels

Table 6. Physical Database Design Issues.

performance and of multi-level storage hierarchies ranging from on-chip caches over main memory and disks to automated tape libraries [Cormack 1985; Graefe and Shapiro 1991; Jagadish 1990; Li, Rotem, and Wong 1987; Lynch and Brownrigg 1981; Olken and Rotem 1989].

File formats concern the most basic level of physical database design: page allocation on disk (e.g., pre-allocation in contiguous extents), grouping of pages into units of I/O (called "clusters" here), division of clusters into records, fixed vs. variable-length records, space management within clusters, and records spanning clusters.

Free space in physical containers such as pages and files is useful for all data organized by some logical rule. For example, to facilitate easy and fast insertion into a file stored in the sort order of one or some of the record fields, some free space is typically left when a file is loaded. Free space can be left within each cluster (page), or a free cluster can be inserted periodically to serve as overflow page for surrounding pages. Free clusters increase the interval between database reorganizations and are particularly effective if their occurrence is coordinated with the size of sequential prefetch or read-ahead or with the physical geometry of the storage device (track or cylinder). Indices are also often created with some initial fraction of free space. The important tradeoff is the ability to insert new data without allocating new disk block, possibly far away, vs. the desire for compact representations and fast scans of large data collections.

Index selection must determine which index structures to create for which attributes and attribute combinations by comparing their benefits for searching against their update costs

[Guenther and Bilmes 1991; Hammer and Chan 1976; Kemper and Wallrath 1987]. Moreover, many authors have suggested various forms of multi-file indices and path indices [Bertino and Kim 1989; Bertino 1990; Bertino 1991; Haerder 1978; Kemper and Moerkotte 1990a; Valduriez 1987] and "partial indices," i.e., indices that include only a subset of an underlying relation or data set.

Replication for reliability and availability ensures access to correct data at any time and often increases retrieval performance during normal operation but requires complex and expensive consistency management on updates and after separation of storage or processing units [Bitton and Gray 1988; Copeland et al. 1988; Hsiao and DeWitt 1991; Patterson, Gibson, and Katz 1988].

Derived information, e.g., materialized relational views, permits fast access to precomputed query results but requires that the derived information is recomputed, invalided, removed, or marked out-of-date after updates to the based data [Blakeley and Martin 1990; Hanson 1987; Hudson and King 1989; Jhingran 1991; Kinsley and Hughes 1992; Qian and Wiederhold 1991; Roussopoulos 1991; Tompa and Blakeley 1988; Yang and Larson 1987]. Another very important form of derived data is replicating individual attribute values for faster access, in particular in database systems that do not support clustering across multiple data types [Babb 1982; Schkolnick and Sorensen 1981]. For example, if two collections *departments* and *employees* are linked by a many-to-one relationship such as a foreign key *employee.dept-id* and the workload requires very frequent joins of these two collections, the database system may support "denormalizing" them by redundantly keeping descriptive attributes from the one-side within each item on the many side, e.g., department name and location with each employee item, and by keeping aggregated information from the many side with each item on the one side, e.g., each department's count of employees. Since these are redundant attributes, the database system should propagate updates and avoid joins automatically. In other words, such propagated attribute values should be supported similarly to redundant index structures as part of physical data independence.

Clustering means assignment of data items such as records to disk locations in order to maximize the relevant information fetched with each I/O operation and therefore to reduce the number of I/O operations required to satisfy a database request — three important problems are prediction of future access patterns, compromising among different access patterns such as navigational access and set-oriented processing, and the dynamic change of access patterns [Chang and Katz 1989; Cheng and Hurson 1991; Ghandeharizadeh and DeWitt 1990b; Harada et al. 1990; Omiecinski and Scheuermann 1990; Snodgrass and Shannon 1990; Tsangaris and Naughton 1991; Tsangaris and Naughton 1992].

Declustering (striping) of pages over multiple disks or multiple nodes in a parallel or distributed database system permits more disk I/Os per unit time and higher total I/O bandwidth for a data collection at the expense of control overhead [Copeland et al. 1988; Gray, Horst, and Walker 1990; Patterson, Gibson, and Katz 1988; Salem and Garcia-Molina 1986; Stonebraker and Schloss 1990; Weikum, Zabback, and Scheuermann 1991].

Horizontal partitioning of items in large data collections using either round-robin partitioning (which is similar to striping) or systematic schemes such as hash- or range-partitioning is generally useful; its problems are selecting the partitioning attribute(s) and that various partitioning schemes interact differently with sort- and hash-based query evaluation algorithms for selection, join, etc. [DeWitt et al. 1986; Gerber 1986; Ghandeharizadeh and DeWitt 1990a; Li, Srivastava, and Rotem 1992; Wolf et al. 1988].

If data sets are partitioned, there are two principal methods of **partitioning the indices** and other redundant data sets such as materialized views: either they are local, i.e., a separate file or index structure is created for each partition of the underlying data set, or they are global, i.e., a single file or index structure for the entire logical data set is partitioned over multiple disks or nodes in parallel or distributed systems. Both schemes have advantages and disadvantages; the choice depends on usage patterns, communication costs, and availability and maintenance effects.

Differential files and **log-structured file systems** enable append-only updates, particularly useful in environments with compression and RAID-style storage devices, but destroys the contiguity of each file's space allocation on the storage media [Rosenblum and Ousterhout 1991; Rosenblum and Ousterhout 1992; Severance and Lohman 1976].

Versions are particularly useful in design environments where falling back to an earlier alternative is often useful after some exploratory updates and where different configurations require different versions of the same data item or object, e.g. [Katz and Lehman 1984; Katz, Chang, and Bhateja 1986; Stonebraker 1987].

Versions have also been used in **optimistic concurrency control** [Haerder and Petry 1987; Kung and Robinson 1981; Merchant et al. 1992; Papadimitriou and Kanellakis 1984], which requires special storage structures for versions e.g., a separate storage area for temporary, transient record versions (version pool) or small version "caches" on each database page [Bober and Carey 1992].

Archival storage has recently been pushed to increased research interest by the very largest database installations and by the emerging interest in database systems for scientific applications; the problems are storage formats appropriate for archival storage devices, selection of data to be moved ("retired") to archives, and the fact that data volumes, bandwidths, and access latencies vary widely from level to level in the storage hierarchy which may not permit direct adaptations of existing buffer replacement strategies [Dozier 1992; Silberschatz, Stonebraker, and Ullman 1991; Stonebraker 1991].

Automatic and timely **staging** of data to higher storage levels is a generalization of prefetching used in all high-performance database systems; the new problems are again the new data volumes, bandwidths, and access latencies [Ghandeharizadeh et al. 1991; Palmer and Zdonik 1991].

It is important to recognize that most of these choices exist independently of the data model. For example, replication has beneficial availability and performance effects in network, relational, semantic, and object-oriented databases alike. On the other hand, clustering might

have more effect in systems with logical or physical references, i.e., network and object-oriented databases, but master-detail clustering is used in relational system as well and is particularly effective in conjunction with index and pointer joins (to be discussed later in the section on binary matching). Thus, extensible database systems designed to build high-performance database systems must allow for a wide array of physical database design options.

Rather than grouping physical database design options by their usability with one or more particular data models, it might be more useful to distinguish them by the amount of semantic knowledge they require. For example, replication (through page-by-page copying) and striping (on the basis of file pages) do not require any knowledge about the semantics of the data. Sorting requires some knowledge pertaining to locations of records in pages and of sort field in records as well as the correct sort order. Derived data require even more. Some other techniques, e.g., clustering and compression, can work with very little semantic information but are more effective as more semantic information is provided.

While the number of choices for physical database design is confusing, the most significant source of complexity in physical database design is that many decisions are interdependent. For example, the optimal clustering of data items depends on whether or not replication of data items is supported. When the clustering module cannot decide among two advantageous locations for a data item, it might choose to place a copy in each location. Replication can increase system performance by giving the optimizer or retrieval algorithm the choice of which copy to use; however, it must not be used too freely since it can also decrease overall performance if the cost of updating and maintaining multiple copies dominates their benefits [Blakeley, Larson, and Tompa 1986; Drew, King, and Hudson 1990; Hudson and King 1989]. Moreover, the performance effects of replication are different depending on whether or not replicas of collections are declustered over multiple storage media as wholes or by means of partitioning, and whether or not the replicas are clustered in the same way. There is only limited research into making physical database design easier, e.g., [Finkelstein, Schkolnick, and Tiberio 1988; Kao 1986]. Considering the complexity of physical database design, automating physical database design in a comprehensive and extensible way seems to be an extremely fruitful area for database research, in particular in light of the added choices and complexity faced by the database implementor and administrator in extensible and object-oriented database management systems.

5. 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 13 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 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 work [Epstein 1979]. 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.

5.1. 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

Department	Count
Toy	3
Shoe	9
Hardware	7

Figure 13. Count of Employees by Department.

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[13]. 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- 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 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- and hash-based duplicate removal and aggregation algorithms are more appropriate.

5.2. 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

---

[13] 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 look-up 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.

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, including a factor 2 for writing and reading, is[14]

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

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

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

$$2 \times 100 \times 2 + 2 \times 1 \times 1000 \times \left(1 / 10^2 - 1 / 10^3 \right) / (1 - 1 / 10)$$

$$= 400 + 2 \times 1000 \times 0.009 / 0.9 = 420 \ MB$$

---

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

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, e.g., if the SQL *distinct* keyword is used[15]. 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.

5.3. 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 counter-acted), 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 inputs 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

---

[15] 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 paragraph refers to the latter type of query.

$K \times M \times G + (M - K \times C) \times G \geq R'$, meaning $K = \lceil (R' / G - M) / (M - C) \rceil$ partition files will be created. The total I/O cost for hybrid hash aggregation is

$$2 \times R \times L + 2 \times F^L \times \left(R' - (M - K \times C) \times G \right)$$

$$= 2 \times \left(R \times (L + 1) - F^L \times (M - K \times C) \times G \right)$$

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

As for sorting, if an aggregate query requires duplicate removal for the input set to the aggregate function[16], 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.

5.4. A Rough Performance Comparison

It is interesting to note that the performance of both sort- and hash-based aggregation is logarithmic and improves with increasing reduction factors. Figure 14 compares the performance of sort- and hash-based aggregation[17] using the formulas developed above for 100

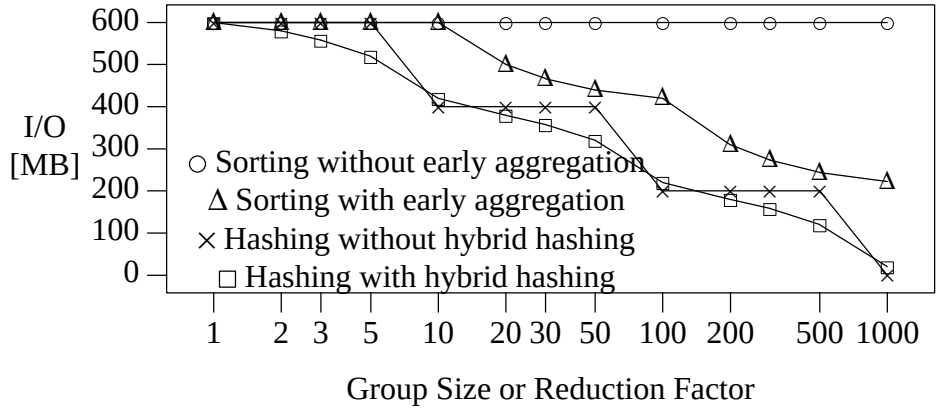


Figure 14. Performance of Sort- and Hash-Based Aggregation.

[16] See an earlier footnote in the previous subsection on counting salaries vs. counting distinct salaries.

[17] Aggregation by nested loops methods is omitted from Figure 14 because it is not competitive for large data sets.

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 14 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 [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 integer 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 14 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 14. The important point is that both their costs are logarithmic with the input size, improve with the group size or reduction factor, and are quite similar overall.

5.5. Additional Remarks on Aggregation

Some applications require multi-level 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.

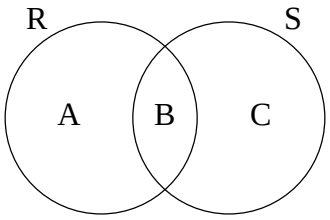
If complex reporting applications are to be supported, the query language should support direct requests (perhaps similar to the syntax suggested above), and the sort operator should be implemented such that it can perform the entire operation in a single sort and one final pass over the sorted data. 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, Schkolnick, and Whang 1987; Hou and Ozsoyoglu 1991; Hou, Ozsoyoglu, and Dogdu 1991; Hou and Ozsoyoglu 1993].

6. Binary Matching Operations

While aggregation is essential for *condensing* 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* in this survey because an input item contributes to the output depending on the its match with one other item. The most prominent among these operations is the relational join. Mishra and Eich have recently written a survey of join algorithms [Mishra and Eich 1992], 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[18], symmetric anti-join[18], intersection, union, left and right differences, and symmetric or



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 15. Binary One-to-One Matching.

anti-difference[18]. Figure 15 shows the basic principle underlying all these operations, namely separation of the matching and non-matching components of two sets, called *R* and *S* in the figure, and production of appropriate subsets[19], possibly after some transformation and combination of records as in the case of a join. 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 (OID's) to physical locations (record identifiers or RID's). Resolving a set of OID's to RID's can be regarded (as well as optimized and executed) as a semi-join of the mapping table and the set of OID's, 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 unioned (for a disjunction) to obtain the list of pointers to items that satisfy the whole query. Moreover, the actual retrieval 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 thesis and the Ingres product [Kooi 1980; Kooi and Frankforth 1982] and a recent study by Shekita and Carey [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, semi-join, and outer join algorithms [Blakeley, Thompson, and Alashqur 1990; Lohman et al. 1991]. Thus, even if relational 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.

---

[18] 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."

[19] 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.

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; Blasgen and Eswaran 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[20], making them problematic for non-trivial queries. The crucial test question is: Does this new technique apply to joining three inputs without interrupting dataflow 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.

6.1. 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

[20] We call it the "Guy Lohman test for join techniques" after the first person who pointed this test out to us.

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 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 forwards and backwards, thus reusing the last page of the previous scan and therefore saving one I/O operation 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/O's 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 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/O's is $\lceil R / (M - C) \rceil \times (S / C) + 1$, which is minimized if $C = M / 2$. 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 retrievals. 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 desigend 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 multi-dimensional. 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. [DeWitt, Naughton, and Burger 1993] demonstrated that index nested loops join can be the fastest join method if one of the inputs is so small and 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. This case is particularly likely if relational queries are written in a navigational style reminiscent of network and hierarchical databases, i.e., the query selects one tuple from one relation and then navigates from relation to relation using join clauses.

If both datasets to be joined are indexed on their join attributes, it is also possible to first join the two indices (since each index contains a join column, this can be done) and then to

retrieve the actual data records from the underlying datasets[21]. The important advantage of this method is that only those records from the underlying datasets that truly contribute to the join result will be fetched from disk. This method was explored by Kooi in his thesis [Kooi 1980] and used in the Ingres product [Kooi and Frankforth 1982]. The second important advantage of this method is that B-tree indices produce sorted scan outputs, which can be joined very efficiently using merge-join (discussed below).

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 retrieval, 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 retrieved in the index on $S.a$. A continuing scan in the index on $S.a$ yields the next possible join attribute value, which is retrieved 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 retrievals 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.

While nested loops algorithms apply to all one-to-one match operations in Figure 15, there is a small problem in right and symmetric outer joins as well as right difference operations, i.e., operations in which non-matching data items from the inner input need to be preserved and included in the output. This problem pertains equally to index nested loops join as to all other variants of nested loops join. Consider a data item in the inner input and the decision that must be made about this item at the end of the outer loop. Since it may have matched with any of the outer data items, it cannot be decided easily whether or not the inner data item actually matched with one of the outer items. Thus, some set data structure must be maintained for the inner input to maintain information about which inner item participated in a match. A number of set data structures exist; typically, a hash-based structure in memory or on disk will be best. On the other hand, since all data items in the inner input must be read at least once (so they can be included in the output), the strongest argument for index nested loops join does not apply and other join algorithms paradigm than nested loops, e.g., the merge-join and hybrid hash join algorithms to be discussed below, might be more efficient algorithm choices. In other words, while nested loops right outer join requires a special auxiliary data structure and its associated processing costs, nested loops for these operations is usually a poor choice in the first place.

---

[21] This methods was mentioned earlier in the section on disk access among the reasons for separating index scans and record retrievals.

Excursus: Is Index Nested Loops Join Sufficient?

Some database systems rely heavily on index nested loops join and do not use memory-based (i.e., sort- and hash-based) algorithms. This approach is fully justified if the predominant access patterns are navigational in nature.[22] For example, in a relational database system, if a single tuple from relation R is selected (by index, typically) and then joined with relation S (again by using an index), memory-based algorithms do not provide any advantage. For applications ported directly from earlier database systems, e.g., network and hierarchical ones, index nested loops join is very important. If, on the other hand, applications and their embedded database queries are written truly in a set-oriented, e.g., the relational, paradigm, i.e., as many loops over sets of data items are "pushed down" from the application code into the database system, memory-based algorithms become very important. Based on a more complete and complex query, the query optimizer will consider index nested loops join as one of the possible algorithm choices; to exploit a set-oriented query execution engine for large data sets, however, the optimizer will also often choose sort- and hash-based algorithms, depending on what algorithms are implemented in the system and are applicable to the query at hand.

There is also an inverse conclusion from this discussion that is often overlooked when "real world" application code is ported from network or hierarchical to relational database systems. Since a relational optimizer will always also consider index nested loops join, an application rewritten in a relational, set-oriented paradigm will run at least as fast as the program logic ported directly from the network or hierarchical database system. In fact, even if the optimizer chooses index nested loops join, the application will run faster if the boundary between application and database management system is crossed only once, not many times. Thus, if a database system does provide efficient set processing, as much of the application logic as possible, in particular loops, should be moved from the application code into the embedded queries and application logic involving loops over sets of data items should be redesigned during the migration to a relational database system. A query optimizer, in a sense, is an expert system to rewrite loops; it is most effective if all pertinent loops are brought into its scope.

Let us clarify this point with an example. Consider an utility company that assembles and prints a number of bills every night. Each bill reflects several services, e.g., gas and electricity, and information about these services provided and consumed during the last billing period must be gathered from a number of record types or relations. In a hierarchical or network database system, the application program written to assemble the bills will loop over all customers, one by one, and gather the information pertaining to a particular customer as this customer’s bill is assembled. If this program logic is ported directly to run on a relational system, index nested loops will play a pivotal role in the query execution plans. On the other hand, if the billing process is rewritten to gather information for all bills at once, not one bill after another, index

[22] It can also be justified if the crucial cost criterion is not the usual resource usage (system throughput) goal but response time for the first result item.

nested loops join will by one, albeit unlikely, algorithm choice among several to be considered by the query optimizer. In most relational database systems, the query optimizer will choose one of the join algorithms discussed in the next two subsections if both join inputs are moderate or large sets of items.

6.2. 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 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 notion 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[23].

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.

---

[23] See an earlier footnote on multiple granule sizes in the section on the architecture of query execution engines.

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 pre-existing B-tree index on the join attribute of the inner input. The result file contains entire tuples from the outer input and record identifiers (RID's, 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 [Blasgen and Eswaran 1976; Blasgen and Eswaran 1977], Kooi [Kooi 1980], and Whang et al. [Whang, Wiederhold, and Saglowicz 1984; Whang, Wiederhold, and Sagalowicz 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 retrieval of data records from index entries as a join [Kooi 1980; Kooi and Frankforth 1982]; this naturally permitted joining two indices or an index with a base relation as in hybrid join.

6.3. 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, Kitsuregawa, and Tanaka 1986; Kitsuregawa, Tanaka, and Motooka 1983; Kitsuregawa, Nakayama, and Takagi 1989; Nakayama, Kitsuregawa, and Takagi 1988; Omiecinski 1991; Schneider and DeWitt 1989; Shapiro 1986; Zeller and Gray 1990; Zeller 1991]. 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; DeWitt et al. 1990; Fushimi, Kitsuregawa, and Tanaka 1986; Kitsuregawa, Tanaka, and Motooka 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[24].

---

[24] Thus, hash join based on partitioning is useful only for joins with equality predicates, often called equi-joins. Fortunately, most joins at least include one equality predicate, typically on keys and foreign keys. For example, the request for employees with salaries higher than their managers

SELECT e.name FROM employee e, employee m WHERE e.mgr-id = m.id AND

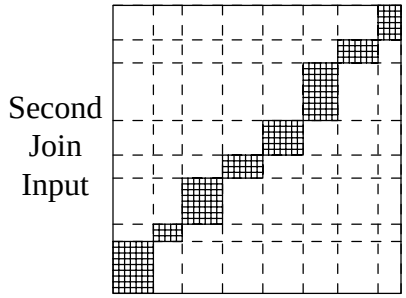


Figure 16. Effect of Partitioning for Join Operations.

Figure 16 shows the effect of partitioning the two inputs of a binary operation such as join into hash buckets and partitions[25]. 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.

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

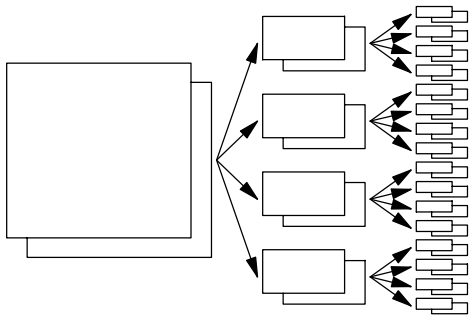


Figure 17. Recursive Partitioning in Binary Operations.

---

e.salary > m.salary

includes a non-equality predicate, but also an equality predicate, and hash join can therefore be used. Some relaxation of this restriction can be found in DeWitt et al.'s band joins [DeWitt, Naughton, and Schneider 1991b].

[25] This figure was adapted from a similar diagram by Kitsuregawa et al. [Kitsuregawa, Tanaka, and Motooka 1983]. Mishra and Eich recently adapted and generalized it in their survey and comparison of relational join algorithms [Mishra and Eich 1992].

partitioning functions. Figure 17 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, Linville, and Shapiro 1994].

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 operations is

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

which can be approximated with $2 \times \log_F (R / M) \times (R + S)$. Thus, the cost of binary hash operations on large inputs is logarithmic; the main difference to the cost of sorting and 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 1993a; Graefe 1994].

6.4. Pointer-Based Joins

Recently, links between data items have found renewed interest, be it in object-oriented systems in the form of object identifiers (OID's) 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 vs. benefit tradeoff between query performance enhancement and maintenance effort. Kooi modeled the retrieval of actual records after index searches as "TID joins" (tuple identifiers permitting direct record access) in his query optimizer for Ingres [Kooi 1980]; 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. modeled joins in a functional way [Batory et al. 1988; Batory, Leung, and Wise 1988], borrowing from research into the database languages FQL [Buneman and Frankel 1979; Buneman, Frankel, and Nikhil 1982], 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 recently analyzed three pointer-based join methods based on nested loops join, merge-join, and hybrid hash join [Shekita and Carey 1990]. 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 non-clustered 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 non-clustered 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, Graefe, and Maier 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. Differently 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 (non-essential links), they were rejected as a relational access method in System R [Chamberlin et al. 1981a] and subsequently in basically all other systems. 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].

6.5. A Rough Performance Comparison

Figure 18 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

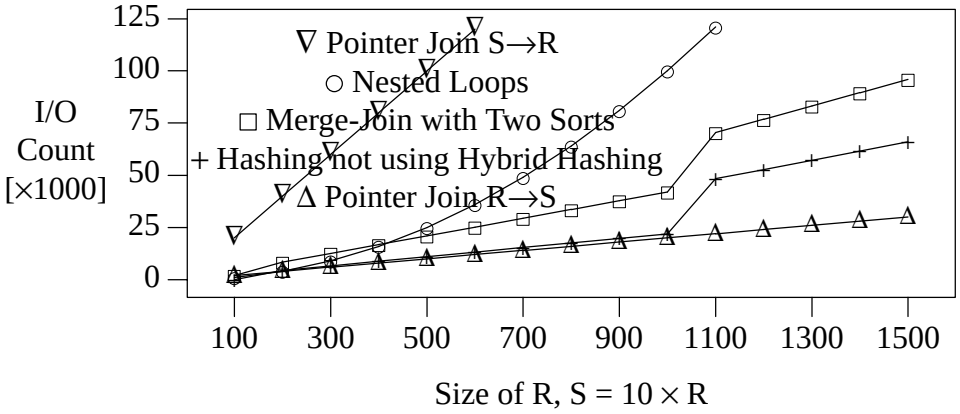


Figure 18. Performance of Alternative Join Methods.

improve and fine-tune the basic algorithms. The relation sizes vary; S is always ten 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 18 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 function, 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 18 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.

7. Universal Quantification

[26]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

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

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-id*'s of *Student*s who have taken at least one course, is called the set of quotient candidates here.

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

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

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.

In this section, we will discuss algorithms for relational division. In order to keep the discussion simple and focused, we restrict ourselves to algorithms for relations without prior structure such as ordering, clustering, indexing, etc. Most of these algorithms can be enhanced by using these structures; we leave it to the reader to invent and consider suitable techniques. Algorithms for relational division 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., a join or semi-join with a third relation) can be advantageous.

Typically, universal quantification can easily be replaced by aggregations[27]. For example, the example query about database courses can be re-stated as "find the students who have taken as many database courses as there are database courses." When specifying this query using aggregate functions, 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. Processing a universal quantification query correctly using aggregation requires an additional semi-join of dividend and divisor unless it is known a priori that referential integrity holds between the dividend's divisor attributes and the divisor, i.e., that there are no 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 non-key attributes.

The complexity of division by counting (aggregation) is immediately obvious in the query plan shown in Figure 19. Notice that all the sort operations before and after the merge-join are

---

[27] Intuitively, *all* universal quantification can be replaced by aggregation. However, we have not found a proof for this statement.

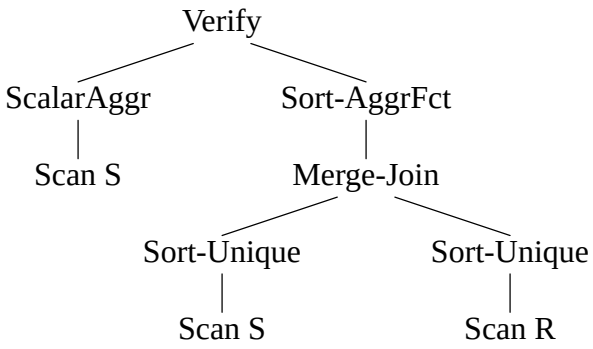


Figure 19. Query Plan for Division by Sort-Based Aggregation.

required because they sort on different attributes: the merge-join's inputs are sorted for the semi-join on the divisor attributes (*course-no*'s in the example), whereas the merge-join's output is sorted and grouped on the quotient attributes (*student-id*'s in the example). The query plan for division using hash-based aggregation is almost the same as the one for sort-based aggregation. Despite their complexity and need for multiple expensive sorts, however, these plans are equivalent to the two direct methods discussed next.

There are four methods to compute the quotient of two relations, a sort-based and a hash-based direct method, and sort- and hash-based aggregation. Table 7 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 [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 not refer to items in the divisor. Thus, neither a preceding semi-join nor explicit duplicate removal step are necessary for naive division. The I/O cost of naive division is the cost of sorting the two inputs

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

Table 7. Classification of Relational Division Algorithms.

plus the cost of repeated scans of the divisor input.

Figure 20 shows two tables, a dividend and a divisor, properly sorted for naive division[28]. 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 includes "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 a 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.

Figure 21 shows the two hash tables used in hash-division. The divisor table on the left contains all divisor tuples and associates a unique sequence number with each item. The quotient table on the right contains quotient candidates, obtained by projecting dividend tuples on their quotient attributes, and a bit map for each item indicating for which divisor tuples there has been a dividend tuple. The fact that "Jack" took only one "Database" course is indicated by the incompletely filled bit map. The "Graphics" course does not appear in either hash table, because it was immediately determined that there was no graphics course in the divisor table.

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 20. Sorted Inputs into Naive Division.

[28] While the attributes in dividend and divisor are typically keys such as course-no's and student-id's, we used course title and student name to make Figure 20 more readable.

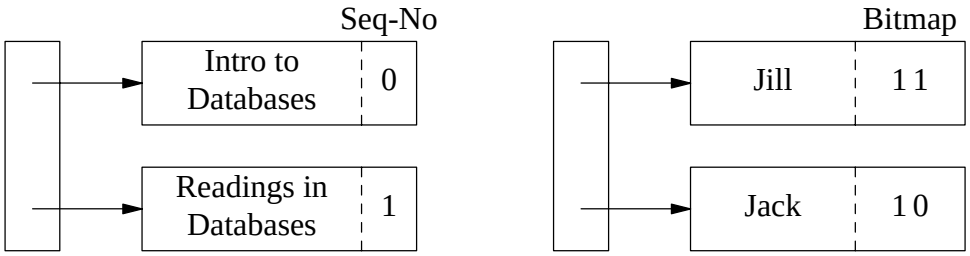


Figure 21. Divisor Table and Quotient Table in Hash-Division.

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 graphics 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 be 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.

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 proposed a generalized division operator called "HAS" that included relational division as a special case [Carlis 1986]. 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 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*s relation 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 results 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 hash table 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: Assume relation R(X,Y) contains 1,000,000 tuples, but only 100,000 tuples if duplicates were removed. Let X and Y be each 100 bytes long (total record size 200 bytes), and assume there are 3,000 unique values of each X and Y. For the standard hash-based duplicate removal algorithm, $100,000 \times 200$ bytes of memory are needed for duplicate removal without use of temporary files. For the modified hash-division algorithm, $2 \times 3,000 \times 100$ bytes are needed for data values, $3,000 \times 4$ bytes for unique sequence numbers, and $3,000 \times 3,000$ bits for bit maps. Thus, the new algorithm works well with 1.66 MB of memory, whereas conventional duplicate removal requires 19.07 MB of memory, or 11.5 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, Schkolnick, and Whang 1987]), optimizer heuristics for choosing this algorithm might be difficult to design and verify.

Hash-division can also be modified into a hash join algorithm with on-the-fly duplicate removal for the probe input. Recall that duplicate removal for the build input of a hash join is easy and inexpensive during construction of a hash table. Recall further that sorting can include duplicate removal and that it is therefore easy and inexpensive to include duplicate removal for both inputs in merge-join processing. In fact, duplicate removal might actually reduce the recursion depth in hashing and the run sizes in sorting [Bitton and DeWitt 1983]. In order to include duplicate removal for the probe input in a hash join, two hash tables are needed, one for each input. The first hash table is the usual one for the build input. The second hash table contains only unique items from the probe input that match with some item in the build input. After the hash table on the build input is complete, each item of the probe input is matched

against the build input, checked for uniqueness in the second hash table of probe items, and then inserted into that second hash table. The performance of this hash join with duplicate removal on both inputs can be expected to be very close to that of two sorts with duplicate removal, with some advantage for hashing because the second hash table contains only items that will actually participate in the join.

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 divides inputs with duplicates tuples 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.

8. Duality of Sort- and Hash-Based Query Processing Algorithms

In this section[29], we conclude the discussion of individual query processing by outlining the

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 $F = \lfloor M / C \rfloor$	Fan-out $F = \lfloor M / C \rfloor$
I/O Optimizations	Read-ahead, forecasting	Write-behind
	Double-buffering input or output	Double-buffering output or input
	Striping merge output	Striping partitioning input
Very large inputs	Multi-level merge	Recursive partitioning
	Merge levels	Recursion depth
	Non-optimal final fan-in	Non-optimal 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

Table 8a. Duality of Sort- and Hash-Based Algorithms.

[29] Parts of this section have been derived from [Graefe, Linville, and Shapiro 1994], 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

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 8 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 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 [Bratbergsengen 1984] and Salzberg [Salzberg 1988]. There exists, however, an important difference. In the sort-based

Aspect	Sorting	Hashing
Algorithm phases	Run generation, intermediate and final merge	Initial and intermediate partitioning, in-memory (hybrid) hashing
Resource sharing	Eager merging Lazy merging	Depth-first partitioning Breadth-first 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

Table 8b. Duality of Sort- and Hash-Based Algorithms.

---

algorithm to the other. The discussion of this section is continued in [Graefe 1993a; Graefe 1994].

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. To summarize, a single-level merge in a sort algorithm is a dual to partitioning in hash algorithms. Figure 22 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. 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 are 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

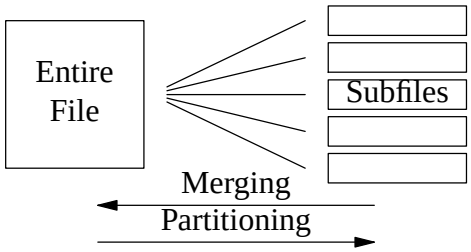


Figure 22. Duality of Partitioning and Merging.

into larger ones. Similarly, partitioning can be repeated recursively, i.e., partition files are re-partitioned, the results re-partitioned, 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, Nakayama, and Takagi 1989] and on histogram-driven recursive hybrid hash join [Graefe 1993a; Graefe 1994].

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, 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 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.[30] If replacement selection is used for run generation, however, the same behavior as with classic hashing is easy to achieve.

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 similar three 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

[30] 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.

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 23 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 1994]. 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. Therefore, 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 must increase as partition files get smaller in hash-based query processing algorithms. For sorting, a suitable choice for such a

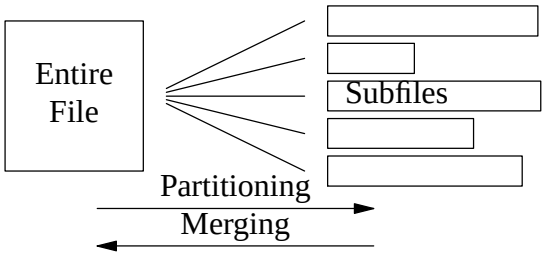


Figure 23. Partitioning Skew.

value is the logarithm of the run size [Graefe 1994]. 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.

Table 9 shows an example calculation of file values. It determines the effectiveness of the final merge step in the optimized merge plan of Figure 6. The unit of the file size and the logarithm's base are immaterial (because they are constant factors in the calculation); value calculations in Table 9 use $log10$ of the indicated file size. The horizontal lines in Table 9 indicate summations or differences. The value of the merge effort is the product of the data volume and the fan-in's logarithm, with $log10(5) = 0.7$. The value increase due to the merge should be equal to the merge effort. However, due to the different input sizes, the value increases by only 59, not 70, or about 85% of the expectation. Notice that a factor 5 between the smallest and largest run sizes is not at all unusual in sorting with optimized merging. If runs from different merge levels are merged in order to minimize the number of merge steps, size differences equal to the fan-in are to be expected. For larger fan-ins, ineffectiveness can even be worse than in Table 9. For example, with a fan-in of 20 and one input run of size 200, one run of size 170, and eighteen runs of size 10, the effectiveness is only about 68%. Extreme cases can even be under 50%.

In hash-based query processing, the corresponding value is the fraction of a partition size relative to the original input size [Graefe 1994]. 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

File	File Size	Item Value log(*Size*)	File Value
Input run 1	50	1.7	85
Input run 2	20	1.3	26
Input run 3	10	1	10
Input run 4	10	1	10
Input run 5	10	1	10
All input runs	100		141
Output file	100	2	200
Value increase			59
Merge effort	100	0.7	70
Ineffectiveness			11

Table 9. Item and File Values in Merging.

fan-out. Thus, preventing or managing skew in partitioning hash functions is very important [Graefe 1993a; Graefe 1994].

Table 10 shows an example calculation of file values for partitioning an input of 100 items into five partitions. For each partition file, the table indicates its size together with the fraction of the input items assigned to that partition and the logarithm of the fraction's denominator. Since the partitions are not of uniform size, these logarithms differ, resulting in the sum of the file values being smaller than the partitioning effort suggests, i.e., the data volume and the logarithm of the fan-out. In fact, due to the partitioning skew, we can say that the partitioning in Table 10 is about 15% less effective than it would be in the ideal case. This is very similar to merging input files of different sizes; however, there is no easy technique equivalent to histograms that ensures that merge input files are of similar sizes. We certainly do not expect size differences among partition files equal to the fan-out in histogram-driven recursive hybrid hash join, except in very rare cases in which a single attribute value dominates an input file and which are easily identified in the histograms. The goal of histogram-driven recursive hybrid hash join is to prevent partitioning skew; thus, while effectiveness of optimized merging is compromised due to the need to merge files of different sizes, histogram-driven recursive hybrid hash join and its N-ary variant do not suffer from suboptimal effectiveness if the histograms indeed permit creating close to evenly sized partition files.

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. The basic idea of bit vector filtering is a large array of bits, which is initialized by hashing items in the first input of a one-to-one match operator and then used to detect items of 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 reduces the

File	File Size	Fraction of Input	Item Value log(*Den.*)	File Value
Input file	100	1/1	0	0
Partition 1	10	1/10	1	10
Partition 2	10	1/10	1	10
Partition 3	10	1/10	1	10
Partition 4	20	1/5	0.7	14
Partition 5	50	1/2	0.3	15
All partitions	100			59
Partitioning effort	100		0.7	70
Ineffectiveness				11

Table 10. Item and File Values in Partitioning.

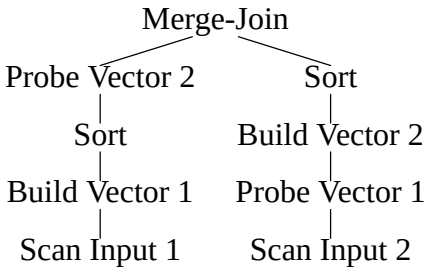


Figure 24. Merge-Join with Symmetric Bit Vector Filtering.

cost of processing the second input is effectively as in *(hh. It can also be used symmetrically in merge-joins as shown in Figure 24. 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 multi-level 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 operator 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 10 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

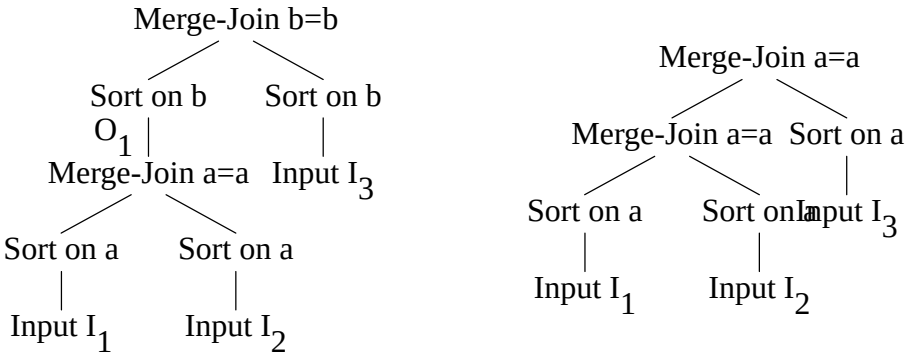


Figure 25. The Effect of Interesting Orderings.

intermediate join results. For joining three relations, as shown in Figure 25, 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.

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. Such set operations and join queries can be executed effectively and efficiently by a hash join algorithm that recursively partitions N inputs concurrently. The recursion terminates when $N - 1$ inputs fit into memory and the N^{th} input is used to probe $N - 1$ 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 26 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 that represents the "many" side of a many-to-one relationship or on a

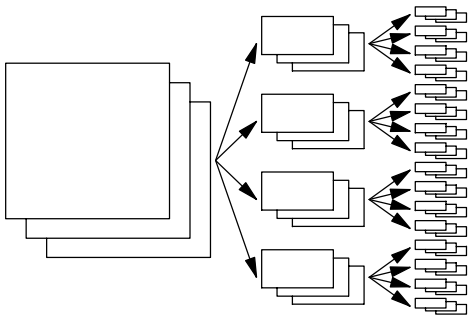


Figure 26. Partitioning in a Multi-Input Hash Join.

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 1993c]. 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. Similarly, duplicate removal during sorting is trivial for all merge-join inputs, while duplicate removal for *all* hash join inputs requires symmetric hash joins.

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 multi-level 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 non-uniform data distributions. However, our experiments with histogram-driven recursive hybrid hash join [Graefe 1993a; Graefe 1993b] indicate that data and hash value skew can typically be controlled and often even be exploited.

The important conclusion from the dualities discussed in this section 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.

For a few additional issues, Table 11 summarizes the comparison of sort-merge-join and histogram-driven recursive hybrid hash join in [Graefe 1994]. The conclusion from this comparison is that sort-merge-join is mostly obsolete with only two exceptions. The first exception is merge-joining items from multiple indices in the order of data values or in the order of hash values. The other exception is a query whose output must be sorted on a join key *and* whose join result is so large that sorting the join output is more expensive than sorting the two join inputs.

9. 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 very 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 non-trivial 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 analyzed several database

Aspect	Sort-Merge-Join	Hybrid Hash Join
Input sizes	Each input determines its number of merge levels	Recursion depth depends on build input only; faster when the build input is smaller than the probe input or dynamic role reversal is employed
Data and hash value skew	No significant performance effect	Performance loss in naive hybrid hash join; performance gain if histograms are used
N set operations or N joins on the same attribute	*Interesting orderings* save sort operations for all N−1 intermediate results	N-ary join saves partitioning all N−1 intermediate results
On-the-fly compression	Applicable, but must be consistent for all inputs	Decided locally within each partitioning step
Bit Vector Filtering	Applicable; although use in each merge level is cumbersome to implement and prevents optimized merging	Easy to implement, increasingly effective in deeper recursion levels
Effectiveness	Not optimal because runs of different sizes must be merged	Can be near-optimal if histograms are used
Asynchronous I/O	Effective read-ahead requires forecasting	Write-behind means flushing blocks present in the buffer; simple to make effective
Disk arrays	Read-ahead and forecasting require extra data structures	Write-behind continues to be simple
Disk caches	Read-ahead is cumbersome to implement and to control; implementations are probably not portable	Write-behind continues to be simple

Table 11. Comparison Summary.

algorithms and found that their cost functions exhibit steps when plotted over available buffer space, and suggested that buffer space should be allocated at the low end of a step for the least buffer use at a given cost [Sacco and Schkolnik 1982; Sacco and Schkolnik 1986]. Chou and DeWitt took this idea further by combining it with separate page replacement algorithms for each relation or scan, following observations by Stonebraker on operating system support for database systems [Stonebraker 1981], and with load control, calling the resulting algorithm DBMIN [Chou 1985; Chou and DeWitt 1985]. Faloutsos et al. generalized this goal and used the classic economic concepts of decreasing marginal gain and balanced marginal gains for maximal overall gain [Faloutsos, Ng, and Sellis 1991; Ng, Faloutsos, and Sellis 1991]. Their measure of gain was the reduction in the number of page faults. Zeller and Gray designed a hash join algorithm that

adapts to the current memory and buffer contention each time a new hash table is built [Zeller and Gray 1990; Zeller 1991]. Most recently, Brown et al. have considered resource allocation tradeoffs among short transactions and complex queries [Brown et al. 1992].

Schneider was 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) [Schneider 1990; Schneider and DeWitt 1990]. Among his most interesting findings are that memory requirements for right-deep plans might actually be comparable to those of left-deep plans if bit vector filtering (discussed later) is used effectively [Schneider 1991]. This work has recently been extended by Chen et al. to bushy plans interpreted and executed as multiple right-deep subplans [Chen et al. 1992].

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 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 only the final merge is left, but no output has been produced yet. Figure 27 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 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*

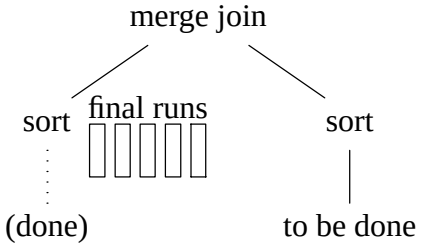


Figure 27. The Stop Point During Sorting.

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 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 multiples 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 proportial to its input size. For example, if two merge-join inputs are of the same size and 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 ¼ of memory while the run-generation (quicksort) should use ½ 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 28 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 top-most operator in a query evaluation plan does not have a comsumer operator with which it shares resources; therefore, this operator should use depth-first partitioning in order to provide better response time, i.e., earlier delivery of the first data item.

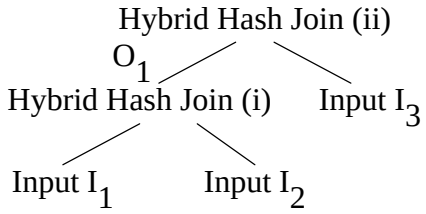


Figure 28. Plan for Joining Three Inputs.

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 continue to 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 multi-processor systems is not entirely understood yet. While all considerations discussed above apply in principle, multi-processors 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 is 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., 3-9 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

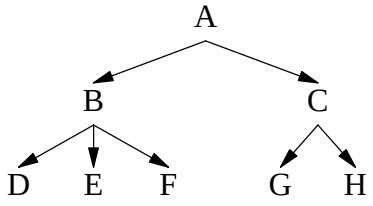


Figure 29. A Decision Tree of Partial Plans.

would anticipate possible outcomes of executing subplans and provide multiple alternative subsequent plans. Figure 29 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. The implementation of such plans is quite simple using improved *choose-plan* operators as implemented in the Volcano system [Graefe and Ward 1989; Graefe 1993c], as shown in Figure 30. These operators first execute zero or more subplans and then decide which of two or more alternative plans to execute. However, realization of sophisticated query optimizers able to produce such plans will require further research, e.g., into determination of when separate cases are warranted and into limitation of the possibly exponential growth of the number of subplans.

Even for moderately complex subplans with about 10 operators, optimal resource allocation during execution time is an important issue as shown by Schneider et al. and by Chen et al. for right-deep plans [Chen et al. 1992; Schneider 1990; Schneider and DeWitt 1990]. In order to allocate resources such as memory, processors, and disks to operators, processes, and queries most fairly and productively, a uniform and comparable measure of their value must be defined. We have defined a productivity measure for this purpose that we introduce here in several steps[31]. This measure is closely related to the concept of "values" introduced in the previous section. It is important that this measure permits comparisons for sort- and hash-based query processing algorithms and their three distinct algorithm phases[32] including resource effectiveness of algorithms in different phases, and resource effectiveness of multiple concurrently executing

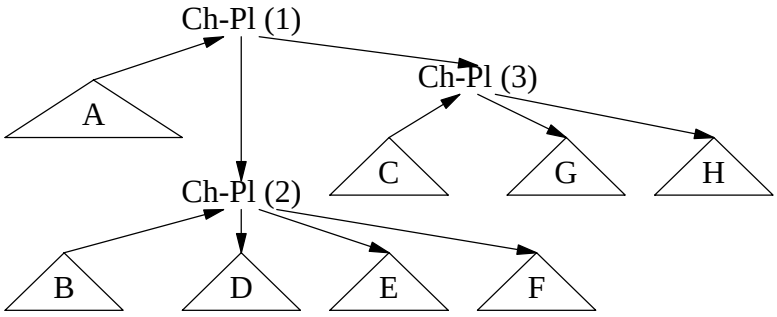


Figure 30. Implementation using Choose-Plan Operators.

[31] This productivity measure was defined in joint research with Diane Davison.

[32] Initial run generation, intermediate and final merging for sorting and initial and intermediate partitioning followed by in-memory or hybrid hashing for hash-based algorithms. The intermediate phases may be missing for modest-size inputs.

plans within a bushy query evaluation plan or in a multi-user environment. The measure uses bandwidth for both I/O and CPU processing, measured in bytes or pages per unit of time. The algorithm bandwidth is a measure of the rate at which the algorithm can move records or bytes through the slowest part of the system (processor, disk, or network); thus, this bandwidth considers both the architectural constraints and algorithm complexity and behavior. CPU performance measures based on MIPS numbers are ignored.

We start the discussion for a single sort and progress to complex plans. Our model compares the effectiveness of a given sort with a fictitious external sort that uses only quicksort with a single page of memory and numerous levels of binary merges. The merge levels used in this sort are called the *required binary merge levels* of any sort and can be calculated as $\log_2(R)$ for an input of R pages. The effectiveness of a real quicksort, replacement selection, or merge step is measured in how many of the required binary merge levels are performed or replaced in that processing step. If a real quicksort may use more memory than one page, say M pages, it makes several binary merge levels unnecessary. The size of the initial runs will be M pages, while the run size in the fictitious sort would reach M pages only after $\log_2(M)$ merge levels. Therefore, we say that the replaced binary merge levels of a quicksort with M pages is $\log_2(M)$. For initial-run generation based on replacement selection, in which the runs are about twice the size of memory, the replaced binary merge levels are $\log_2(M) + 1$. In a merge step with fan-in F, the size of the output run is F times larger than the input. In order to achieve the same increase in run size with binary merges, $\log_2(F)$ binary merge levels would be required. Therefore, we say that the replaced binary merge levels of a merge step with fan-in F is $\log_2(F)$.

The required binary merge levels multiplied with the data volume, i.e., the input size, are called the *required contribution*. The replaced binary merge levels multiplied with the data volume affected by a quicksort or a merge step (not the total input size) are called *performed contributions*. The important aspects of required and performed contributions are that the required contribution is presumed to be independent of the sort algorithm used (if the algorithm always merges the smallest files and has $N \log N$ complexity) and that the performed contributions always add up to the required contribution, even if memory allocation and fan-in are varied at appropriate points during the sort due to changing resource contention in the system.[33] Thus, performed contributions are a direct measure of the value of a processing step to the completion of a sort, a database operator such as a merge-join, a process participating in a parallel query evaluation plan, or an entire query evaluation plan.

---

[33] These assumptions ignore the reduced effectiveness of merging if the input runs are of different sizes (see the previous section) and make our productivity measure only an approximation. However, these assumptions create some desirable properties that make resource allocation relatively simple and elegant.

The performed contribution of an operation divided over its required amount of time defines the *productivity* of the operation. This productivity measure is equal to the replaced binary merge levels multiplied with the bandwidth of the operation, which links this discussion to the paragraphs on tuning based on bandwidth and latency in the earlier section on sorting. Thus, productivity relates an operation's value to the length of time it requires that resources be allocated to the operation.

In complex plans, algorithm phases often overlap. For example, the final merge phases of two sorts feeding into a merge-join overlap with each other and the run-generation phase of a sort consuming the join output. We call this combination of concurrent algorithm phases a phase of a complex plan, or a *plan phase*. Since producers and consumers work under flow control, be it as iterators within a single process or in a multi-process environment, the bandwidths of producers and consumers are proportional to their respective data volumes, e.g., the two final merges and the run generation in the example above. A plan phase's contribution is the sum of the contributions of the participating algorithm phases. A plan phase's contribution divided by its execution time is the plan phase's productivity.

For hash-based query processing algorithms, the required and replaced binary merge levels are defined similarly as $\log_2(R)$ for a (build) input of size R, $\log_2(M)$ for a hash table of size M, and $\log_2(F)$ for a partitioning step with fan-out F. Bandwidth and productivity of hash-based algorithms is determined just as for sort-based algorithms. This defition correctly reflects the fact that hash-based matching is much more efficient than sort-based matching (merge-join) if the build input is significantly smaller than the probe input.

For resource allocation, *marginal productivity* is defined for each resource as the additional productivity gained by one additional resource unit. It is assumed that marginal productivities are *positive*, i.e., that an additional unit of a resource increases the replaced binary merge levels or improves an operations bandwidth, and that they are *monotonically non-increasing*, i.e., that the positive effect of adding two units of the same resource is at most twice as large as the effect of one unit of that resource. Considering the logarithmic nature of replaced binary merge levels and the linear performance effect of CPU's and disks, these assumptions seem justified. However, it might require that some resources be "bundled," e.g., each disk drive "comes with" some amount of memory for read-ahead and write-behind buffer space. Otherwise, increasing the number of disk drives might decrease the amount of memory available for run generation or hash tables, thus decreasing the number of replaced binary merge levels.

Marginal gains can be used to maximize the combined performed contributions of all operations active in a computer system by appropriate resource allocations to operations, processes, and query plans. Among two contenders for a resource such as a unit of memory, the resource is allocated to the requestor that can derive the higher (marginal) gain from it. Thus, optimal resource allocation is achieved if the marginal gains are *balanced* among all contenders for each resource, where multiple contenders for a resource may be query plan operators in the same plan or in different plans. The latter point is important, because it permits resource allocation within a single query as well as among multiple concurrent queries. If different priorities have been assigned to concurrent queries, these can be reflected as factors in the

productivity comparisons. Among the open issues to be resolved for resource allocation based on marginal gains are (i) the impact of the assumption that the required contributions are constant and equal to the performed contributions, both on the efficiency of individual merge or partitioning algorithms and on the overall system throughput; (ii) how interdependent allocation of different resources is, i.e., whether or not query processing algorithms can benefit from an additional unit of one resource such as processing power without an additional unit of another resource such as memory; (iii) how much resource allocation is affected by the logarithmic approximation of algorithms' costs; (iv) whether or not resources can be "bundled" effectively such that the two assumptions on marginal gains can be justified; and (v) how this new productivity measure can be generalized from queries and all their algorithm and plan phases to environments executing both short transactions and complex queries. We are currently working towards answers to these questions.

10. 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 speed-up and scale-up, and speed-up results are frequently used to demonstrate the accomplishments of a design and its implementation. speed-up considers additional hardware resources for a constant problem size; linear speed-up is considered optimal. N times as many resources should solve a constant-size problem in $1 / N$ of the time. speed-up can also be expressed as parallel efficiency, i.e., a measure of how close a system comes to linear speed-up. For example, if solving a problem takes 1,200 seconds on a single machine and 100 seconds on 16 machines, the speed-up 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 scale-up, in which the problem size is altered with the resources. Linear scale-up is achieved when N times as many resources can solve a problem with N times as much data in the same amount of time. Scale-up can also be expressed using parallel efficiency, but since speed-up and scale-up 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 speed-up 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 speed-up can frequently be more than linear, or super-linear. 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 speed-up. 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 speed-up super-linear.

10.1. Parallel vs. 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, meta-data (catalogs), query processing, etc. 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 performs 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 have discussed distributed database systems in much more detail [Ozsu and Valduriez 1991a; Ozsu and Valduriez 1991b]. 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. 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, Kitsuregawa, and Tanaka 1986; Kitsuregawa, Tanaka, and Motooka 1983], and Volcano [Graefe 1990b; Graefe 1993c; Graefe and Davison 1993], and products, e.g., Tandem’s NonStop SQL [Zeller and Gray 1990; Zeller 1991]. Englert Tandem 1989 NonStop SQL Near-Linear .], Teradata’s DBC/1012 [Neches 1984; Neches 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.[34] 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 database 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 CPU's and disks and a large shared memory. Stonebraker 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 [Stonebraker 1986b]. 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].

Moreover, in all but shared-everything systems, distances between pairs of nodes may differ when measured in bandwidth or latency, and nodes may differ from one another. These differences may be in the nodes' hardware or their software, e.g., in storage capacities (size and reliability), processing power (CPU and disk bandwidths, memory), connections to the "outside world," hardware architecture and model, data representation, software capabilities, and software versions. It seems that for a real system, some heterogeneity must be permitted, e.g., the type and number of disk drives. On the other hand, radical differences may create a plethora of difficulties that are impossible to overcome. For example, the problems in query optimization and resource scheduling due to heterogeneity in parallel database systems have basically not been explored to-date.

10.2. Forms of Parallelism

There are several forms of parallelism that are interesting to designers and implementors of query processing systems. *Inter-query* parallelism is a direct result of the fact that most database management systems can service multiple requests concurrently. Multiple queries (transactions) can be executing concurrently within a single database management system. In this form of parallelism, resource contention 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 is only now developing. *Inter-operator* parallelism is basically pipelining, or parallel execution of different operators in a single query. For example, the iterator concept discussed earlier has also been

[34] In some organizations, two different database management systems may run on the same (fairly large) computer. Their interactions could be called non-distributed 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.

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.

Inter-operator parallelism can be used in two forms, either to execute producers and consumers in pipelines, called *vertical inter-operator* parallelism here, or to execute independent subtrees in a complex, bushy query evaluation plan concurrently, called *horizontal inter-operator* 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 *intra-operator* 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 represent sequences, for example time series in a scientific database management system, partitioning into subsets to be operated upon independently is not feasible or requires additional synchronization when putting the independently obtained results together.

Both vertical inter-operator parallelism and intra-operator parallelism are used in database query processing to obtain higher performance. Beyond the obvious opportunities for speed-up and scale-up 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 intra-operator, partitioning-based parallelism, load balance and performance are optimal if the partitions are all of equal size; however, load balance can be hard to achieve if value distributions in the inputs are skewed.

10.3. 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 31, 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

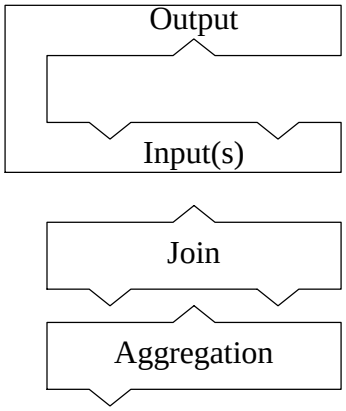


Figure 31. Bracket Model of Parallelization.

which then controls execution; network I/O on the receiving 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 co-routine 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

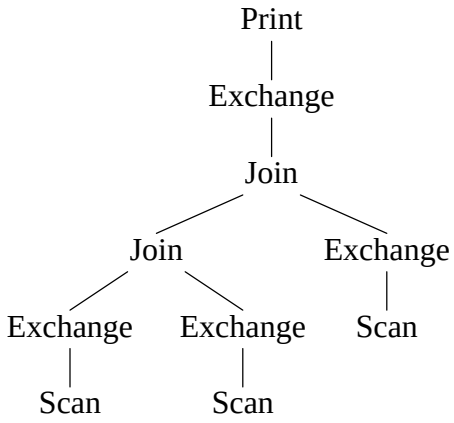


Figure 32. Operator Model of Parallelization.

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 inter-process 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 32 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 not affected at all 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 inter-process 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; Graefe 1994] without regard to the execution environment.

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 33 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 1993c].

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, intra-operator parallelism based on partitioning requires that the processes exchange data among themselves outside of the stream paradigm.

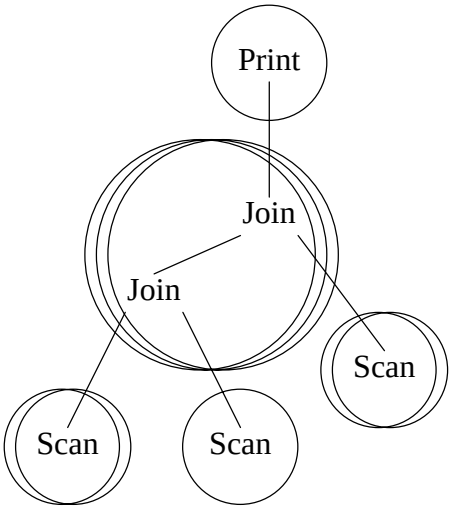


Figure 33. Processes Created by Exchange Operators.

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. [Keller, Graefe, and Maier 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 intra-operator data exchange for intra-operator parallelism.

10.4. Load Balancing and Skew

For optimal speed-up and scale-up, pieces of the processing load must be assigned carefully to individual processors and disks to ensure equal completion times for all pieces. In inter-operator 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 inter-operator parallelism. In intra-operator 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, Naughton, and Schneider 1991a; Hua and Lee 1991; Kitsuregawa and Ogawa 1990; Lakshmi and Yu 1988; Lakshmi and Yu 1990; Omiecinski 1991; Seshadri and Naughton 1992; Walton 1989; Walton, Dale, and Jenevein 1991; Wolf, Dias, and Yu 1990; Wolf et al. 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 permanent 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.

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 re-hashing in hash partitioning or on quantile adjustment in range partitioning. Since hash partitioning tends to create fairly even loads and 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, Naughton et al. 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 [DeWitt, Naughton, and Schneider 1991a; Seshadri and Naughton 1992]. 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. A skew of 1.0 indicates a perfectly even distribution. Figure 34 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., CPU's 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 become 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.

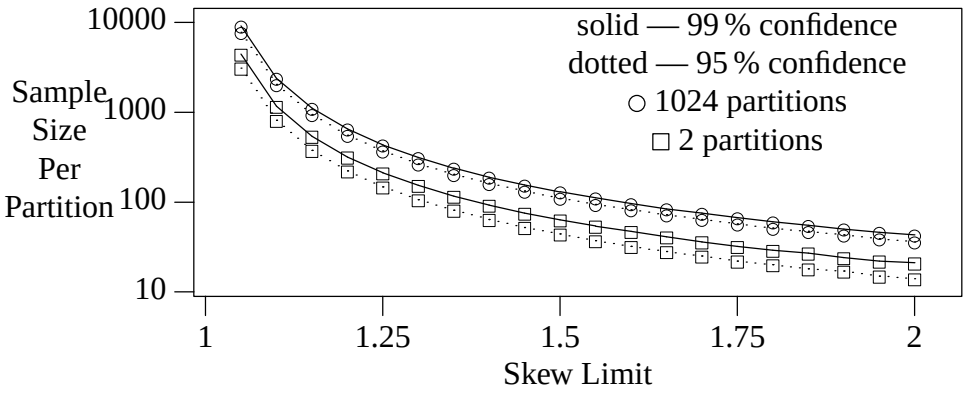


Figure 34. Skew Limit, Confidence, and Sample Size per Partition.

10.5. Tuning a Parallel System

Beyond skew, the major impediments to effective use of parallelism in database query processing are *control overhead*, in particular initialization delays, *interference* on lower software and hardware levels, and *synchronization delays*.

Control overhead was one of the major obstacles to obtaining linear speed-ups in early database machine designs, for example DIRECT [Boral et al. 1982]. The main issue here is that the number of control messages should be linear to the number of processes, not with the number of data pages or items. If distribution of initiation messages is a problem and broadcasting is not available or not practical, a linear number of messages can be executed in a logarithmic number of single node-to-node messages using a tree-shaped propagation scheme, as proposed for example for a highly parallel implementation of the Gamma database machine [Gerber 1986; Gerber and DeWitt 1987].

Another possible approach to reducing control overhead is to use not only primed processes (i.e., a pool of processes waiting until work packets arrive), but primed processes with primed connections. Instead of allocating processes and then establishing connections between them, a pool of process groups is used, each group already connected in a pattern typically found in parallel query processing. This idea clearly has a lot of promise for massively parallel systems but it requires determining the size of process groups and suitable "typical" connections. Furthermore, the query optimizer must be designed to exploit such preconnected process groups.

Interference can occur both on a hardware level and in lower software levels. The prime example for hardware interference is bus contention in shared-memory multi-processors, the reason for the limited scalability of single-bus shared-memory architectures. Lower software levels can also introduce bottlenecks and contention, for example a buffer manager with its residency lookup table or a bit map for disk page allocation. A study of software interference in a more general sense has been discussed by Fontenot [Fontenot 1989].

Some implementations of intra-operator parallelism exploit the relatively inexpensive communication and synchronization via shared memory even within a single, parallel algorithm. For example, the XPRS implementation of hash join seems to use only a single hash table [Hong and Stonebraker 1993]. All processes may insert tuples into this hash table, and all processes may probe into the hash table. The advantage of this implementation is that the two input need not be repartitioned prior to a join; on the other hand, care must be taken to ensure that the shared data structure is always properly locked. In order to minimize interference, it might be necessary to lock the hash table on the bucket level.

As an example, Figure 35, taken from [Graefe and Thakkar 1992], shows the effect of tuning and removing interference among processes on a shared-memory machine executing a parallel sort of 100 MB (10^6 records of 100 B) with an equal number of processors and disks in each measurement. The time measurements are shown using solid lines and refer to the labels on the left. The speed-ups are shown with dashed lines and refer to the labels on the right. The initial times and speed-ups are marked with □'s while the final ones are marked with Δ's. The ideal, linear speed-up is shown by the dotted line. It is immediately obvious from the solid lines that the final times are significantly lower than the initial ones, demonstrating the effect of the

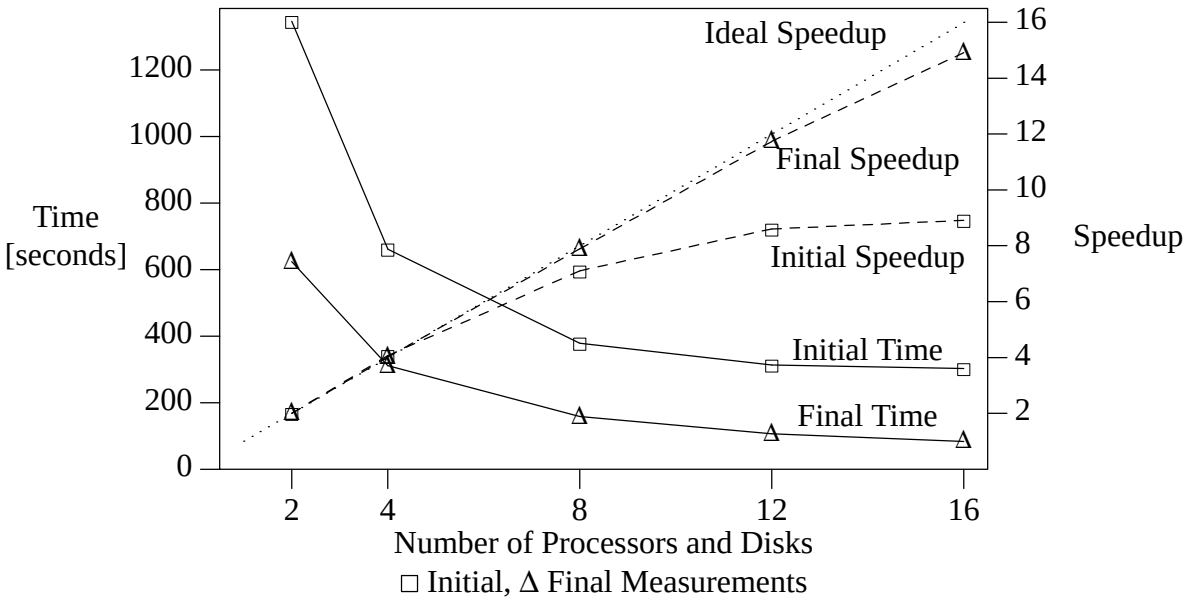


Figure 35. Tuning Effectiveness for a Parallel Algorithm.

tuning measures. For two to eight processors and disks, the observed performance improvements by a factor slightly more than two are largely due to increased cluster size and reduced I/O cost. Beyond eight processors and disks, the dashed lines indicate that the modifications and adjustments also improved the speed-up which had been completely unsatisfactory with the initial software. For sixteen processors and disks, the fully tuned software performed about 3½ times better than the original version. A comparison of the dashed and dotted lines shows very close to linear speed-up with the fully tuned software. Thus, tuning improved the parallel behavior as well as the absolute performance. The main reason why the speed-up could be improved was the removal of a latch contention bottleneck from the buffer manager, i.e., interference of multiple processes using a shared resource.

Synchronization delays can also be induced by hardware and software. Software delays may occur if multiple processes in a pipeline operate with different throughput and require the standard producer-consumer synchronization. Synchronizing multiple caches can introduce significant delays, in particular since today's cache sizes can almost be thought of as making a shared-memory machine into a distributed-memory machine. Thus, effective load balancing through a single run-queue, usually considered one of the advantages of shared-memory machines, might be counter-productive if processes (and their cache residency set) tend to migrate too much or too often. Our recommendation is to make processes "sticky," i.e., to try but not to force process-to-processor affinity [Graefe and Thakkar 1992]. The importance of stickiness in process-to-processor allocation is increasing because current hardware trends indicate much faster growth in CPU performance than in bus bandwidth [Markatos and LeBlanc 1992]. It might be very interesting to separate bus latency and bandwidth and to include techniques similar to read-ahead in these analyses.

10.6. Architectures and Architecture-Independence

[35]Many database research projects have investigated hardware architectures for parallelism in database systems. Stonebraker compared shared-nothing (distributed-memory), shared-disk (distributed-memory with multi-ported 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 [Stonebraker 1986b]. 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 that 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.

Bhide and Stonebraker compared architectural alternatives for transaction processing [Bhide 1988; Bhide and Stonebraker 1988] 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. [Pirahesh et al. 1990] and Boral et al. [Boral et al. 1990], but none of these authors elaborate on the idea's generality or potential. Kitsuregawa and Ogawa's new database machine SDC uses multiple shared-memory nodes (plus custom hardware such as the Omega network and a hardware sorter) [Kitsuregawa and Ogawa 1990], although the effect of the hardware design on operators other than join is not evaluated in [Kitsuregawa and Ogawa 1990].

Customized parallel hardware was investigated but largely abandoned after Boral and DeWitt's influential analysis [Boral and DeWitt 1983] that compared CPU and I/O speeds and their trends. They 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, Aoki, and Seltzer 1988; Stonebraker et al. 1988]. Its designers believe that modern bus architectures can handle up to 2,000 transactions per second. 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. 1988] requires

---

[35] Much of this section has been derived from [Graefe and Davison 1993; Graefe et al. 1994].

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 CPU's 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. [Richardson, Lu, and Mikkilineni 1987] performed an analytical study of parallel join algorithms on multiple shared-memory "clusters" of CPU's. 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 CPU's (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 CPU's 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 developed the concept of "local sufficiency" [Wong and Katz 1983] 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, Lee, and Chang 1986; Ghandeharizadeh and DeWitt 1990a; Hsiao and DeWitt 1990; 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, Tzou, and Graham 1988], the Wisconsin Multicube [Goodman and Woest 1988], and the Paradigm project [Cheriton, Goosen, and Boyle 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 36 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 between 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

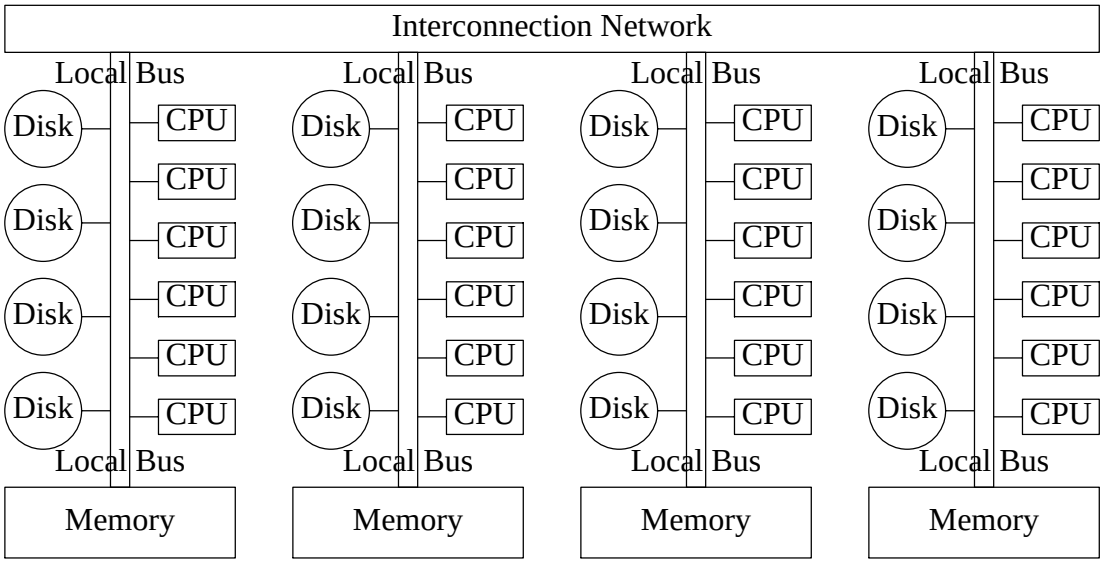


Figure 36. A Hierarchical-Memory Architecture.

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, i.e., the ability to add, remove, and replace individual units. 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 CPU's 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 CPU's into one with 8×8 CPU's.

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 CPU's either in 100 nodes or in 10 nodes of 10 CPU's each, it is much faster to distribute query plans to all CPU's and much easier to achieve reasonably balanced loads in the second case than in the first case. Within each shared-memory parallel node, load 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]. 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.

11. 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; 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 concluded that parallelism is no substitute for effective and efficient indices [Boral and DeWitt 1983]. 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.

11.1. 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 transfer 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.

11.2. 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, Bitton, and Wilkinson 1988; Bitton-Friedland 1982; Graefe 1990a; Iyer and Dias 1990; Kitsuregawa, Yang, and Fushimi 1989; Lorie and Young 1989; Menon 1986; Salzberg et al. 1990; Young and Swami 1992]. Continuing our discussion of Table 2, 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). As sequential input or output restrict the throughput of parallel sorts, we assume a multiple-input multiple-output parallel sort here, and 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. However, if detailed quantiles are available, they can be used not only for the partitioning step but also for external bucket sorting within each site [Knuth 1973; Young and Swami 1992].

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 (i) multiple consumers feed multiple producers, and (ii) each producer produces a sorted stream and each consumer merges multiple sorted streams, and (iii) some key-based partitioning rule is used other than range partitioning, i.e., hash partitioning, and (iv) flow control is enabled, and (v) the data distribution is particularly unfortunate.

Figure 37 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 37 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,

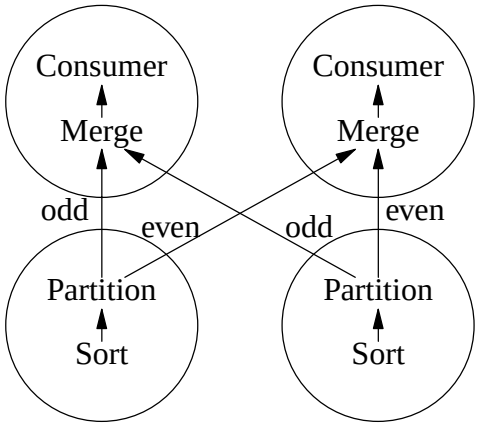


Figure 37. Scenario with Possible Deadlock.

all 500 for one consumer) before they will send their first item to the other consumer. The data exchange buffer needs to hold 1000 items at one point of time, 500 on each side of Figure 37. If flow control is enabled and 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 37) 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 focussed 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.

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 38. 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 37 is that 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 39 illustrates the problem with

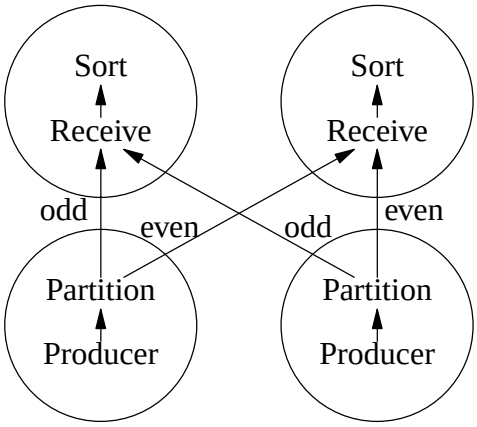


Figure 38. Deadlock-Free Scenario.

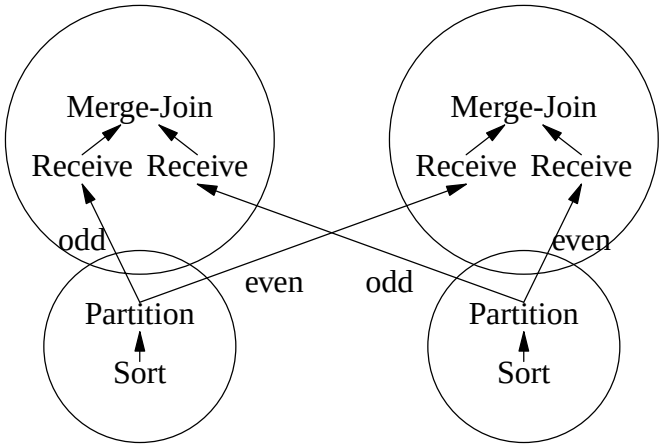


Figure 39. Deadlock Danger Due to a Binary Operation in the Consumer.

a merge-join operation executed in two consumer processes. Notice that the left and right producers in Figure 39 are different inputs of the merge-join, not processes executing the same operators as in Figure 37 and Figure 38. The consumer in Figure 39 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 37. Again, if the data exchange buffer (flow control slack) is too small, deadlock will occur. Similar to the deadlock avoidance tactic in Figure 38, deadlock in Figure 39 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 39, namely moving only one of the sort operators, not both, into the consumer

processes[36].

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 (see below) could be used [Epstein, Stonebraker, and Wong 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 join. 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 (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[37] 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

---

[36] This scenario and this solution were contributed by Andy Kashyap.

[37] This solution was suggested by Bob Gerber and Dave Clay.

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 simultaneously 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. Large degrees of parallelism may impede performance because they increase the number of merge levels. Figure 40 shows how the number of merge levels growths 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].

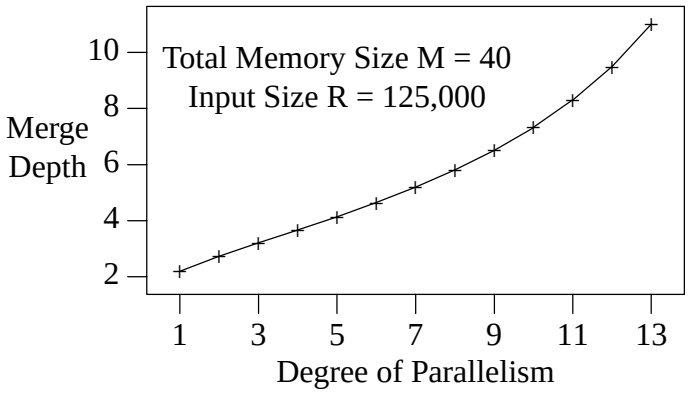


Figure 40. Merge Depth as a Function of Parallelism.

11.3. 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 when resolving 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.[38] The advantage of this technique 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.

11.4. 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 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; Hong and Stonebraker 1993; Stonebraker, Aoki, and Seltzer 1988; Stonebraker et al. 1988] 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

---

[38] 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 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 it was cheaper to send a small input to a small number of sites than to partition both a small and a large input. Fragment-and-replicate query decomposition has been used in some parallel implementations of the Oracle relational product [Tseng and Reiner 1993], in which a complex query involving multiple tables is partitioned according to the stored partitions of only one large table, which is called the query's "driving table." 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. 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 subsection. Furthermore, there are probabilistic variants of this idea that use bit vector filtering instead of semi-joins, discussed later in its own subsection.

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

$$R \ JOIN \ S = R \ JOIN \ (\ S \ SEMIJOIN \ \pi_A R \ )$$

$$= (\ R\ SEMIJOIN\ \pi_A(\ S\ SEMIJOIN\ \pi_A R\ )\ )\ JOIN\ (\ S\ SEMIJOIN\ \pi_A R\ ) \qquad \text{(a)}$$

$$= \left(\ R\ \overline{SEMIJOIN}\ \pi_A\left(\ S\ \overline{SEMIJOIN}\ \pi_A R\ \right)\right) JOIN\ (\ S\ SEMIJOIN\ \pi_A R\ ), \qquad \text{(b)}$$

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 non-matching 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 non-matching[39] 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, 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 41, processors are organized into rows and columns. One input relation is partitioned over rows and partitions are

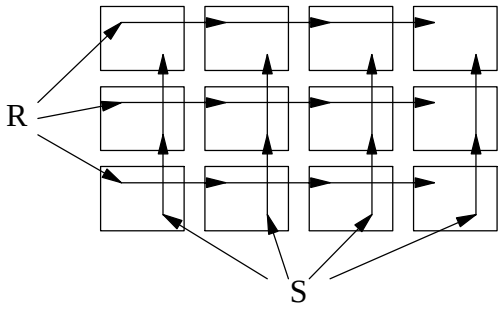


Figure 41. Symmetric Fragment-and-Replicate Join.

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

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.

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 to the first site that will be required for the join computation. 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.

11.5. 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 independently 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.

12. Non-Standard Query Processing Algorithms

In this section, we briefly review the query processing needs of data models and database systems for non-standard 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 control-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 control-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 and McKenna 1993; Graefe et al. 1994], optimization rules that can move an unusual database operator in a query plan are easy to implement. For a sampling operator, there might be a rule that transforms sampling a query result into querying a sample.

12.1. 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 42 shows a

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 42. Nested Relation and Equivalent Flat Relations.

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 depends on another entity as for order entries in Figure 42. In general, nested sub-tuples 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, that many frequent join operations can be avoided, and that 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, Ozsoyoglu, and Matos 1987; Roth, Korth, and Silberschatz 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 a 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 42 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 present a recent investigation of "keying methods" for this purpose [Ozsoyoglu and Wang 1992].

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 relation is a relation, any relational calculus and algebra expression should be permitted for it. In the example of Figure 42, 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. [Rosenthal, Rich, and Scholl 1991].

Deshpande and Larson 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" [Deshpande and Larson 1992]. 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 with 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. [Kuespert, Saake, and Wegner 1989; Saake 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, Paul, and Schek 1986; Deshpande and van Gucht 1988; Hafez and Ozsoyoglu 1988; Ozsoyoglu and Wang 1992; Scholl, Paul, and Schek 1987; Scholl 1988]

12.2. 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 its 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 1990a; 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; Leung and Muntz 1992; Segev and Gunadhi 1989]. While logical query algebras require extensions to accommodate time, only some storage structures and algorithms, e.g., multi-dimensional 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. Schneider et al. 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 [DeWitt, Naughton, and Schneider 1991b]. Such join predicates operations 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. In order to verify this fit, interpolation, extrapolation, and digital filtering were prototyped in the Volcano system, 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 (multi-dimensional sub-array) extraction, are not as easy to integrate into database query processing systems. Some of them seem to fit better into the storage management sub-system 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 multi-dimensional 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).

12.3. 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, Su, and Lam 1991; Mitschang 1989; Shaw and Zdonik 1989a; Shaw and Zdonik 1989b; Shaw and Zdonik 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 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 inter-object 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 in the Gemstone object-oriented database system product [Maier and Stein 1986], Bertino et al. in the Orion project [Bertino and Kim 1989; Bertino 1990; Bertino 1991; Bertino 1994], and by Kemper et al. in the GOM project [Kemper and Moerkotte 1990a; Kemper and Moerkotte 1990b; Kemper, Kilger, and Moerkotte

1991; Kemper and Moerkotte 1992]. At this point, it is to 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 recently proposed keeping access patterns or sequences and activating read-ahead if accesses equal or similar to a stored pattern are detected [Palmer and Zdonik 1991]. 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, Graefe, and Maier 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].

12.4. 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 supporting common subexpressions. Common subexpressions can be executed efficiently with an operator that passes the result of a common subexpression to multiple consumers, as mentioned briefly in the section of 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 control-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, dataflow is usually paced or driven from the top, the consumer. The left-most diagram of Figure 43 shows the control flow of normal iterators. (Notice that the arrows in Figure 43 show the *control* flow; the data flow of all diagrams in Figure 43 is assumed to be upward. In data-driven dataflow, control and data flows point in the same direction; in demand-driven dataflow, 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 43, might be more

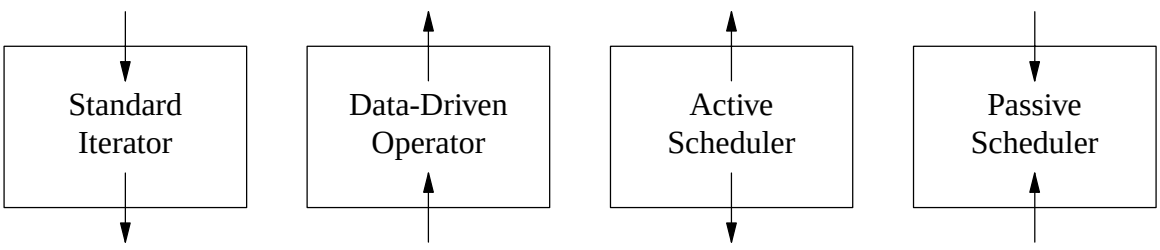


Figure 43. Operators, Schedulers, and Control Flow.

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 43. 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 co-routine fashion to ensure that the resumed neighbor will eventually demand the item the scheduler just received. The final diagram of Figure 43 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, Chu, and Sager 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 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 [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, Barnes, and Morris 1989].

13. 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.

13.1. 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 upon 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 based data. Not surprisingly, 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 non-redundant 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 "0^{th}" prototype [Chamberlin et al. 1981a], view indices as analyzed by Roussopoulos [Roussopoulos 1991], two-relation *join indices* as proposed by Valduriez [Valduriez 1987], and 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[40], explored

predominantly for joins. The simplest form of precomputation in support of binary operations are 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 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 [Babb 1982]. Larson et al. investigated storing and maintaining materialized views in relational database systems [Blakeley, Coburn, and Larson 1989; Blakeley and Martin 1990; Larson and Yang 1985; Medeiros and Tompa 1985; Tompa and Blakeley 1988; Yang and Larson 1987]. 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 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 [Blakeley and Martin 1990]. 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 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 vs. deferred updates [Hanson 1987]. 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 analyzed caching of results in a query language called Quel+ (which is a subset of Postquel [Stonebraker, Rowe, and Hirohama 1990]) 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

[40] This paragraph was written using ideas and notes by José A. Blakeley.

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. 1990; Stonebraker, Rowe, and Hirohama 1990]. 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.

13.2. Data Compression

[41]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, Witten, and Cleary 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 datasets 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

[41] This section has been derived from [Graefe and Shapiro 1991].

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 transferred faster to and from disk. 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/O's. 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 uncompressed. Third, both the join attributes nor other attributes can to be compressed 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 and 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 the number of records 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.

For the purpose of a small performance comparison, consider the I/O costs for hybrid hash join using 400 pages of memory of two relations R and S that require uncompressed 1,000 pages for R and 5,000 pages for S, or half as much compressed. For simplicity, we ignore

fragmentation and assume uniform hash value distributions.

First, consider the cost using uncompressed data. The cost of reading the stored data is 6,000 I/Os. When building the hash table on R, 398 pages can remain in memory and 602 pages must be written to two build overflow files. During probing with input S, records equivalent to 1,990 pages can be joined immediately with the resident hash table, and 3,010 pages must be written to two overflow files. Next, the two pairs of overflow files must be joined, requiring 602+3,010 I/Os to read them. The entire cost is 6,000 I/Os on permanent files and 7,224 I/Os on temporary files, for a total of 13,224 I/Os.

Now consider joining compressed input relations. Reading the initial compressed data requires 500+2,500 I/Os. In the build phase, 399 pages remain in memory, 101 pages are written to an overflow file. In the probe phase, records equivalent to 1,995 pages are joined immediately, and 505 pages are written to disk. Joining the two overflow files requires 101+505 I/Os. The entire cost is 3,000 I/Os to permanent files and 1,212 I/Os to temporary files, for a total of 4,212 I/Os.

The total I/O costs differ by a factor of more than three. While the I/O costs for the permanent files differ by a factor of two, as expected for a compression ratio of 50%, the I/O costs for temporary files differ by a factor of almost six. A factor of two could easily be expected; however, the improved utilization of memory (more records remain in the hash table during the build phase) significantly reduces the number of records that must be written to overflow files. Thus, compression reduces both the number and size of records written to temporary files, resulting in a reduction of I/O costs on temporary files by a factor of six.

If the compression scheme had been a little more effective, i.e., a factor of 2½ instead of 2 or a reduction to 40% instead of 50%, overflow files would have been avoided entirely for

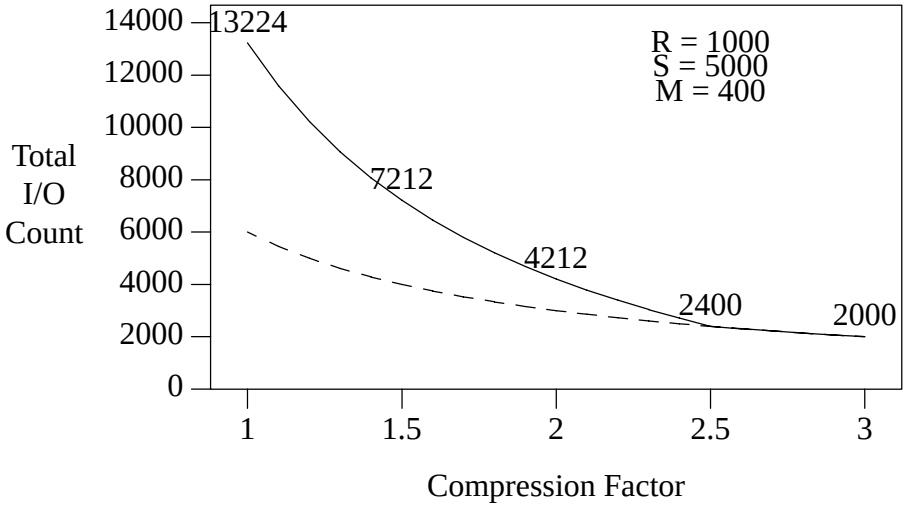


Figure 44. Effect of Compression on Hybrid Hash Join Performance.

compressed data, leaving only the I/O on permanent data. The total I/O costs would have differed by a factor of 5½, 2,400 to 13,224 I/Os. Figure 44 shows the effect of the compression factor on hybrid hash join performance for relations R and S. The numbers above the curve indicate the exact I/O cost for the compression factors marked at the bottom axis.

The graph can be divided into two regions. For compression factors below 2½, the build input is larger than memory, hash table overflow occurs, and I/O reduction by compression is more than the compression factor, similar to the example above. For compression factors above 2½, no overflow occurs and compression only reduces I/O on permanent files. It is very encouraging to observe in this graph that already very moderate compression factors, e.g., 1½, reduce the total I/O cost significantly. Even if some additional cost is incurred for decompressing output data, the performance gain through compressed permanent and temporary data on disk and in memory far outweights the costs of decompression.

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 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.

13.3. 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 means by which surrogate processing improves query processing performance is reduction in the amount of data being copied. 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 RID's or pointers to the original records if they are kept in memory. If a record is 50 times larger than a RID, e.g., 8 B vs. 400 B, 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 CPU's accessing shared memory over a single 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 permits 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 RID's 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 earlier 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 used RID's 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 re-partitioning and merging records very quickly by their rank alone without further data comparisons [Lorie and Young 1989].

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, Kent, and Ramamohanarao 1987]. Berra et al. considered indexing and retrieval organizations for very large (relational) knowledge bases and databases [Berra, Chung, and Hachem 1987; Chung and Berra 1988]. They employed three techniques, concatenated code words (CCW's), superimposed code words (SCW's), and transformed inverted lists (TIL's). TIL's are normal index structures for all attributes of a relation that permit answering conjunctive queries by bitwise *and*ing. CCW's and SCW's 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. consider associative memory and optical computing techniques to search efficiently through such indices, although conventional software techniques could be used as well.

Techniques based on hash values and bit patterns have also been used in other contexts; we discuss bit vector filtering in the next subsection.

13.4. 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 communication 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, require 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 analyzed the use of multiple bit vector filters in detail [Babb 1979].

The Gamma relational database machine demonstrated the effectiveness of bit vector filtering in relational join processing on distributed-memory hardware [DeWitt et al. 1986; DeWitt, Ghandeharizadeh, and Schneider 1988; DeWitt et al. 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 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" [Mullin 1990]. 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 discussed a limited form of this idea using only two bit vector filters and augmenting it with bit vector filter compression [Qadah 1988].

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 non-matching 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 a single input item (e.g., a duplicate removal with very few duplicates in the input) and if the bit vector filter is quite large.

In that case, the duplicate-free partition files are very large, and the smaller partition files 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. 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.

13.5. 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 [Su 1988]. Most of this research was abandoned after Boral and DeWitt's influential analysis [Boral and DeWitt 1983] 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 CPU's is available and can 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 towards 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 management software.

14. Query Optimization

After this detailed discussion of query evaluation techniques, let us briefly consider query optimization again. The relationship between query optimization and query evaluation was briefly touched upon in the introduction — the facilities of the query execution engine define the space of possible plans that can be chosen by the query optimizer. There is also an alternative interpretation: a query optimizer an expert system that finds the best plan given the query semantics, the current database state, and the capabilities of the query evaluation system. The database state includes both logical properties such as schemas and set cardinalities and physical properties such as sort order, indices, clustering, compression, and partitioning in parallel and

distributed systems. Logical properties are those that can be determined for intermediate query results by knowing the logical algebra operators only, whereas derivation of physical properties requires knowledge of algorithm choices. The prototypical example for a logical property is the schema. The set of attributes and their values present in an intermediate result depends only on the logical operators; it had better not depend on algorithm choices. The typical physical property is sort order in relational systems. On the logical level, relations are sets and sort order makes no sense, whereas sort order is an important consideration on the algorithm level.

The distinction of logical and physical aspects in database query processing has also been used in query optimization. On the logical level, there is an algebra of operations that typically can be implemented by a variety of algorithms, e.g., relational join. The physical algebra is the set of specific and concrete algorithms, and has been the focus of this paper.

Table 12 illustrates many of the concerns that determine query processing performance and allocates these concerns to phases from the database design to the actual execution[42]. Notice that logical and physical database design are included in the table. Given a logical and physical database design as well as the data in the database, i.e., the database state, semantic optimization derives implied predicates (e.g., $r.A = s.A$ and $s.A = t.A$ implies $r.A = t.A$), logical optimization considers alternative formulations of the query, and physical optimization maps a given query formulation to the optimal combination of execution algorithms. Along with algebraic manipulation of the query, first in the logical and then in the physical algebra, a query optimizer must keep track of properties, such as the schemas of intermediate results (a logical property) and their sort orders (a physical property).

The logical algebra for relational database systems, the relational algebra, is well-understood, whereas research into physical algebras continues to advance. Cost models in relational query optimization typically focus on throughput-oriented measures such as CPU- and I/O-time; more complex models could be based on 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), the time-space-product of locked database items and their lock duration, 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 [Ganguly, Hasan, and Krishnamurthy 1992; Lohman et al. 1985; Selinger et al. 1979].

Selectivity estimation has been explored using a wide variety of accuracies and techniques. The most basic methods are based on the assumptions of uniform distributions between minimum and maximum value for each attribute and independent distributions among multiple attributes [Christodoulakis 1984]. There have been a number of refinements for this method: key and foreign keys and the implied join output size, the number of distinct values for each attribute,

---

[42] While these phases are quite distinct in most relational systems, these distinctions might become cumbersome and be removed in future, more dynamic query processing systems.

Stage	Typical Concerns
Logical database design	Entities and their relationships Integrity constraints Object behavior Mapping to desired data model, e.g., relational
Physical database design	File formats Index selection Pointer usage Partitioning Striping Replication
Semantic query optimization	Attribute equivalence classes Simplification using integrity constraints
Logical query optimization	Calculus-to-algebra translation Algebraic transformations Intermediate type and schema derivations Statistical profile, selectivity, size estimation
Physical query optimization	Algorithm selection Interesting orderings, algorithm cooperation Index usage Resource allocation (CPUs, disks, memory) Cost model and cost calculations
Query evaluation	Resource negotiation and adjustment Dynamic algorithm choices Scheduling in complex query plans Skew management

Table 12. Query Processing: Phases and Concerns.

histograms [Kooi 1980; Lynch 1988; Muralikrishna and DeWitt 1988], parametric statistics, non-parametric density functions [Graefe 1987b; Sun et al. 1993], the most frequent values and their frequency [Ioannidis and Christodoulakis 1991; Wang 1992], and multi-attribute variations of these. Mannino et al. give a survey of many selectivity estimation methods [Mannino, Chu, and Sager 1988].

Among the open issues in selectivity estimation and cost modeling are statistical summary data and appropriate estimation techniques for large and object-oriented queries in particular summary data about data clustering and the composition about heterogeneous sets; scheduling and cost estimation of complex query plans [Chen et al. 1992; Schneider 1990; Schneider and DeWitt 1990] (see the exponential propagation of estimation errors illustrated in Table 13); appropriate considerations of time and total effort in parallel systems [Ganguly, Hasan, and

Operators	Estimation Error per Operator 2%	5%	10%	20%
5	1.10	1.28	1.61	2.49
10	1.22	1.63	2.59	6.19
20	1.49	2.65	6.73	38.3
50	2.69	11.5	117	9,100
100	7.24	132	13,780	82,817,977

Table 13. Estimation Error Factors for Very Complex Queries.

Krishnamurthy 1992]; and the division between optimization and execution time, i.e., delaying optimization decisions until start-up or run-time [Cole and Graefe 1993; Graefe and Ward 1989; Ioannidis et al. 1992; Seppi, Barnes, and Morris 1989].

For extensible and object-oriented database systems, both suitable logical and physical algebras are still the subject of intense debates. In the area of optimization, this debate has led to numerous proposals for extensible and modular query optimizers, e.g. [Becker and Gueting 1992; Freytag 1987; Graefe 1987a; Mitchell, Zdonik, and Dayal 1992; Rosenthal and Chakravarthy 1988; Sciore and Sieg 1990; Sieg 1989]. These proposals permit adding new logical and physical algebra operators and encapsulate logical and physical properties in abstract data types [Graefe and DeWitt 1987; Graefe and McKenna 1993; Graefe et al. 1994; Lee, Freytag, and Lohman 1988; Lohman 1988]. Their search strategies are modular or parameterized. Since dynamic programming was applied very successfully to database query optimization in the System R project [Selinger et al. 1979], others have adapted it for extensible optimizers [Graefe and McKenna 1993; Graefe et al. 1994; Lee, Freytag, and Lohman 1988].

In extensible optimizers, the capabilities of each physical algorithm are described by implementation rules and a set of functions. The implementation rules capture the correspondence between operators of the logical and the physical algebras. The most obvious of the functions required for each query processing algorithm is the cost function, which translates the operator arguments (e.g., a predicate) and the logical and physical properties of the algorithm's inputs (e.g., size and sort order) into a cost value. The profile functions compute the values for the abstract data types representing logical and physical properties from the corresponding values for the operator's or algorithm's inputs. For example, the output of a sort operator is sorted and either compressed or decompressed depending on the sort's input, while the output of a decompression operator is decompressed and unsorted unless an order-preserving compression scheme was used. Depending on the search mechanisms used in the optimizer, a function may be useful that determines whether or not an algorithm can produce its result with certain physical properties, together with a determination of the physical properties required of the algorithm's inputs. For example, a hybrid hash join can produce decompressed results provided its inputs are decompressed, and a merge-join can produce decompressed results provided its inputs are sorted appropriately and decompressed.

In order to ensure that intermediate results have the desired physical properties, some extensible optimizers consider a special class of algorithms called "glue" operators (they make subplans compatible, or glue them together) or "enforcers" (they enforce desired physical properties) [Blakeley, McKenna, and Graefe 1993; Graefe and McKenna 1993; Graefe et al. 1994; Lee, Freytag, and Lohman 1988] These operators do not contribute to data manipulation on the logical level but enable the use of data manipulation algorithms that require special input characteristics, which is a generalization of the concept of sort orders and interesting orderings used in the System R optimizer [Selinger et al. 1979]. Typical glue operators are sort, data exchange, and decompression. This set is likely to expand with more research into efficient object-oriented database systems and their query processing techniques. The set of useful physical properties and the set of enforcers depends on the set of query evaluation algorithms and the input characteristics that affect their performance. If the set of query evaluation algorithms is extensible, the set of possibly interesting characteristics might grow, and the set of physical properties must also be kept extensible. Ongoing research into query optimization and evaluation techniques in parallel, distributed, extensible, and object-oriented database systems will most certainly introduce new requirements for the next generation of extensible query optimization systems, in particular in the almost entirely untouched subject of parallel object-oriented database systems and query processing.

15. Tuning Query Performance

In the previous section, we considered a number of stages that determine query processing performance in database systems. In this section, we will try to translate the issues raised to give some guidelines on how to improve query performance if it does not match one’s expectations.

The stages discerned in the previous section pertain to database design, query compilation (and optimization), and query evaluation. For the present discussion, we may want to include an earlier stage, namely the implementation by the implementor or vendor of a database management system. Moreover, we are going to rearrange the issues a little bit according to who is most concerned. The three groups of issues in this section will be (i) concerns for the implementor and vendor, (ii) concerns for the database administrator, (iii) concerns for the application programmer. We address each of these groups in turn.

Of course, any performance and tuning study has to start by determining the bottleneck, and then focus on those performance enhancements that alleviate that bottleneck. Let us briefly consider a few typical issues before we more systematically assign concerns into the groups above. If a database system is CPU-bound, the first step must be to analyze which functions require the most CPU effort. The "usual suspects" are buffer management, locking and latching, copying, and expression evaluation (predicates interpretation). Buffer management effort can be reduced not only by improving the buffer code but also by reducing the number of buffer manager calls. Locking and latching can be improved by faster code, function in-lining, and by larger locking granules. Expression evaluation can be sped up by further precompilation, e.g., into compact, easy-to-interpret byte codes or, at the expense of portability, machine code. If a database system is I/O-bound, on the other hand, indices, striping, more disk drives, larger units of I/O, and a larger buffer are the most effective remedies. Other remedies include improved

buffer replacement more efficient read-ahead and write-behind, and data compression. If a system is neither CPU-bound nor I/O-bound, the bottleneck can be the system bus, communication, concurrency control, or some load control mechanism such as a transaction processing monitor [Bernstein 1990]. Insufficient degrees of concurrency can typically be increased by smaller locking granules but sometime require improved concurrency control protocols in database search structures. Moreover, in all cases, an unsuitable query evaluation plan might be a more fundamental cause for unsatisfactory performance.

Let us now assign issues to the groups introduced above. Quite obviously, there are a large number of performance issues that must be addressed by the database system implementor, and we can therefore only identify the "usual suspects." In order to permit most efficient I/O, the database system should support indexing, striping, and clustering. Striping increase the "raw" I/O bandwidth usable for a scan, whereas indexing uses key comparisons to support some types of retrieval predicates. The more index types are available, the larger the set of retrieval predicates that can be supported. Moreover, multiple different index types permit matching the index structure accurately to the anticipated retrieval requests.

The database implementor must also ensure that the query optimizer is reliable and accurate. Query optimizer improvements must be made with great care with respect to the future user: users typically are very confused by a new product release that behaves erratically or not as well as earlier releases in some cases. Even elementary semantic query optimization can have a significant performance impact, both positive and negative (if it is missing). Since they are the Achilles heels of query optimization, selectivity estimation, cost calculations, and ranking of plans should be as accurate as possible. Value distributions and semantic information should taken into account as well as the effect of I/O buffers and their replacement choices.

In order to improve query evaluation performance, the database system implementor can use a wide array of techniques. Most of them have been mentioned earlier; we list here the ones that could truly be improved in many systems. An efficient implementation of asynchronous, striped I/O using large units of I/O is important to deliver as much as possible of the hardware's raw I/O power to the database system. Index retrievals should not immediately retrieve entire records, but should be combined with the operations on RID list discussed in Section 4. The suite of available one-to-one match algorithms should include nested loops join (for non-equijoin predicates), index nested loops join (for small datasets and fast response time), merge-join (to exploit sorted data sets and indices, including those sorted on hash values and compressed values), and hybrid hash join (to operate efficiently on very large, unsorted inputs). Similarly, grouping, aggregation, and duplicate removal should be implemented using both sort- and hash-based algorithms. Pipelining between operators within a process must incur minimal overhead; if possible, copying should be avoided. Parallelism must be implemented in a very flexible way, because parallel hardware designs and customer configurations vary widely (and will continue to do so, maybe increasingly so). Since query evaluation algorithms are predictable, the query evaluation engine should give replacement hints to the buffer manager. Expression evaluation should be tuned finely, because the expression evaluator may be invoked many millions of times in a single query. Copying should be avoided, in particular in shared memory systems; for example, all query evaluation algorithms should be implemented such that they can operate as

efficiently on small surrogate records as obtained from indices (key-RID pairs, for example). Resource allocation needs to be at least somewhat dynamic in order to accommodate highly loaded systems as well as exploit lightly load systems. Moreover, by implementing utilities such as database loading and unloading, index creation, statistics gathering (for query optimization), and data replication should be implemented as operators in the physical algebra to ensure that they benefit from all improvements of the query evaluation engine (e.g., asynchronous I/O and parallelism). Finally, appropriate locking granules must be provided for in the concurrency control mechanisms and used in database utilities as well as the query evaluation plans.

Presuming that the database system implementor provides efficient and powerful facilities, the database administrator has to make sure that they are used. For example, the optimizer requires detailed, up-to-date statistics on the database in order to find good plans. The physical database design must be carefully chosen, particularly with respect to partitioning of data sets over many devices, indexing, free space allocation, and caching of derived data. Moreover, processing resources must be available and efficient, in particular working space and buffer memory and disk bandwidth and space for temporary files. The last database administrator concern listed here are the granules of concurrency control and recovery, which must be a defined in a system- and application-specific tradeoff between concurrency and overhead.

The last link in improving query processing performance are the application programs that include database requests. The most important issue is that they contain rich, high-level queries and do not implement functionality that could be delegated to (and optimized by!) the database system. Our discussion on index nested loops join vs. memory-based join technique (Section 6) illustrates this point. Another issue specific to the SQL query language and its optimization is to avoid nested subqueries, if possible. In some systems, optimization goals and the expected result size can be specified, e.g., first-item response time or total resource consumption; such specifications can help the query optimizer to find better plans. And finally, just like the database system implementor and the database administrator, the application programmer has to think about transactions, in particular about transaction boundaries and application check-points in order to avoid excessive lock retention times and restart effort in the case of a failure.

16. Directions

After this long tour through the world of database query processing, one might ask into which directions this field will go next. There is a large number of very different directions that are being considered, but we will divide them into a few categories. First, there are efforts underway for functionality and performance. With the development of new query languages and facilities, in particular SQL-2 and SQL-3 in the relational world, query processing algorithms must be extended and adapted. For example, all SQL-compliant systems need to support three forms of outer joins and set operations. At the same time, many real decisions are made based on performance, although it could be argued that more disks and the next generation of processing hardware will improve performance more drastically than yet another little algorithm trick. The main direction in the performance arena is effective use of parallelism, both in shared-memory for medium-size databases and in distributed-memory for very large databases [DeWitt and Gray 1992]. Distributed memory ("shared nothing") machines also receive a lot of attention

due to their potential (with suitable software) to be fault-tolerant and highly available.

Second, research and development efforts can be divided by their emphasis on query optimization or query execution. For a number of important objectives, Table 14 lists some approaches that are currently being investigated. Of course, multiple objectives might coincide, e.g., more and more complex queries over larger and larger databases in market analysis applications, making progress in query processing technology even more urgent. Many databases today are larger than 100 GB, and relational tables with more than one million rows must be processed on a regular basis.

Finally, there are some directions that we could consider paradigm shifts in query processing rather than incremental refinements. In the recent past, the move to parallel query evaluation has been such a shift, bringing with it a spirited discussion and interesting paper titles, e.g., [Agrawal and DeWitt 1984; Bic and Hartman 1985; Boral and DeWitt 1983; DeWitt and Gray 1990; DeWitt and Gray 1992]. Current ones include encapsulating behavior with database

Objective	Query Optimization Techniques	Query Execution Techniques
Efficient query processing over very large databases	Improved selectivity estimation methods Refined cost models (e.g., clustering, buffering) Planning for parallel execution Planning bushy trees and allocating resources	Disk striping and I/O parallelism Parallel processing Algorithm refinements Dynamic adjustment of resource allocations
Efficient processing of very complex queries	Improved selectivity estimation methods Planning bushy trees Heuristic guidance and pruning	Dynamic resource management
Complex object structures	Statistical summary data and cost estimation for clustering Rewrite rules for path expressions and join operations	Parallel (multi-disk, multi-node) complex object assembly
Encapsulated behavior and heterogeneous collections	Optimizable query sub-languages Partial type binding at compile-time Assistance by type implementors, e.g., for selectivity estimation	Mutual recursion between query execution engine and language interpreter Operator implementations extensible wrt. instance types

Table 14. Directions.

data, which brings a lot of advantages from a software engineering point of view (data abstraction, integrity enforcement) but also brings almost insurmountable obstacles in query processing. Most importantly, query optimization critically depends on the ability to reason about the semantics of a query, to generate equivalent forms, and to anticipate execution costs. If behavior is expressed in a computationally complete language, there is no hope of reasoning about equivalent behaviors. The best approach, it seems, is to specify a powerful but not complete sublanguage that can be optimized, and to invoke the query optimizer for those queries and methods that are given entirely within that sublanguage. Of course, the discussion about such sublanguages is far from over, although the discussion now seems to have settled on algebraic languages over "bulk" types.

Another paradigm shift concerns the division of physical database design, optimization (compile-time) and and execution (run-time). Since the success of compile-time optimization in System R [Chamberlin et al. 1981b; Selinger et al. 1979] and the poor fairing of Ingres Decomposition [Wong and Youssefi 1976; Youssefi and Wong 1979], it has been common wisdom that these three phases should be separated[43]. Since then, only very few database system have employed run-time optimization or re-optimization except for infeasible plans, e.g., IBM's AS/400 [Fawcette 1989]. However, these two alternatives define the end points of a spectrum,

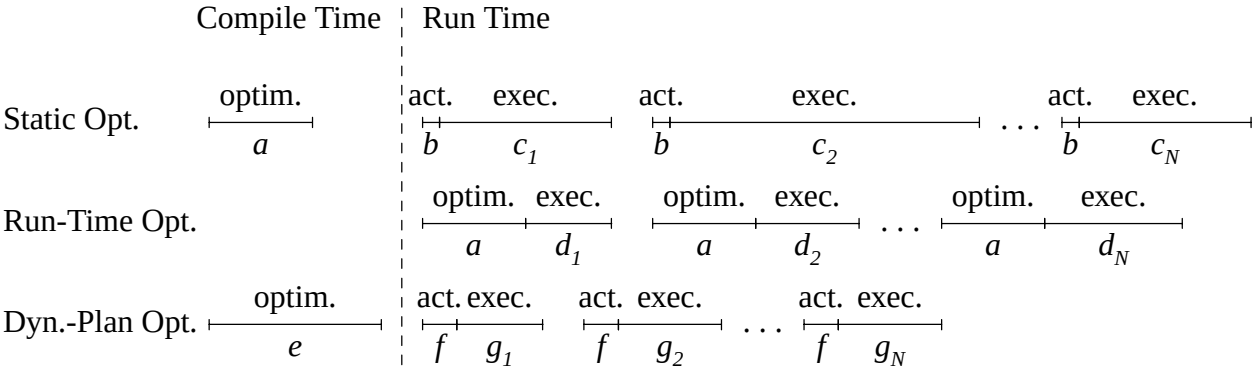


Figure 45. Optimization Options for Queries with Run-Time Bindings.

[43] In retrospect, Ingres Decomposition had several separate elements that each contributed to its poor performance. First, it delayed all optimization until run-time. Second, multi-variable queries (joins) induced repetitive optimization of similar queries. Third, a greedy optimization strategy was used, i.e., only one choice was explored at any choice-point. Fourth, the only join method employed was nested loops join, either naive or preferably index nested loops join. Fifth, temporary files for intermediate results increased I/O costs. To the best of our knowledge, there has never been a study to separate the impact of each of the elements on Ingres Decomposition query processing performance.

but not the entire spectrum. A possible intermediate point is compile-time optimization into "dynamic plans," i.e., plans that are completely optimized at compile-time but have some, selected optimization decisions delayed into run-time, namely those that depend on variable bindings that won't be available until run-time [Cole and Graefe 1993; Graefe and Ward 1989; Ioannidis et al. 1992]. If there are many delayed decisions to be made at run-time, the plan activiation time is increased by decision procedures beyond the traditional validation that the plan is still feasible. This is but one example for reconsidering the division of query processing activities into compile-time and run-time; other options could and probably should be explored. For example, instead of producing multiple complete plans at compile-time, an optimizer might produce an expression over a mixed algebra of logical and physical operations, leaving the logical operations to be optimized at start-up time by reinvoking the optimizer on these (presumably small) subexpressions.

17. Summary and Outlook

Database management systems provide three essential groups of services. First, they maintain both data and associated meta-data 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." 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, parallelization, and physical database design may be crucial contributions to achieving this goal, in addition to the query execution techniques surveyed here.

References

Adam and Wortmann 1989: N. R. Adam and J. C. Wortmann, Security-Control Methods for Statistical Databases: A Comparative Study, *ACM Computing Surveys 21*, 4 (December 1989), 515.

Agrawal and DeWitt 1984: R. Agrawal and D. J. DeWitt, Whither Hundreds of Processors in a Database Machine, *Proc. Int'l. Workshop on High-Level Architecture*, Los Angeles, CA, 1984.

Ahn and Snodgrass 1988: I. Ahn and R. Snodgrass, Partitioned storage for temporal databases, *Inf. Sys. 13*, 4 (1988), 369.

Albert 1991: J. Albert, Algebraic Properties of Bag Data Types, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 211.

Analyti and Pramanik 1992: A. Analyti and S. Pramanik, Fast Search in Main Memory Databases, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 215.

Anderson, Tzou, and Graham 1988: D. P. Anderson, S. Y. Tzou, and G. S. Graham, The DASH Virtual Memory System, November 1988.

Antoshenkov 1993: G. Antoshenkov, Dynamic Query Optimization in Rdb/VMS, *Proc. IEEE Int'l. Conf. on Data Eng.*, Vienna, Austria, April 1993, 538.

Astrahan et al. 1976: M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, System R: A Relational Approach to Database Management, *ACM Trans. on Database Sys. 1*, 2 (June 1976), 97. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Astrahan, Schkolnick, and Whang 1987: M. M. Astrahan, M. Schkolnick, and K. Y. Whang, Approximating the number of unique values of an attribute without sorting, *Inf. Sys. 12*, 1 (1987), 11.

Atkinson and Buneman 1987: M. P. Atkinson and O. P. Buneman, Types and Persistence in Database Programming Languages, *ACM Computing Surveys 19*, 2 (June 1987), 105.

Babb 1979: E. Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Trans. on Database Sys. 4*, 1 (March 1979), 1.

Babb 1982: E. Babb, Joined Normal Form: A Storage Encoding for Relational Databases, *ACM Trans. on Database Sys. 7*, 4 (December 1982), 588.

Baeza-Yates and Larson 1989: R. A. Baeza-Yates and P. A. Larson, Performance of B+-Trees with Partial Expansions, *IEEE Trans. on Knowledge and Data Eng. 1*, 2 (June 1989), 248.

Bancilhon and Ramakrishnan 1986: F. Bancilhon and R. Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1986, 16. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Barghouti and Kaiser 1991: N. S. Barghouti and G. E. Kaiser, Concurrency Control in Advanced Database Applications, *ACM Computing Surveys 23*, 3 (September 1991), 269.

Baru and Frieder 1989: C. K. Baru and O. Frieder, Database Operations in a Cube-Connected Multicomputer System, *IEEE Trans. on Computers 38*, 6 (June 1989), 920.

Batini, Lenzerini, and Navathe 1986: C. Batini, M. Lenzerini, and S. B. Navathe, A Comparative Analysis of Methodologies for Database Schema Integration, *ACM Computing Surveys 18*, 4 (December 1986), 323.

Batory et al. 1988: D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise, GENESIS: An Extensible Database Management System, *IEEE Trans. on Softw. Eng. 14*, 11 (November 1988), 1711.

Batory, Leung, and Wise 1988: D. S. Batory, T. Y. Leung, and T. E. Wise, Implementation Concepts for an Extensible Data Model and Data Language, *ACM Trans. on Database Sys. 13*, 3 (September 1988), 231.

Baugsto and Greipsland 1989: B. A. W. Baugsto and J. F. Greipsland, Parallel Sorting Methods for Large Data Volumes on a Hypercube Database Computer, *Proc. Sixth Int'l Workshop on Database Machines*, Deauville, France, June 1989, 127.

Bayer and McCreighton 1972: R. Bayer and E. McCreighton, Organisation and Maintenance of Large Ordered Indices, *Acta Informatica 1*, 3 (1972), 173.

Beck, Bitton, and Wilkinson 1988: M. Beck, D. Bitton, and W. K. Wilkinson, Sorting Large Files on a Backend Multiprocessor, *IEEE Trans. on Computers 37*, 7 (July 1988), 769.

Becker, Six, and Widmayer 1991: B. Becker, H. W. Six, and P. Widmayer, Spatial Priority Search: An Access Technique for Scaleless Maps, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 128.

Becker and Gueting 1992: L. Becker and R. H. Gueting, Rule-Based Optimization and Query Processing in an Ex-

tensible Geometric Database System, *ACM Trans. on Database Sys. 17*, 2 (June 1992), 247.

Beckley, Evans, and Raman 1985: D. A. Beckley, M. W. Evans, and V. K. Raman, Multikey Retrieval from KD-Trees and Quad-Trees, *Proc. ACM SIGMOD Conf.*, Austin, TX, May 1985, 291.

Beckmann et al. 1990: N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, The R*-tree: An Efficient and Robust Access Method for Points and Rectangles, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 322.

Bell, Witten, and Cleary 1989: T. Bell, I. H. Witten, and J. G. Cleary, Modelling for Text Compression, *ACM Computing Surveys 21*, 4 (December 1989), 557.

Bentley 1975: J. L. Bentley, Multidimensional Binary Search Trees Used for Associative Searching, *Comm. of the ACM 18*, 9 (September 1975), 509.

Bernstein and Goodman 1981: P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys 13*, 2 (June 1981), 185.

Bernstein et al. 1981: P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Query Processing in a System for Distributed Databases (SDD-1), *ACM Trans. on Database Sys. 6*, 4 (December 1981), 602.

Bernstein, Hadzilacos, and Goodman 1987: P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

Bernstein 1990: P. Bernstein, Transaction Processing Monitors, *Comm. of the ACM 33*, 11 (November 1990), 75.

Berra, Chung, and Hachem 1987: P. B. Berra, S. M. Chung, and N. I. Hachem, Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base, *IEEE Computer 20*, 3 (March 1987), 25.

Bertino and Kim 1989: E. Bertino and W. Kim, Indexing Techniques for Queries on Nested Objects, *IEEE Trans. on Knowledge and Data Eng. 1*, 2 (June 1989), 196.

Bertino 1990: E. Bertino, Optimization of Queries Using Nested Indices, *Lecture Notes in Comp. Sci. 416*(March 1990), 44, Springer Verlag.

Bertino 1991: E. Bertino, An Indexing Technique for Object-Oriented Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 160.

Bertino 1994: E. Bertino, A Survey of Indexing Techniques for Object-Oriented Databases, in *Query Processing for Advanced Database Applications*, J. C. Freytag, G. Vossen and D. Maier (ed.), Morgan-Kaufman, San Mateo, CA, 1994, 383.

Bhide 1988: A. Bhide, An Analysis of Three Transaction Processing Architectures, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 339.

Bhide and Stonebraker 1988: A. Bhide and M. Stonebraker, A Performance Comparison of Two Architectures for Fast Transaction Processing, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1988, 536.

Bic and Hartman 1985: L. Bic and R. L. Hartman, Hither Hundreds of Processors in a Database Machine, *Proc. Fourth Int'l Workshop on Database Machines*, Grand Bahama Island, March 1985, 153.

Bitton and DeWitt 1983: D. Bitton and D. J. DeWitt, Duplicate Record Elimination in Large Data Files, *ACM Trans. on Database Sys. 8*, 2 (June 1983), 255.

Bitton et al. 1984: D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon, A Taxonomy of Parallel Sorting, *ACM Computing Surveys 16*, 3 (September 1984), 287.

Bitton, Hanrahan, and Turbyfill 1987: D. Bitton, M. B. Hanrahan, and C. Turbyfill, Performance of Complex Queries in Main Memory Database Systems, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1987, 72.

Bitton and Gray 1988: D. Bitton and J. Gray, Disk Shadowing, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 331.

Bitton-Friedland 1982: D. Bitton-Friedland, Design, Analysis, and Implementation of Parallel External Sorting Algorithms, *Ph.D. Thesis, Univ. of Wisconsin − Madison*, January 1982. Also Comp. Sci. Tech. Rep. 464.

Blakeley, Larson, and Tompa 1986: J. A. Blakeley, P. A. Larson, and F. W. Tompa, Efficiently Updating Materialized Views, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1986, 61.

Blakeley, Coburn, and Larson 1989: J. A. Blakeley, N. Coburn, and P. A. Larson, Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates, *ACM Trans. on Database Sys. 14*, 3 (September 1989), 369.

Blakeley and Martin 1990: J. A. Blakeley and N. L. Martin, Join Index, Materialized View, and Hybrid Hash-Join: A Performance Analysis, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 256.

Blakeley, Thompson, and Alashqur 1990: J. A. Blakeley, C. W. Thompson, and A. M. Alashqur, Strawman reference model of object query languages, *Proc. First OODB Standardization Workshop*

X3/SPARC/DBSSG/OODBTG, Atlantic City, NJ, 1990.

Blakeley, McKenna, and Graefe 1993: J. A. Blakeley, W. J. McKenna, and G. Graefe, Experiences Building the Open OODB Query Optimizer, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 287.

Blasgen and Eswaran 1976: M. Blasgen and K. Eswaran, On the Evaluation of Queries in a Relational Database System, *IBM Res. Rep. RJ 1745*, San Jose, CA, April 8, 1976.

Blasgen and Eswaran 1977: M. Blasgen and K. Eswaran, Storage and Access in Relational Databases, *IBM Sys. J. 16*, 4 (1977), 363.

Bloom 1970: B. H. Bloom, Space/Time Tradeoffs in Hash Coding with Allowable Errors, *Comm. of the ACM 13*, 7 (July 1970), 422.

Bober and Carey 1992: P. M. Bober and M. J. Carey, On Mixing Queries and Transactions via Multiversion Locking, *Proc. IEEE Int'l. Conf. on Data Eng.*, Tempe, AR, February 1992, 535.

Boral et al. 1982: H. Boral, D. DeWitt, D. Friedland, N. Jarrell, and W. Wilkinson, Implementation of the Database Machine DIRECT, *IEEE Trans. on Softw. Eng. 8*, 6 (November 1982), 533.

Boral and DeWitt 1983: H. Boral and D. J. DeWitt, Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, *Proc. Int'l. Workshop on Database Machines*, Munich, 1983, 166. Reprinted in A. R. Hurson, L. L. Miller, and S. H. Pakzad, Parallel Architectures for Database Systems, IEEE Computer Society Press, Washington, D.C., 1989.

Boral 1988: H. Boral, Parallelism in Bubba, *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, December 1988, 68.

Boral et al. 1990: H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, Prototyping Bubba, A Highly Parallel Database System, *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 4.

Bratbergsengen 1984: K. Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Proc. Int'l. Conf. on Very Large Data Bases*, Singapore, August 1984, 323.

Brown et al. 1992: K. P. Brown, M. J. Carey, D. J. DeWitt, M. Mehta, and J. F. Naughton, Scheduling Issues for Complex Database Workloads, *Univ. of Wisconsin − Madison Comp. Sci. Tech. Rep. 1095*, July 1992.

Bucheral, Theverin, and Valduriez 1990: P. Bucheral, J. M. Theverin, and P. Valduriez, Efficient Main Memory Data Management Using the DBGraph Storage Model, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 683.

Buneman and Frankel 1979: P. Buneman and R. E. Frankel, FQL - A Functional Query Language, *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 52.

Buneman, Frankel, and Nikhil 1982: P. Buneman, R. E. Frankel, and R. Nikhil, An Implementation Technique for Database Query Languages, *ACM Trans. on Database Sys. 7*, 2 (June 1982), 164.

Cacace, Ceri, and Houtsma 1993: F. Cacace, S. Ceri, and M. A. W. Houtsma, A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs, *to appear in Distr. and Parallel Databases*, 1993.

Carey et al. 1986: M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, Object and File Management in the EXODUS Extensible Database System, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 91.

Carey et al. 1990: M. J. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson, An Incremental Join Attachment for Starburst, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 662.

Carey, Haas, and Livny 1993: M. Carey, L. Haas, and M. Livny, Tapes Hold Data, Too: Challenges of Tuples on Tertiary Store, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 413.

Carlis 1986: J. V. Carlis, HAS: A Relational Algebra Operator, or Divide Is Not Enough to Conquer, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1986, 254.

Carter and Wegman 1979: J. L. Carter and M. N. Wegman, Universal Classes of Hash Functions, *J. of Computers and System Science 18*, 2 (1979), 143.

Chamberlin et al. 1981a: D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolo, P. G. Selinger, M. Schkolnik, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, A History and Evaluation of System R, *Comm. of the ACM 24*, 10 (October 1981), 632. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Chamberlin et al. 1981b: D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnik, P. G. Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost, Support for Repetitive Transactions and Ad Hoc Queries in System R, *ACM Trans. on Database Sys. 6*, 1 (March 1981), 70.

Chang and Katz 1989: E. Chang and R. Katz, Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS, *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 348.

Chen 1976: P. P. Chen, The Entity Relationship Model - Toward an Unified View of Data, *ACM Trans. on Database Sys. 1*, 1 (March 1976), 9. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Chen and Kuck 1984: H. Chen and S. M. Kuck, Combining Relational and Network Retrieval Methods, *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 131.

Chen and Patterson 1990: P. M. Chen and D. A. Patterson, Maximizing Performance in a Striped Disk Array, *Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News 18*, 2 (June 1990), 322.

Chen et al. 1992: M. S. Chen, M. L. Lo, P. S. Yu, and H. C. Young, Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins, *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, August 1992, 15.

Cheng et al. 1991: J. Cheng, D. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, and Y. Wang, An Efficient Hybrid Join Algorithm: A DB2 Prototype, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 171.

Cheng and Hurson 1991: J. R. Cheng and A. R. Hurson, Effective Clustering of Complex Objects in Object-Oriented Databases, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 22.

Cheriton, Goosen, and Boyle 1991: D. R. Cheriton, H. A. Goosen, and P. D. Boyle, Paradigm: A Highly Scalable Shared-Memory Multicomputer, *IEEE Computer 24*, 2 (February 1991), 33.

Chiu and Ho 1980: D. M. Chiu and Y. C. Ho, A Methodology for Interpreting Tree Queries Into Optimal Semi-Join Expressions, *Proc. ACM SIGMOD Conf.*, Santa Monica, CA, May 1980, 169.

Chou 1985: H. T. Chou, Buffer Management of Database Systems, *Ph.D. Thesis, Univ. of Wisconsin − Madison*, May 1985.

Chou and DeWitt 1985: H. T. Chou and D. J. DeWitt, An Evaluation of Buffer Management Strategies for Relational Database Systems, *Proc. Int'l. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985, 127. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Christodoulakis 1984: S. Christodoulakis, Implications of Certain Assumptions in Database Performance Evaluation, *ACM Trans. on Database Sys. 9*, 2 (June 1984), 163.

Chung and Berra 1988: S. M. Chung and P. B. Berra, A Comparison of Concatenated and Superimposed Code Word Surrogate Files for Very Large Data/Knowledge Bases, *Lecture Notes in Comp. Sci. 303*(April 1988), 364, Springer Verlag.

Cluet et al. 1989: S. Cluet, C. Delobel, C. Lecluse, and P. Richard, Reloops, an Algebra Based Query Language for an Object-Oriented Database System, *Proc. First Int'l. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 4-6, 1989.

Cole and Graefe 1993: R. L. Cole and G. Graefe, An Optimizer for Dynamic Query Execution Plans, *in preparation*, 1993.

Comer 1979: D. Comer, The Ubiquitous B-Tree, *ACM Computing Surveys 11*, 2 (June 1979), 121.

Copeland et al. 1988: G. Copeland, W. Alexander, E. Boughter, and T. Keller, Data Placement in Bubba, *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 99.

Copeland et al. 1989: G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith, The Case for Safe RAM, *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, 327.

Cormack 1985: G. V. Cormack, Data Compression In a Database System, *Comm. of the ACM 28*, 12 (December 1985), 1336.

Cornell and Yu 1987: D. W. Cornell and P. S. Yu, A Vertical Partitioning Algorithm for Relational Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1987, 30.

Cornell and Yu 1990: D. Cornell and P. Yu, An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases, *IEEE Trans. on Softw. Eng. 16*, 2 (February 1990), 248.

Dadam et al. 1986: P. Dadam, K. Kuespert, F. Anderson, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch, A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1986, 356.

Daniels and Ng 1982: D. Daniels and P. Ng, Distributed Query Compilation and Processing in R*, *IEEE Database*

Eng. 5, 3 (September 1982), .

Daniels et al. 1991: S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt, and B. Vance, Query Optimization in Revelation, an Overview, *IEEE Database Eng. 14*, 2 (June 1991), 58.

Davidson, Garcia-Molina, and Skeen 1985: S. B. Davidson, H. Garcia-Molina, and D. Skeen, Consistency in Partitioned Networks, *ACM Computing Surveys 17*, 3 (September 1985), 341.

Davis 1992: D. D. Davis, Oracle's Parallel Punch for OLTP, *Datamation*, August 1, 1992, 67.

Davison 1992: W. Davison, Parallel Index Building in Informix OnLine 6.0, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 103.

DeWitt and Hawthorn 1981: D. J. DeWitt and P. B. Hawthorn, A Performance Evaluation of Database Machine Architectures, *Proc. Int'l. Conf. on Very Large Data Bases*, Cannes, France, September 1981, 199.

DeWitt et al. 1984: D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, Implementation Techniques for Main Memory Database Systems, *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 1.

DeWitt and Gerber 1985: D. J. DeWitt and R. H. Gerber, Multiprocessor Hash-Based Join Algorithms, *Proc. Int'l. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985, 151.

DeWitt et al. 1986: D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, GAMMA - A High Performance Dataflow Database Machine, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 228. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

DeWitt, Ghandeharizadeh, and Schneider 1988: D. J. DeWitt, S. Ghandeharizadeh, and D. Schneider, A Performance Analysis of the GAMMA Database Machine, *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 350.

DeWitt et al. 1990: D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, The Gamma Database Machine Project, *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 44.

DeWitt and Gray 1990: D. DeWitt and J. Gray, Parallel Database Systems: The Future of Database Processing or a Passing Fad, *ACM SIGMOD Record, Special Issue on Directions for Future Database Research and Development 19*, 4 (December 1990), 104.

DeWitt 1991: D. J. DeWitt, The Wisconsin Benchmark: Past, Present, and Future, in *Database and Transaction Processing Sys. Performance Handbook*, J. Gray (ed.), Morgan-Kaufman, San Mateo, CA, 1991.

DeWitt, Naughton, and Schneider 1991a: D. DeWitt, J. Naughton, and D. Schneider, Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting, *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, Miami Beach, FL, December 1991.

DeWitt, Naughton, and Schneider 1991b: D. J. DeWitt, J. E. Naughton, and D. A. Schneider, An Evaluation of Non-Equijoin Algorithms, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 443.

DeWitt and Gray 1992: D. J. DeWitt and J. Gray, Parallel Database Systems: The Future of High-Performance Database Systems, *Comm. of the ACM 35*, 6 (June 1992), 85.

DeWitt, Naughton, and Burger 1993: D. DeWitt, J. Naughton, and J. Burger, Nested Loops Revisited, *Proc. Parallel and Distr. Inf. Sys.*, San Diego, CA, January 1993.

Deppisch, Paul, and Schek 1986: U. Deppisch, H. B. Paul, and H. J. Schek, A Storage System for Complex Objects, *Proc. Int'l Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, September 1986, 183.

Deshpande and van Gucht 1988: A. Deshpande and D. van Gucht, An Implementation for Nested Relational Databases, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 76.

Deshpande and Larson 1991: V. Deshpande and P. A. Larson, *An Algebra for Nested Relations With Support for Nulls and Aggregates*, Comp. Sci. Dept., Univ. of Waterloo, Waterloo, Ontario, Canada, April 1991.

Deshpande and Larson 1992: V. Deshpande and P. A. Larson, The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors, *Proc. IEEE Int'l. Conf. on Data Eng.*, Tempe, AR, February 1992, 68.

Dozier 1992: J. Dozier, Keynote Address: Access to Data in NASA's Earth Observing Systems, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 1.

Drew, King, and Hudson 1990: P. Drew, R. King, and S. Hudson, The Performance and Utility of the Cactis Implementation Algorithms, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 135.

Effelsberg and Haerder 1984: W. Effelsberg and T. Haerder, Principles of Database Buffer Management, *ACM Trans. on Database Sys. 9*, 4 (December 1984), 560.

Elhard and Bayer 1984: K. Elhard and R. Bayer, A Database Cache for High Performance and Fast Restart in Database Systems, *ACM Trans. on Database Sys. 9*, 4 (December 1984), 503.

Enbody and Du 1988: R. J. Enbody and H. C. Du, Dynamic Hashing Schemes, *ACM Computing Surveys 20*, 2 (June 1988), 85.

Epstein, Stonebraker, and Wong 1978: R. Epstein, M. Stonebraker, and E. Wong, Distributed Query Processing in a Relational Database System, *Proc. ACM SIGMOD Conf.*, Austin, TX, May 1978.

Epstein 1979: R. Epstein, Techniques for Processing of Aggregates in Relational Database Systems, *Univ. of California at Berkeley, UCB/ERL Memorandum M79/8*, February 1979.

Epstein and Stonebraker 1980: R. Epstein and M. Stonebraker, Analysis of Distributed Data Base Processing Strategies, *Proc. Int'l. Conf. on Very Large Data Bases*, Montreal, Canada, October 1980, 92.

Fagin et al. 1979: R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, Extendible Hashing: A Fast Access Method for Dynamic Files, *ACM Trans. on Database Sys. 4*, 3 (September 1979), 315.

Faloutsos 1985: C. Faloutsos, Access Methods for Text, *ACM Computing Surveys 17*, 1 (March 1985), 49.

Faloutsos, Ng, and Sellis 1991: C. Faloutsos, R. Ng, and T. Sellis, Predictive Load Control for Flexible Buffer Allocation, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 265.

Fang, Lee, and Chang 1986: M. T. Fang, R. C. T. Lee, and C. C. Chang, The Idea of Declustering and Its Applications, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 181.

Fawcette 1989: J. E. Fawcette, Understanding IBM's Relational Database Technology, *Inside IBM's Database Strategy, an IBM Sponsored Supplement to DBMS*, Redwood City, CA, September 1989, 9.

Finkel and Bentley 1974: R. A. Finkel and J. L. Bentley, Quad Trees: A Data Structure for Retrieval on Composite Keys, *Acta Informatica 4*, 1 (1974), 1.

Finkelstein, Schkolnick, and Tiberio 1988: S. Finkelstein, M. Schkolnick, and P. Tiberio, Physical Database Design for Relational Databases, *ACM Trans. on Database Sys. 13*, 1 (March 1988), 91.

Fontenot 1989: M. Fontenot, Software Congestion, Mobile Servers, and the Hyperbolic Model, *IEEE Trans. on Softw. Eng. 15*, 8 (August 1989), 947.

Freytag 1987: J. C. Freytag, A Rule-Based View of Query Optimization, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 173.

Freytag and Goodman 1989: J. C. Freytag and N. Goodman, On the Translation of Relational Queries into Iterative Programs, *ACM Trans. on Database Sys. 14*, 1 (March 1989), 1.

Fushimi, Kitsuregawa, and Tanaka 1986: S. Fushimi, M. Kitsuregawa, and H. Tanaka, An Overview of the System Software of a Parallel Relational Database Machine GRACE, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 209.

Gallaire, Minker, and Nicolas 1984: H. Gallaire, J. Minker, and J. M. Nicolas, Logic and Databases: A Deductive Approach, *ACM Computing Surveys 16*, 2 (June 1984), 153.

Ganguly, Hasan, and Krishnamurthy 1992: S. Ganguly, W. Hasan, and R. Krishnamurthy, Query Optimization for Parallel Execution, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 9.

Garcia-Molina and Salem 1988: H. Garcia-Molina and K. Salem, The Impact of Disk Striping on Reliability, *Princeton Univ. Comp. Sci. Tech. Rep.*, January 1988.

Gerber 1986: R. H. Gerber, Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms, *Ph.D. Thesis, Univ. of Wisconsin − Madison*, October 1986.

Gerber and DeWitt 1987: R. H. Gerber and D. J. DeWitt, The Impact of Hardware and Software Alternatives on the Performance of the Gamma Database Machine, *Univ. of Wisconsin − Madison Comp. Sci. Tech. Rep. 708*, July 1987.

Ghandeharizadeh and DeWitt 1990a: S. Ghandeharizadeh and D. J. DeWitt, Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 481.

Ghandeharizadeh and DeWitt 1990b: S. Ghandeharizadeh and D. J. DeWitt, A Multiuser Performance Analysis of Alternative Clustering Strategies, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 466.

Ghandeharizadeh et al. 1991: S. Ghandeharizadeh, L. Ramos, Z. Asad, and W. Qureshi, Object Placement in Parallel Hypermedia Systems, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 243.

Gibson et al. 1989: G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson, Failure Correction

Techniques for Large Disk Arrays, *Third Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1989, 123.

Goodman and Woest 1988: J. R. Goodman and P. J. Woest, The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor, *Univ. of Wisconsin − Madison Comp. Sci. Tech. Rep. 766*, April 1988.

Gouda and Dayal 1981: M. G. Gouda and U. Dayal, Optimal Semijoin Schedules for Query Processing in Local Distributed Database Systems, *Proc. ACM SIGMOD Conf.*, Ann Arbor, MI, April-May 1981, 164.

Graefe 1987a: G. Graefe, Software Modularization with the EXODUS Optimizer Generator, *IEEE Database Eng. 10*, 4 (December 1987), .

Graefe 1987b: G. Graefe, Selectivity Estimation Using Moments and Density Functions, *OGC Comp. Sci. Tech. Rep. 87-012*, Beaverton, OR, November 1987.

Graefe and DeWitt 1987: G. Graefe and D. J. DeWitt, The EXODUS Optimizer Generator, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 160.

Graefe and Maier 1988: G. Graefe and D. Maier, Query Optimization in Object-Oriented Database Systems: A Prospectus, in *Advances in Object-Oriented Database Sys.*, vol. Lecture Notes in Comp. Sci. 334, K.R. Dittrich (ed.), Springer-Verlag, September 1988, 358.

Graefe 1989: G. Graefe, Relational Division: Four Algorithms and Their Performance, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1989, 94.

Graefe and Ward 1989: G. Graefe and K. Ward, Dynamic Query Evaluation Plans, *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 358.

Graefe 1990a: G. Graefe, Parallel External Sorting in Volcano, *Univ. of Colorado at Boulder Comp. Sci. Tech. Rep. 459*, 1990.

Graefe 1990b: G. Graefe, Encapsulation of Parallelism in the Volcano Query Processing System, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 102. Reprinted in M. Stonebraker, Readings in Database Sys., 2nd ed., Morgan-Kaufman, San Mateo, CA, 1993.

Graefe 1991: G. Graefe, Heap-Filter Merge Join: A New Algorithm for Joining Medium-Size Inputs, *IEEE Trans. on Softw. Eng. 17*, 9 (September 1991), 979.

Graefe and Shapiro 1991: G. Graefe and L. D. Shapiro, Data Compression and Database Performance, *Proc. ACM/IEEE-Comp. Sci. Symp. on Applied Computing*, Kansas City, MO, April 1991.

Graefe and Thakkar 1992: G. Graefe and S. S. Thakkar, Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor, *Software − Practice and Experience 22*, 7 (July 1992), 495.

Graefe and Wolniewicz 1992: G. Graefe and R. H. Wolniewicz, Algebraic Optimization and Parallel Execution of Computations over Scientific Databases, *Proc. Workshop on Metadata Management in Scientific Databases*, Salt Lake City, UT, November 3-5, 1992.

Graefe 1993a: G. Graefe, Performance Enhancements for Hybrid Hash Join, *submitted for publication*, 1993.

Graefe 1993b: G. Graefe, A Performance Evaluation of Histogram-Driven Recursive Hybrid Hash Join, *submitted for publication*, August 1993.

Graefe 1993c: G. Graefe, Volcano, An Extensible and Parallel Dataflow Query Processing System, *to appear in IEEE Trans. on Knowledge and Data Eng. 5*, 6 (December 1993), .

Graefe and Cole 1993: G. Graefe and R. L. Cole, Fast Algorithms for Universal Quantification in Large Databases, *submitted for publication*, March 1993.

Graefe and Davison 1993: G. Graefe and D. L. Davison, Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Processing, *IEEE Trans. on Softw. Eng. 19*, 8 (August 1993), 749.

Graefe and McKenna 1993: G. Graefe and W. J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, *Proc. IEEE Int'l. Conf. on Data Eng.*, Vienna, Austria, April 1993, 209.

Graefe 1994: G. Graefe, Sort-Merge-Join: An Idea Whose Time Has(h) Passed?, *to appear in Proc. IEEE Int'l. Conf. on Data Eng.*, Houston, TX, February 1994.

Graefe et al. 1994: G. Graefe, R. L. Cole, D. L. Davison, W. J. McKenna, and R. H. Wolniewicz, Extensible Query Optimization and Parallel Execution in Volcano, in *Query Processing for Advanced Database Applications*, J. C. Freytag, G. Vossen and D. Maier (ed.), Morgan-Kaufman, San Mateo, CA, 1994, 305.

Graefe, Linville, and Shapiro 1994: G. Graefe, A. Linville, and L. D. Shapiro, Sort versus Hash Revisited, *to appear in IEEE Trans. on Knowledge and Data Eng.*, 1994.

Gray et al. 1981: J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolo, and I. Traiger, The Recovery Manager of the System R Database Manager, *ACM Computing Surveys 13*, 2 (June 1981), 223.

Gray and Putzolo 1987: J. Gray and F. Putzolo, The 5 Minute Rule for Trading Memory for Disc Accesses and The 10 Byte Rule for Trading Memory for CPU Time, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 395.

Gray 1990: J. Gray, A Census of Tandem System Availability Between 1985 and 1990, *Tandem Computers Tech. Rep. 90.1*, Cupertino, CA, January 1990.

Gray, Horst, and Walker 1990: J. Gray, B. Horst, and M. Walker, Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 148.

Gray and Reuter 1991: J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufman, San Mateo, CA, 1991.

Gremillion 1982: L. L. Gremillion, Designing a Bloom Filter for Differential File Access, *Comm. of the ACM 25*, 9 (September 1982), 600.

Gruenwald and Eich 1991: L. Gruenwald and M. H. Eich, MMDB Reload Algorithms, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 397.

Guenther and Bilmes 1991: O. Guenther and J. Bilmes, Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation, *IEEE Trans. on Knowledge and Data Eng. 3*, 3 (September 1991), 342.

Guibas and Sedgewick 1978: L. Guibas and R. Sedgewick, A Dichromatic Framework for Balanced Trees, *Proc. 19th Symp. on the Found. of Comp. Sci.*, 1978.

Gunadhi and Segev 1990: H. Gunadhi and A. Segev, A Framework for Query Optimization in Temporal Databases, *Proc. Fifth Int'l. Conf. on Statistical and Scientific Database Management*, April 1990.

Gunadhi and Segev 1991: H. Gunadhi and A. Segev, Query Processing Algorithms for Temporal Intersection Joins, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 336.

Gunther and Wong 1987: O. Gunther and E. Wong, A Dual Space Representation for Geometric Data, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 501.

Gunther 1989: O. Gunther, The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1989, 598.

Guo, Su, and Lam 1991: M. Guo, S. Y. W. Su, and H. Lam, An Association Algebra for Processing Object-Oriented Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 23.

Guttman 1984: A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 47. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Haas et al. 1982: L. M. Haas, P. G. Selinger, E. Bertino, D. Daniels, B. Lindsay, G. Lohman, Y. Masunaga, C. Mohan, P. Ng, P. Wilms, and R. Yost, *R*: A Research Project on Distributed Relational DBMS*, IBM Res. Division, San Jose CA, October 1982.

Haas et al. 1989: L. Haas, J. C. Freytag, G. Lohman, and H. Pirahesh, Extensible Query Processing in Starburst, *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 377.

Haas et al. 1990: L. Haas, W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita, Starburst Mid-Flight: As the Dust Clears, *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 143.

Haerder 1978: T. Haerder, Implementing a Generalized Access Path Structure for a Relational Database System, *ACM Trans. on Database Sys. 3*, 3 (September 1978), 285.

Haerder and Reuter 1983: T. Haerder and A. Reuter, Principles of Transaction-Oriented Database Recovery, *ACM Computing Surveys 15*, 4 (December 1983), . Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Haerder and Petry 1987: T. Haerder and E. Petry, Evaluation of a multiple version scheme for concurrency control, *Inf. Sys. 12*, 1 (1987), 83.

Hafez and Ozsoyoglu 1988: A. Hafez and G. Ozsoyoglu, Storage Structures for Nested Relations, *IEEE Database Eng. 11*, 3 (September 1988), 31.

Hagmann 1986: R. B. Hagmann, An Observation on Database Buffering Performance Metrics, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 289.

Hammer and Chan 1976: M. Hammer and A. Chan, Index Selection in a Self-Adaptive Data Base Management System, *Proc. ACM SIGMOD Conf.*, 1976, 1.

Hammer and Niamir 1979: M. Hammer and B. Niamir, A Heuristic Approach to Attribute Partitioning, *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 93.

Hamming 1977: R. W. Hamming, *Digital Filters*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

Hanani 1977: M. Z. Hanani, An Optimal Evaluation of Boolean Expressions in an Online Query System, *Comm. of the ACM 20*, 5 (May 1977), 344.

Hanson 1987: E. N. Hanson, A Performance Analysis of View Materialization Strategies, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 440.

Harada et al. 1990: L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi, Query Processing Method for Multi-Attribute Clustered Relations, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 59.

Henrich, Six, and Widmayer 1989: A. Henrich, H. W. Six, and P. Widmayer, The LSD Tree: Spatial Access to Multi-Dimensional Point and Non-point Objects, *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, 45.

Hoel and Samet 1992: E. G. Hoel and H. Samet, A Qualitative Comparison Study of Data Structures for Large Linear Segment Databases, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 205.

Hong and Stonebraker 1991: W. Hong and M. Stonebraker, Optimization of Parallel Query Execution Plans in XPRS, *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, Miami Beach, FL, December 1991.

Hong and Stonebraker 1993: W. Hong and M. Stonebraker, Optimization of Parallel Query Execution Plans in XPRS, *Distr. and Parallel Databases 1*, 1 (January 1993), 9.

Hou and Ozsoyoglu 1991: W. C. Hou and G. Ozsoyoglu, Statistical Estimators for Aggregate Relational Algebra Queries, *ACM Trans. on Database Sys. 16*, 4 (December 1991), 600.

Hou, Ozsoyoglu, and Dogdu 1991: W. C. Hou, G. Ozsoyoglu, and E. Dogdu, Error-Constrained COUNT Query Evaluation in Relational Databases, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 278.

Hou and Ozsoyoglu 1993: W. C. Hou and G. Ozsoyoglu, Processing Time-Constrained Aggregation Queries in CASE-DB, *ACM Trans. on Database Sys. 18*, 2 (June 1993), 224.

Hsiao and DeWitt 1990: H. I. Hsiao and D. J. DeWitt, Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 456.

Hsiao and DeWitt 1991: H. Hsiao and D. DeWitt, A Performance Study of Three High Availability Data Replication Strategies, *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, Miami Beach, FL, December 1991.

Hua and Lee 1990: K. A. Hua and C. Lee, An Adaptive Data Placement Scheme for Parallel Database Computer Systems, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 493.

Hua and Lee 1991: K. A. Hua and C. Lee, Handling Data Skew in Multicomputer Database Computers Using Partition Tuning, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 525.

Hudson and King 1989: S. E. Hudson and R. King, Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System, *ACM Trans. on Database Sys. 14*, 3 (September 1989), 291.

Hull and King 1987: R. Hull and R. King, Semantic Database Modeling: Survey, Applications, and Research Issues, *ACM Computing Surveys 19*, 3 (September 1987), 201.

Hutflesz, Six, and Widmayer 1988a: A. Hutflesz, H. W. Six, and P. Widmayer, Twin Grid Files: Space Optimizing Access Schemes, *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 183.

Hutflesz, Six, and Widmayer 1988b: A. Hutflesz, H. W. Six, and P. Widmayer, The Twin Grid File: A Nearly Space Optimal Index Structure, *Lecture Notes in Comp. Sci. 303*(April 1988), 352, Springer Verlag.

Hutflesz, Six, and Widmayer 1990: A. Hutflesz, H. W. Six, and P. Widmayer, The R-File: An Efficient Access Structure for Proximity Queries, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 372.

Ioannidis and Christodoulakis 1991: Y. E. Ioannidis and S. Christodoulakis, On the Propagation of Errors in the Size of Join Results, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 268.

Ioannidis et al. 1992: Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis, Parametric Query Processing, *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, August 1992, 103.

Iyer and Dias 1990: B. R. Iyer and D. M. Dias, System Issues in Parallel Sorting for Database Systems, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 246.

Jagadish 1990: H. V. Jagadish, A Compression Technique to Materialize Transitive Closure, *ACM Trans. on Database Sys. 15*, 4 (December 1990), 558.

Jagadish 1991: H. V. Jagadish, A Retrieval Technique for Similar Shapes, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 208.

Jarke and Koch 1984: M. Jarke and J. Koch, Query Optimization in Database Systems, *ACM Computing Surveys 16*, 2 (June 1984), 111.

Jarke and Vassiliou 1985: M. Jarke and Y. Vassiliou, A Framework for Choosing a Database Query Language, *ACM Computing Surveys 17*, 3 (September 1985), 313.

Jhingran 1991: A. Jhingran, Precomputation in a Complex Object Environment, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 652.

Kao 1986: S. Kao, DECIDES: An Expert System Tool for Physical Database Design, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1986, 671.

Katz and Wong 1983: R. H. Katz and E. Wong, Resolving Conflicts in Global Storage Design Through Replication, *ACM Trans. on Database Sys. 8*, 1 (March 1983), 110.

Katz and Lehman 1984: R. H. Katz and T. J. Lehman, Database Support for Versions and Alternatives of Large Design Files, *IEEE Trans. on Softw. Eng. 10*, 2 (March 1984), 191.

Katz, Chang, and Bhateja 1986: R. H. Katz, E. Chang, and R. Bhateja, Version Modeling Concepts for Computer-Aided Design Databases, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1986, 379.

Katz 1990: R. H. Katz, Towards A Unified Framework for Version Modeling in Engineering Databases, *ACM Computing Surveys 22*, 3 (December 1990), 375.

Keller, Graefe, and Maier 1991: T. Keller, G. Graefe, and D. Maier, Efficient Assembly of Complex Objects, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 148.

Kemper and Wallrath 1987: A. Kemper and M. Wallrath, An Analysis of Geometric Modeling in Database Systems, *ACM Computing Surveys 19*, 1 (March 1987), 47.

Kemper and Moerkotte 1990a: A. Kemper and G. Moerkotte, Access Support in Object Bases, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 364.

Kemper and Moerkotte 1990b: A. Kemper and G. Moerkotte, Advanced Query Processing in Object Bases Using Access Support Relations, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 290.

Kemper, Kilger, and Moerkotte 1991: A. Kemper, C. Kilger, and G. Moerkotte, Function Materialization in Object Bases, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 258.

Kemper and Moerkotte 1992: A. Kemper and G. Moerkotte, Access support relations: an indexing method for object bases, *Inf. Sys. 17*, 2 (1992), 117.

Kernighan and Ritchie 1978: B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

Kim 1980: W. Kim, A New Way to Compute the Product and Join of Relations, *Proc. ACM SIGMOD Conf.*, Santa Monica, CA, May 1980, 179.

Kim 1984: W. Kim, Highly Available Systems for Database Applications, *ACM Computing Surveys 16*, 1 (March 1984), 71.

Kinsley and Hughes 1992: K. C. Kinsley and C. E. Hughes, Analysis of a Virtual Memory Model For Maintaining Database Views, *IEEE Trans. on Softw. Eng. 18*, 5 (May 1992), 402.

Kitsuregawa, Tanaka, and Motooka 1983: M. Kitsuregawa, H. Tanaka, and T. Motooka, Application of Hash to Data Base Machine and Its Architecture, *New Generation Computing 1*, 1 (1983), 63.

Kitsuregawa, Nakayama, and Takagi 1989: M. Kitsuregawa, M. Nakayama, and M. Takagi, The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method, *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, 257.

Kitsuregawa, Yang, and Fushimi 1989: M. Kitsuregawa, W. Yang, and S. Fushimi, Evaluation of 18-Stage Pipeline Hardware Sorter, *Proc. Sixth Int'l Workshop on Database Machines*, Deauville, France, June 1989, 142.

Kitsuregawa and Ogawa 1990: M. Kitsuregawa and Y. Ogawa, Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Skew in the Super Database Computer (SDC), *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 210.

Klug 1982: A. Klug, Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions, *J. of the ACM 29*, 3 (July 1982), 699.

Knapp 1987: E. Knapp, Deadlock Detection in Distributed Databases, *ACM Computing Surveys 19*, 4 (December 1987), 303.

Knuth 1973: D. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

Kolovson and Stonebraker 1991: C. P. Kolovson and M. Stonebraker, Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 138.

Kooi 1980: R. P. Kooi, The Optimization of Queries in Relational Databases, *Ph.D. Thesis, Case Western Reserve Univ.*, September 1980.

Kooi and Frankforth 1982: R. P. Kooi and D. Frankforth, Query Optimization in Ingres, *IEEE Database Eng. 5*, 3 (September 1982), 2.

Kriegel and Seeger 1987: H. P. Kriegel and B. Seeger, Multidimensional Dynamic Hashing Is Very Efficient for Nonuniform Record Distributions, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1987, 10.

Kriegel and Seeger 1988: H. P. Kriegel and B. Seeger, PLOP-Hashing: A Grid File without Directory, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1988, 369.

Krishnamurthy, Boral, and Zaniolo 1986: R. Krishnamurthy, H. Boral, and C. Zaniolo, Optimization of Nonrecursive Queries, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 128.

Kuespert, Saake, and Wegner 1989: K. Kuespert, G. Saake, and L. Wegner, Duplicate Detection and Deletion in the Extended NF2 Data Model, *Proc. 3rd Int'l. Conf. on Found. of Data Org. and Algorithms*, Paris, France, June 1989. Also IBM Sci. Ctr. Heidelberg Tech. Rep. 88.11.012.

Kumar and Burger 1991: V. Kumar and A. Burger, Performance Measurement of Some Main Memory Database Recovery Algoritms, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 436.

Kung and Robinson 1981: H. T. Kung and J. T. Robinson, On Optimistic Methods for Concurrency Control, *ACM Trans. on Database Sys. 6*, 2 (June 1981), 213. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Lakshmi and Yu 1988: M. S. Lakshmi and P. S. Yu, Effect of Skew on Join Performance in Parallel Architectures, *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, December 1988, 107.

Lakshmi and Yu 1990: M. S. Lakshmi and P. S. Yu, Effectiveness of Parallel Joins, *IEEE Trans. on Knowledge and Data Eng. 2*, 4 (December 1990), 410.

Lanka and Mays 1991: S. Lanka and E. Mays, Fully Persistent B+-trees, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 426.

Larson 1981: P. A. Larson, Analysis of Index-Sequential Files with Overflow Chaining, *ACM Trans. on Database Sys. 6*, 4 (December 1981), 671.

Larson and Yang 1985: P. Larson and H. Yang, Computing Queries from Derived Relations, *Proc. Int'l. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985, 259.

Lee, Freytag, and Lohman 1988: M. Lee, J. C. Freytag, and G. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 218.

Lehman and Carey 1986: T. J. Lehman and M. J. Carey, Query Processing in Main Memory Database Systems, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1986, 239.

Lelewer and Hirschberg 1987: D. A. Lelewer and D. S. Hirschberg, Data Compression, *ACM Computing Surveys 19*, 3 (September 1987), 261.

Leung and Muntz 1990: T. Y. C. Leung and R. R. Muntz, Query Processing in Temporal Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 200.

Leung and Muntz 1992: T. Y. C. Leung and R. R. Muntz, Temporal Query Processing and Optimization in Multiprocessor Database Machines, *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, August 1992, 383.

Li, Rotem, and Wong 1987: J. Li, D. Rotem, and H. Wong, A New Compression Method with Fast Searching on Large Data Bases, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 311.

Li and Naughton 1988: K. Li and J. Naughton, Multiprocessor Main Memory Transaction Processing, *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, December 1988, 177.

Li, Srivastava, and Rotem 1992: J. Li, J. Srivastava, and D. Rotem, CMD: A Multidimensional Declustering Method for Parallel Data Systems, *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, August 1992, 3.

Litwin 1980: W. Litwin, Linear Hashing: A New Tool For File and Table Addressing, *Proc. Int'l. Conf. on Very Large Data Bases*, Montreal, Canada, October 1980, 212. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Litwin, Mark, and Roussopoulos 1990: W. Litwin, L. Mark, and N. Roussopoulos, Interoperability of Multiple Autonomous Databases, *ACM Computing Surveys 22*, 3 (September 1990), 267.

Lohman et al. 1985: G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, Query Processing in R*, in *Query Processing in Database Sys.*, W. Kim, D. S. Reiner, and D. S. Batory (ed.), Springer, Berlin, 1985, 31.

Lohman 1988: G. M. Lohman, Grammar-Like Functional Rules for Representing Query Optimization Alternatives, *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 18.

Lohman et al. 1991: G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer, Extensions to Starburst: Objects, Types, Functions, and Rules, *Comm. of the ACM, Special Section on Next-Generation Database Systems 34*, 10 (October 1991), 94.

Lomet and Salzberg 1990a: D. Lomet and B. Salzberg, The Performance of a Multiversion Access Method, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 353.

Lomet and Salzberg 1990b: D. B. Lomet and B. Salzberg, The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance, *ACM Trans. on Database Sys. 15*, 4 (December 1990), 625.

Lomet 1992: D. Lomet, A Review of Recent Work on Multi-Attribute Access Methods, *ACM SIGMOD Record 21*, 3 (September 1992), 56.

Lorie and Nilsson 1979: R. A. Lorie and J. F. Nilsson, An Access Specification Language for a Relational Database Management System, *IBM J. of Res. and Development 23*, 3 (May 1979), 286.

Lorie and Young 1989: R. A. Lorie and H. C. Young, A Low Communication Sort Algorithm for a Parallel Database Machine, *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, 125.

Lynch and Brownrigg 1981: C. A. Lynch and E. B. Brownrigg, Application of Data Compression to a Large Bibliographic Data Base, *Proc. Int'l. Conf. on Very Large Data Bases*, Cannes, France, September 1981, 435.

Lynch 1988: C. Lynch, Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 240.

Lyytinen 1987: K. Lyytinen, Different Perspectives on Information Systems: Problems and Solutions, *ACM Computing Surveys 19*, 1 (March 1987), 5.

Mackert and Lohman 1989: L. F. Mackert and G. M. Lohman, Index Scans Using a Finite LRU Buffer: A Validated I/O Model, *ACM Trans. on Database Sys. 14*, 3 (September 1989), 401.

Maier 1983: D. Maier, *The Theory of Relational Databases*, Comp. Sci. Press, Rockville, MD, 1983.

Maier and Stein 1986: D. Maier and J. Stein, Indexing in an Object-Oriented DBMS, *Proc. Int'l Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, September 1986, 171.

Maier et al. 1992: D. Maier, G. Graefe, L. Shapiro, S. Daniels, T. Keller, and B. Vance, Issues in Distributed Complex Object Assembly, *Proc. Workshop on Distr. Object Management*, Edmonton, BC, Canada, August 1992.

Maier and Vance 1993: D. Maier and B. Vance, A Call to Order, *Proc. ACM SIGACT News-SIGMOD-SIGART Symp. on Principles of Database Sys.*, Washington, DC, May 1993, 1.

Mannino, Chu, and Sager 1988: M. V. Mannino, P. Chu, and T. Sager, Statistical Profile Estimation in Database Systems, *ACM Computing Surveys 20*, 3 (September 1988), .

Markatos and LeBlanc 1992: E. P. Markatos and T. J. LeBlanc, Shared-Memory Multiprocessor Trends and the Implication for Parallel Program Performance, *Univ. of Rochester Tech. Rep. 420*, Rochester, NY, May 1992.

McKenzie and Snodgrass 1991: L. E. McKenzie and R. T. Snodgrass, Evaluation of Relational Algebras Incorporating the Time Dimension in Databases, *ACM Computing Surveys 23*, 4 (December 1991), .

Medeiros and Tompa 1985: C. Medeiros and F. Tompa, Understanding the Implications of View Update Policies, *Proc. Int'l. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985, 316.

Menon 1986: J. Menon, A Study of Sort Algorithms for Multiprocessor Database Machines, *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 197.

Merchant et al. 1992: A. Merchant, K. L. Wu, P. S. Yu, and M. S. Chen, Performance Analysis of Dynamic Finite Versioning for Concurrent Transaction and Query Processing, *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Int'l. Conf. on Measurement and Modeling of Computer Systems 20*, 1 (June 1-5, 1992), 103.

Mishra and Eich 1992: P. Mishra and M. H. Eich, Join Processing in Relational Databases, *ACM Computing Surveys 24*, 1 (March 1992), 63.

Mitchell, Zdonik, and Dayal 1992: G. Mitchell, S. B. Zdonik, and U. Dayal, An Architecture for Query Processing

in Persistent Object Stores, *Proc. Hawaii Conf. on System Sciences, Vol. 2*, January 1992, 787.

Mitschang 1989: B. Mitschang, Extending the Relational Algebra to Capture Complex Objects, *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, 297.

Mohan et al. 1990: C. Mohan, D. Haderle, Y. Wang, and J. Cheng, Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques, *Lecture Notes in Comp. Sci. 416*(March 1990), 29, Springer Verlag.

Motro 1989: A. Motro, An Access Authorization Model for Relational Databases Based on Algebraic Manipulation of View Definitions, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1989, 339.

Mullin 1983: J. K. Mullin, A Second Look at Bloom Filters, *Comm. of the ACM 26*, 8 (August 1983), 570.

Mullin 1990: J. K. Mullin, Optimal Semijoins for Distributed Database Systems, *IEEE Trans. on Softw. Eng. 16*, 5 (May 1990), 558.

Muntz and Lui 1990: R. R. Muntz and J. C. S. Lui, Performance of Disk Arrays Under Failure, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 162.

Muralikrishna and DeWitt 1988: M. Muralikrishna and D. J. DeWitt, Equi-depth Multi-dimensional Histograms, *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 28.

Muthuraj et al. 1993: J. Muthuraj, S. Chakravarthy, R. Varadarajan, and S. Navathe, A Formal Approach to the Vertical Partitioning Problem in Distributed Database Design, *Proc. Parallel and Distr. Inf. Sys.*, San Diego, CA, January 1993.

Nakayama, Kitsuregawa, and Takagi 1988: M. Nakayama, M. Kitsuregawa, and M. Takagi, Hash-Partitioned Join Method Using Dynamic Destaging Strategy, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 468.

Navathe et al. 1984: S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, Vertical Partitioning Algorithms for Database Design, *ACM Trans. on Database Sys. 9*, 4 (December 1984), 680.

Navathe and Ra 1989: S. Navathe and M. Ra, Vertical Partitioning for Database Design − A Graphical Algorithm, *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 440.

Neches 1984: P. M. Neches, Hardware Support for Advanced Data Management Systems, *IEEE Computer 17*, 11 (November 1984), 29.

Neches 1988: P. M. Neches, The Ynet: An Interconnect Structure for a Highly Concurrent Data Base Computer System, *Proc. 2nd Symp. on the Frontiers of Massively Parallel Computation*, Fairfax, October 1988.

Neugebauer 1991: L. Neugebauer, Optimization and Evaluation of Database Queries Including Embedded Interpolation Procedures, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 118.

Ng, Faloutsos, and Sellis 1991: R. Ng, C. Faloutsos, and T. Sellis, Flexible Buffer Allocation Based on Marginal Gains, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 387.

Nievergelt, Hinterberger, and Sevcik 1984: J. Nievergelt, H. Hinterberger, and K. C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure, *ACM Trans. on Database Sys. 9*, 1 (March 1984), 38.

Nyberg et al. 1993: C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, AlphaSort: A RISC Machine Sort, *Digital Equipment Corp. San Francisco Systems Center Tech. Rep. 93.2*, April 1993.

Olken and Rotem 1989: F. Olken and D. Rotem, Rearranging Data to Maximize the Efficiency of Compression, *J. of Computer and System Sciences 38*, 2 (1989), 405.

Omiecinski 1985: E. Omiecinski, Incremental File Reorganization Schemes, *Proc. Int'l. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985, 346.

Omiecinski and Lin 1989: E. Omiecinski and E. Lin, Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers, *IEEE Trans. on Knowledge and Data Eng. 1*, 3 (September 1989), 329.

Omiecinski and Scheuermann 1990: E. Omiecinski and P. Scheuermann, A Parallel Algorithm for Record Clustering, *ACM Trans. on Database Sys. 15*, 4 (December 1990), 599.

Omiecinski 1991: E. Omiecinski, Performance Analysis of A Load Balancing Relational Hash-Join Algorithm for a Shared-Memory Multiprocssor, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 375.

Ono and Lohman 1990: K. Ono and G. M. Lohman, Measuring the Complexity of Join Enumeration in Query Optimization, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 314.

Ousterhout and Douglis 1989: J. Ousterhout and F. Douglis, Beating the I/O Bottleneck: A Case for Log-Structured File Systems, *Operating Sys. Review*, January 1989, 11.

Ousterhout 1990: J. Ousterhout, Why Aren't Operating Systems Getting Faster as Fast as Hardware?, *Proc.*

USENIX Summer Conf., Anaheim, CA, June 1990.

Ozsoyoglu, Ozsoyoglu, and Matos 1987: G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos, Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions, *ACM Trans. on Database Sys. 12*, 4 (December 1987), 566.

Ozsoyoglu and Wang 1992: Z. M. Ozsoyoglu and J. Wang, A Keying Method for a Nested Relational Database Management System, *Proc. IEEE Int'l. Conf. on Data Eng.*, Tempe, AR, February 1992, 438.

Ozsu and Valduriez 1991a: M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Ozsu and Valduriez 1991b: M. T. Ozsu and P. Valduriez, Distributed Database Systems: Where Are We Now?, *IEEE Computer 24*, 8 (August 1991), 68.

Palmer and Zdonik 1991: M. Palmer and S. B. Zdonik, FIDO: A Cache that Learns to Fetch, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 255.

Papadimitriou and Kanellakis 1984: C. H. Papadimitriou and P. C. Kanellakis, On Concurrency Control by Multiple Versions, *ACM Trans. on Database Sys. 9*, 1 (March 1984), 89.

Patterson, Gibson, and Katz 1988: D. A. Patterson, G. Gibson, and R. H. Katz, A Case for Redundant Arrays of Inexpensive Disks (RAID), *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 109.

Peckham and Maryanski 1988: J. Peckham and F. Maryanski, Semantic Data Models, *ACM Computing Surveys 20*, 3 (September 1988), 153.

Pirahesh et al. 1990: H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger, Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches, *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 1990, 4.

Qadah 1988: G. Z. Qadah, Filter-Based Join Algorithms on Uniprocessor and Distributed-Memory Multiprocessor Database Machines, *Lecture Notes in Comp. Sci. 303*(April 1988), 388, Springer Verlag.

Qian and Wiederhold 1991: X. Qian and G. Wiederhold, Incremental Recomputation of Active Relational Expressions, *IEEE Trans. on Knowledge and Data Eng. 3*, 3 (September 1991), 337.

Rew and Davis 1990: R. K. Rew and G. P. Davis, The Unidata NetCDF: Software for Scientific Data Access, *Sixth Int'l. Conf. on Interactive Inf. and Processing Sys. for Meteorology, Oceanography, and Hydrology*, Anaheim, CA, February 1990.

Richardson and Carey 1987: J. E. Richardson and M. J. Carey, Programming Constructs for Database System Implementation in EXODUS, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 208.

Richardson, Lu, and Mikkilineni 1987: J. P. Richardson, H. Lu, and K. Mikkilineni, Design and Evaluation of Parallel Pipelined Join Algorithms, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 399.

Robinson 1981: J. T. Robinson, The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indices, *Proc. ACM SIGMOD Conf.*, Ann Arbor, MI, April-May 1981, 10.

Rosenblum and Ousterhout 1991: M. Rosenblum and J. K. Ousterhout, The Design and Implementation of a Log-Structured File System, *Proc. Thirteenth ACM Symp. on Operating System Principles 25*, 5 (October 1991), 1.

Rosenblum and Ousterhout 1992: M. Rosenblum and J. K. Ousterhout, The Design and Implementation of a Log-Structured File System, *ACM Trans. on Computer Sys. 10*, 1 (February 1992), 26.

Rosenthal and Reiner 1985: A. Rosenthal and D. S. Reiner, Querying Relational Views of Networks, in *Query Processing in Database Sys.*, W. Kim, D. S. Reiner, and D. S. Batory (ed.), Springer, Berlin, 1985, 109.

Rosenthal and Chakravarthy 1988: A. Rosenthal and U. Chakravarthy, Anatomy of a Modular Multiple Query Optimizer, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 230.

Rosenthal, Rich, and Scholl 1991: A. Rosenthal, C. Rich, and M. Scholl, Reducing Duplicate Work in Relational Join(s): A Modular Approach Using Nested Relations, *ETH Tech. Rep.*, Zurich, Switzerland, June 1991.

Rotem and Segev 1987: D. Rotem and A. Segev, Physical Organization of Temporal Data, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1987, 547.

Roth, Korth, and Silberschatz 1988: M. A. Roth, H. F. Korth, and A. Silberschatz, Extended Algebra and Calculus for Nested Relational Databases, *ACM Trans. on Database Sys. 13*, 4 (December 1988), 389.

Rothnie et al. 1980: J. B. Rothnie, P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong, Introduction to a System for Distributed Databases (SDD-1), *ACM Trans. on Database Sys. 5*, 1 (March 1980), 1.

Roussopoulos 1991: N. Roussopoulos, An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis, *ACM Trans. on Database Sys. 16*, 3 (September 1991), 535.

Roussopoulos and Kang 1991: N. Roussopoulos and H. Kang, A Pipeline N-way Join Algorithm Based on the 2-way Semijoin Program, *IEEE Trans. on Knowledge and Data Eng. 3*, 4 (December 1991), 486.

Ruth and Keutzer 1972: S. S. Ruth and P. J. Keutzer, Data compression for business files, *Datamation 18*(September 1972), 62.

Saake et al. 1989: G. Saake, V. Linnemann, P. Pistor, and L. Wegner, Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype, *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, 307. Extended version in IBM Sci. Ctr. Heidelberg Tech. Rep. 89.03.008, March 1989.

Sacco and Schkolnik 1982: G. M. Sacco and M. Schkolnik, A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model, *Proc. Int'l. Conf. on Very Large Data Bases*, Mexico City, Mexico, September 1982, 257.

Sacco and Schkolnik 1986: G. M. Sacco and M. Schkolnik, Buffer Management in Relational Database Systems, *ACM Trans. on Database Sys. 11*, 4 (December 1986), 473.

Sacco 1987: G. Sacco, Index Access with a Finite Buffer, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 301.

Sacks-Davis and Ramamohanarao 1983: R. Sacks-Davis and K. Ramamohanarao, A two-level superimposed coding scheme for partial match retrieval, *Inf. Sys. 8*, 4 (1983), 273.

Sacks-Davis, Kent, and Ramamohanarao 1987: R. Sacks-Davis, A. Kent, and K. Ramamohanarao, Multikey Access Methods Based on Superimposed Coding Techniques, *ACM Trans. on Database Sys. 12*, 4 (December 1987), 655.

Salem and Garcia-Molina 1986: K. Salem and H. Garcia-Molina, Disk Striping, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1986, 336.

Salzberg 1988: B. Salzberg, *File Structures: An Analytic Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

Salzberg 1990: B. Salzberg, Merging Sorted Runs Using Large Main Memory, *Acta Informatica 27*(1990), 195, Springer.

Salzberg et al. 1990: B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan, FastSort: An Distributed Single-Input Single-Output External Sort, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 94.

Samet 1984: H. Samet, The Quadtree and Related Hierarchical Data Structures, *ACM Computing Surveys 16*, 2 (June 1984), 187.

Schek and Scholl 1986: H. J. Schek and M. H. Scholl, The relational model with relation-valued attributes, *Inf. Sys. 11*, 2 (1986), 137.

Schkolnick and Sorensen 1981: M. Schkolnick and P. Sorensen, The Effects of Denormalization on Database Performance, *IBM Res. Report RJ3082*, 1981.

Schneider and DeWitt 1989: D. Schneider and D. DeWitt, A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 110.

Schneider 1990: D. A. Schneider, Complex Query Processing in Multiprocessor Database Machines, *Ph.D. Thesis, Univ. of Wisconsin − Madison*, 1990. Also Comp. Sci. Tech. Rep. 965.

Schneider and DeWitt 1990: D. A. Schneider and D. J. DeWitt, Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 469.

Schneider 1991: D. A. Schneider, Bit Filtering and Multi-Way Join Query Processing, *unpublished manuscript*, Palo Alto, CA, 1991.

Scholl, Paul, and Schek 1987: M. Scholl, H. B. Paul, and H. J. Schek, Supporting Flat Relations by a Nested Relational Kernel, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 137.

Scholl 1988: M. H. Scholl, The Nested Relational Model − Efficient Support for a Relational Database Interface, *Ph.D. Thesis, Technical Univ. Darmstadt*, 1988. In German.

Sciore and Sieg 1990: E. Sciore and J. Sieg, A Modular Query Optimizer Generator, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 146.

Seeger and Larson 1991: B. Seeger and P. A. Larson, Multi-Disk B-trees, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 436.

Segev and Gunadhi 1989: A. Segev and H. Gunadhi, Event-Join Optimization in Temporal Relational Databases,

Proc. Int'l. Conf. on Very Large Data Bases, Amsterdam, The Netherlands, August 1989, 205.

Selinger et al. 1979: P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, Access Path Selection in a Relational Database Management System, *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 23. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Sellis 1987: T. K. Sellis, Efficiently Supporting Procedures in Relational Database Systems, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 278.

Seppi, Barnes, and Morris 1989: K. Seppi, J. Barnes, and C. Morris, A Bayesian Approach to Query Optimization in Large Scale Data Bases, *The Univ. of Texas at Austin ORP 89-19*, Austin, TX, 1989.

Serlin 1991: O. Serlin, The TPC Benchmarks, in *Database and Transaction Processing Sys. Performance Handbook*, J. Gray (ed.), Morgan-Kaufman, San Mateo, CA, 1991.

Seshadri and Naughton 1992: S. Seshadri and J. F. Naughton, Sampling Issues in Parallel Database Systems, *Proc. Int'l. Conf. on Extending Database Technology*, Vienna, Austria, March 1992.

Severance and Lohman 1976: D. Severance and G. Lohman, Differential Files: Their Application to the Maintenance of Large Databases, *ACM Trans. on Database Sys. 1*, 3 (September 1976), .

Severance 1983: D. G. Severance, A practitioner's guide to data base compression, *Inf. Sys. 8*, 1 (1983), 51.

Severance, Pramanik, and Wolberg 1990: C. Severance, S. Pramanik, and P. Wolberg, Distributed Linear Hashing and Parallel Projection in Main Memory Databases, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 674.

Shapiro 1986: L. D. Shapiro, Join Processing in Database Systems with Large Main Memories, *ACM Trans. on Database Sys. 11*, 3 (September 1986), 239.

Shaw and Zdonik 1989a: G. M. Shaw and S. B. Zdonik, An object-oriented query algebra, in *Proc. 2nd Intl. Workshop on Database Programming Languages*, R. Hull, R. Morrison, and D. Stemple (ed.), Morgan Kaufmann, Gleneden Beach, Oregon, June 1989, 103.

Shaw and Zdonik 1989b: G. Shaw and S. Zdonik, A Object-Oriented Query Algebra, *IEEE Database Eng. 12*, 3 (September 1989), 29.

Shaw and Zdonik 1990: G. M. Shaw and S. B. Zdonik, A Query Algebra for Object-Oriented Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 154.

Shekita and Carey 1990: E. J. Shekita and M. J. Carey, A Performance Evaluation of Pointer-Based Joins, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 300.

Sherman and Brice 1976: S. W. Sherman and R. S. Brice, Performance of a Database Manager in a Virtual Memory System, *ACM Trans. on Database Sys. 1*, 4 (December 1976), 317.

Sheth and Larson 1990: A. P. Sheth and J. A. Larson, Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, *ACM Computing Surveys 22*, 3 (September 1990), 183.

Shipman 1981: D. W. Shipman, The Functional Data Model and the Data Language DAPLEX, *ACM Trans. on Database Sys. 6*, 1 (March 1981), 140. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Sieg 1989: J. Sieg, Making Extensible Database Technology Work, *Ph.D. Thesis, Boston Univ.*, Boston, MA, May 1989.

Sikeler 1988: A. Sikeler, VAR-PAGE-LRU: A Buffer Replacement Algorithm Supporting Different Page Sizes, *Lecture Notes in Comp. Sci. 303*(April 1988), 336, Springer Verlag.

Silberschatz, Stonebraker, and Ullman 1991: A. Silberschatz, M. Stonebraker, and J. Ullman, Database Systems: Achievements and Opportunities, *Comm. of the ACM, Special Section on Next-Generation Database Systems 34*, 10 (October 1991), 110.

Six and Widmayer 1988: H. W. Six and P. Widmayer, Spatial Searching in Geometric Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1988, 496.

Smith and Chang 1975: J. M. Smith and P. Y. T. Chang, Optimizing the Performance of a Relational Algebra Database Interface, *Comm. of the ACM 18*, 10 (October 1975), 568.

Snodgrass 1990: R. Snodgrass, Temporal Databases: Status and Research Directions, *ACM SIGMOD Record, Special Issue on Directions for Future Database Research and Development 19*, 4 (December 1990), 83.

Snodgrass and Shannon 1990: R. Snodgrass and K. Shannon, Semantic Clustering, *Fourth Int'l Workshop on Persistent Object Sys.*, Martha's Vineyard, MA, September 1990, 361.

Sockut and Goldberg 1979: G. H. Sockut and R. P. Goldberg, Database Reorganization − Principles and Practice, *ACM Computing Surveys 11*, 4 (December 1979), 371.

Solworth and Orji 1990: J. A. Solworth and C. U. Orji, Write-Only Disk Caches, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 123.

Srinivasan and Carey 1991: V. Srinivasan and M. J. Carey, Performance of B-Tree Concurrency Control Algorithms, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 416.

Srinivasan and Carey 1992: V. Srinivasan and M. J. Carey, Performance of On-Line Index Construction Algorithms, *Proc. Int'l. Conf. on Extending Database Technology*, Vienna, Austria, March 1992.

Stamos and Young 1989: J. W. Stamos and H. C. Young, A Symmetric Fragment and Replicate Algorithm for Distributed Joins, *IBM Almaden Res. Lab. Tech. Rep. RJ-7188*, San Jose, CA, December 1989.

Stonebraker 1975: M. Stonebraker, Implementation of Integrity Constraints and Views by Query Modification, *Proc. ACM SIGMOD Conf.*, San Jose, CA, June 1975.

Stonebraker 1981: M. Stonebraker, Operating System Support for Database Management, *Comm. of the ACM 24*, 7 (July 1981), 412. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Stonebraker 1986a: M. Stonebraker, The Design and Implementation of Distributed INGRES, in *The INGRES Papers*, M. Stonebraker (ed.), Addison-Wesley, Reading, MA, 1986, 187.

Stonebraker 1986b: M. Stonebraker, The Case for Shared-Nothing, *IEEE Database Eng. 9*, 1 (March 1986), .

Stonebraker 1987: M. Stonebraker, The Design of the POSTGRES Storage System, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 289. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Stonebraker, Aoki, and Seltzer 1988: M. Stonebraker, P. Aoki, and M. Seltzer, Parallelism in XPRS, *Univ. of California UCB/ERL Memorandum M89/16*, Berkeley, February 1989.

Stonebraker et al. 1988: M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, The Design of XPRS, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 318.

Stonebraker 1989: M. Stonebraker, The Case for Partial Indexes, *ACM SIGMOD Record 18*, 4 (December 1989), 4.

Stonebraker et al. 1990: M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, On Rules, Procedures, Caching and Views in Data Base Systems, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 281.

Stonebraker, Rowe, and Hirohama 1990: M. Stonebraker, L. A. Rowe, and M. Hirohama, The Implementation of Postgres, *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 125.

Stonebraker and Schloss 1990: M. Stonebraker and G. A. Schloss, Distributed RAID − A New Multiple Copy Algorithm, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 430.

Stonebraker 1991: M. Stonebraker, Managing Persistent Objects in a Multi-Level Store, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 2.

Straube and Ozsu 1989: D. D. Straube and M. T. Ozsu, Query transformation rules for an object algebra, Univ. of Alberta, Dept. of Computing Sciences Tech. Rep. 89-23, August 1989.

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

Sun et al. 1993: W. Sun, Y. Ling, N. Rishe, and Y. Deng, An Instant and Accurate Size Estimation Method for Joins and Selections in a Retrieval-Intensive Environment, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 79.

Tansel and Garnett 1992: A. U. Tansel and L. Garnett, On Roth, Korth, and Silberschatz's Extended Algebra and Calculus for Nested Relational Databases, *ACM Trans. on Database Sys. 17*, 2 (June 1992), 374.

Teoroy, Yang, and Fry 1986: T. J. Teoroy, D. Yang, and J. P. Fry, A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model, *ACM Computing Surveys 18*, 2 (June 1986), 197.

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

Thomas et al. 1990: G. Thomas, G. R. Thompson, C. W. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman, Heterogeneous Distributed Database Systems for Production Use, *ACM Computing Surveys 22*, 3 (September 1990), 237.

Tompa and Blakeley 1988: F. W. Tompa and J. A. Blakeley, Maintaining materialized views without accessing base data, *Inf. Sys. 13*, 4 (1988), 393.

Toyoma 1993: M. Toyoma, Counter Reduction Technique for Combining Processing of Selection and Join, *Inf. Sys. 18*, 1 (January 1993), 23.

Traiger 1982: I. L. Traiger, Virtual Memory Management for Data Base Systems, *ACM Operating Sys. Review 16*,

4 (October 1982), 26.
Traiger et al. 1982: I. L. Traiger, J. Gray, C. A. Galtieri, and B. G. Lindsay, Transactions and Consistency in Distributed Database Systems, *ACM Trans. on Database Sys. 7*, 3 (September 1982), 323.
Tsangaris and Naughton 1991: M. M. Tsangaris and J. F. Naughton, A Stochastic Approach for Clustering in Object Bases, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 12.
Tsangaris and Naughton 1992: M. M. Tsangaris and J. F. Naughton, On the Performance of Object Clustering Techniques, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 144.
Tseng and Reiner 1993: E. Tseng and D. Reiner, Parallel Database Processing on the KSR1 Computer, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 453.
Tsur and Zaniolo 1984: S. Tsur and C. Zaniolo, An Implementaton of GEM - Supporting a Semantic Data Model on Relational Back-end, *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 286.
Tukey 1977: J. W. Tukey, *Exploratory Data Analysis*, Addison-Wesley, Reading, MA, 1977.
Unidata 1991: Unidata, NetCDF User's Guide, An Interface for Data Access, *NCAR Technical Note TS-334+1A*, Boulder, CO, April 1991. Version 1.11.
Unnikrishnan, Shankar, and Venkatesh 1988: A. Unnikrishnan, P. Shankar, and Y. V. Venkatesh, Threaded Linear Hierarchical Quadtrees for Computation of Geometric Properties of Binary Images, *IEEE Trans. on Softw. Eng. 14*, 5 (May 1988), 659.
Valduriez 1987: P. Valduriez, Join Indices, *ACM Trans. on Database Sys. 12*, 2 (June 1987), 218.
Vandenberg and DeWitt 1991: S. L. Vandenberg and D. J. DeWitt, Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 158.
Walton 1989: C. B. Walton, Investigating Skew and Scalability in Parallel Joins, *Univ. of Texas, Austin Comp. Sci. Tech. Rep. Tech. Rep.-89-39*, Austin, TX, December 1989.
Walton, Dale, and Jenevein 1991: C. B. Walton, A. G. Dale, and R. M. Jenevein, A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 537.
Wang 1992: Y. Wang, Experience from a Real Life Query Optimizer, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 286.
Weikum, Zabback, and Scheuermann 1991: G. Weikum, P. Zabback, and P. Scheuermann, Dynamic File Allocation in Disk Arrays, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 406.
Whang, Wiederhold, and Saglowicz 1984: K. Y. Whang, G. Wiederhold, and D. Saglowicz, Separability − an Approach to Physical Database Design, *IEEE Trans. on Computers 33*, 3 (March 1984), 209.
Whang, Wiederhold, and Sagalowicz 1985: K. Y. Whang, G. Wiederhold, and D. Sagalowicz, The Property of Separability and Its Application to Physical Database Design, in *Query Processing in Database Sys.*, W. Kim, D. S. Reiner, and D. S. Batory (ed.), Springer, Berlin, 1985, 297.
Whang and Krishnamurthy 1990: K. Y. Whang and R. Krishnamurthy, Query Optimization in a Memory-Resident Domain Relational Calculus Database System, *ACM Trans. on Database Sys. 15*, 1 (March 1990), 67.
Williams et al. 1982: P. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost, R*: An Overview of the Architecture, in *Improving Database Usability and Responsiveness*, P. Scheuermann (ed.), Academic Press, New York, NY, 1982. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.
Wilschut 1993: A. Wilschut, Parallel Query Execution in a Main-Memory Database System, *Ph.D. Thesis, Univ. Twente, Netherlands, Dept. of Comp. Sci.*, April 1993.
Wilschut and Apers 1993: A. N. Wilschut and P. M. G. Apers, Dataflow Query Execution in a Parallel Main-Memory Environment, *Distr. and Parallel Databases 1*, 1 (January 1993), 103.
Wolf et al. 1988: J. L. Wolf, D. M. Dias, B. R. Iyer, and P. S. Yu, A Hybrid Data Sharing - Data Partitioning Architecture for Transaction Processing, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1988, 520.
Wolf, Dias, and Yu 1990: J. L. Wolf, D. M. Dias, and P. S. Yu, An Effective Algorithm for Parallelizing Sort Merge in the Presence of Data Skew, *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 1990, 103.
Wolf et al. 1991: J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek, An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 200.
Wolniewicz and Graefe 1993: R. H. Wolniewicz and G. Graefe, Algebraic Optimization of Computations over Sci-

entific Databases, *Proc. Int'l. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993.

Wong and Youssefi 1976: E. Wong and K. Youssefi, Decomposition - A Strategy for Query Processing, *ACM Trans. on Database Sys. 1*, 3 (September 1976), 223.

Wong and Katz 1983: E. Wong and R. H. Katz, Distributing a Database for Parallelism, *Proc. ACM SIGMOD Conf.*, San Jose, CA, May 1983, 23.

Yang and Larson 1987: H. Yang and P. A. Larson, Query Transformation for PSJ-queries, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 245.

Young and Swami 1992: H. C. Young and A. N. Swami, A Family of Round-Robin Partitioned Parallel External Sort Algorithms, *IBM Almaden Res. Report RJ9104*, November 1992.

Youssefi and Wong 1979: K. Youssefi and E. Wong, Query Processing in a Relational Database Management System, *Proc. Int'l. Conf. on Very Large Data Bases*, Rio de Janeiro, October 1979, 409.

Yu and Chang 1984: C. T. Yu and C. C. Chang, Distributed Query Processing, *ACM Computing Surveys 16*, 4 (December 1984), 399.

Yu and Osborn 1991: L. Yu and S. L. Osborn, An Evaluation Framework for Algebraic Object-Oriented Query Models, *Proc. IEEE Int'l. Conf. on Data Eng.*, Kobe, Japan, April 1991, 670.

Zaniolo 1979: C. Zaniolo, Design of Relational Views Over Network Schemas, *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 179.

Zaniolo 1983: C. Zaniolo, The Database Language Gem, *Proc. ACM SIGMOD Conf.*, San Jose, CA, May 1983, 207. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

Zeller and Gray 1990: H. Zeller and J. Gray, An Adaptive Hash Join Algorithm for Multiuser Environments, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 186.

Zeller 1991: H. Zeller, Adaptive Hash-Join-Algorithmen (in German), *Ph.D. Thesis, Univ. of Stuttgart*, November 1991.

Table of Contents

Abstract 1
Acknowledgements 2
1. Introduction 3
2. Architecture of Query Execution Engines 7
3. Sorting and Hashing 16
3.1. Sorting 16
3.2. Hashing 25
Excursus 1: Reducing Memory-to-Memory Copying 30
Excursus 2: Hash Table Organization in Volcano 31
Excursus 3: Larger Units of I/O through Dynamic Allocation 32
Excursus 4: Temporary Files and Complex Objects 33
4. Disk Access 34
4.1. File Scans 34
4.2. Associative Access using Indices 35
4.3. Unclustered Index Lookup and Complex Object Assembly 40
4.4. Faster Storage Techniques 42
4.5. Buffer Management 45
4.6. Physical Database Design 46
5. Aggregation and Duplicate Removal 50
5.1. Aggregation Algorithms Based on Nested Loops 51
5.2. Aggregation Algorithms Based on Sorting 52
5.3. Aggregation Algorithms Based on Hashing 54
5.4. A Rough Performance Comparison 55
5.5. Additional Remarks on Aggregation 56
6. Binary Matching Operations 57
6.1. Nested Loops Join Algorithms 59
Excursus: Is Index Nested Loops Join Sufficient? 62
6.2. Merge-Join Algorithms 63
6.3. Hash Join Algorithms 64
6.4. Pointer-Based Joins 66
6.5. A Rough Performance Comparison 67
7. Universal Quantification 68
8. Duality of Sort- and Hash-Based Query Processing Algorithms 75
9. Execution of Complex Query Plans 86
10. Mechanisms for Parallel Query Execution 94
10.1. Parallel vs. Distributed Database Systems 95
10.2. Forms of Parallelism 96
10.3. Implementation Strategies 97
10.4. Load Balancing and Skew 101
10.5. Tuning a Parallel System 103
10.6. Architectures and Architecture-Independence 105
11. Parallel Algorithms 108
11.1. Parallel Selections and Updates 108

11.2. Parallel Sorting 109
11.3. Parallel Aggregation and Duplicate Removal 114
11.4. Parallel Joins and Other Binary Matching Operations 114
11.5. Parallel Universal Quantification 117
12. Non-Standard Query Processing Algorithms 118
12.1. Nested Relations 118
12.2. Temporal and Scientific Database Management 121
12.3. Object-Oriented Database Systems 122
12.4. More Control-Operators 123
13. Additional Techniques for Performance Improvement 126
13.1. Precomputation and Derived Data 126
13.2. Data Compression 128
13.3. Surrogate Processing 131
13.4. Bit Vector Filtering 132
13.5. Specialized Hardware 135
14. Query Optimization 135
15. Tuning Query Performance 139
16. Directions 141
17. Summary and Outlook 144
References 146

List of Figures

Figure 1. Query Processing in a Database System 3
Figure 2. Query Processing Steps 4
Figure 3. Logical and Physical Algebra Expressions 8
Figure 4. Left-Deep, Bushy, and Right-Deep Plans 12
Figure 5. Two Operators in a Volcano Query Plan 14
Figure 6. Naive and Optimized Merging 22
Figure 7. Stronger Effect of Optimized Merging 22
Figure 8. Effect of Cluster Size Optimizations 24
Figure 9. Hybrid Hashing 26
Figure 10. Recursive Partitioning 28
Figure 11. A Node in a Quadtree 36
Figure 12. A Complex Object with Multiple Levels of Subcomponents 41
Figure 13. Count of Employees by Department 51
Figure 14. Performance of Sort- and Hash-Based Aggregation 55
Figure 15. Binary One-to-One Matching 57
Figure 16. Effect of Partitioning for Join Operations 65
Figure 17. Recursive Partitioning in Binary Operations 65
Figure 18. Performance of Alternative Join Methods 68
Figure 19. Query Plan for Division by Sort-Based Aggregation 71
Figure 20. Sorted Inputs into Naive Division 72
Figure 21. Divisor Table and Quotient Table in Hash-Division 73
Figure 22. Duality of Partitioning and Merging 77
Figure 23. Partitioning Skew 80
Figure 24. Merge-Join with Symmetric Bit Vector Filtering 83
Figure 25. The Effect of Interesting Orderings 83
Figure 26. Partitioning in a Multi-Input Hash Join 84
Figure 27. The Stop Point During Sorting 88
Figure 28. Plan for Joining Three Inputs 89
Figure 29. A Decision Tree of Partial Plans 90
Figure 30. Implementation using Choose-Plan Operators 91
Figure 31. Bracket Model of Parallelization 98
Figure 32. Operator Model of Parallelization 98
Figure 33. Processes Created by Exchange Operators 100
Figure 34. Skew Limit, Confidence, and Sample Size per Partition 102
Figure 35. Tuning Effectiveness for a Parallel Algorithm 104
Figure 36. A Hierarchical-Memory Architecture 107
Figure 37. Scenario with Possible Deadlock 110
Figure 38. Deadlock-Free Scenario 111
Figure 39. Deadlock Danger Due to a Binary Operation in the Consumer 111
Figure 40. Merge Depth as a Function of Parallelism 113
Figure 41. Symmetric Fragment-and-Replicate Join 116
Figure 42. Nested Relation and Equivalent Flat Relations 119

Figure 43. Operators, Schedulers, and Control Flow 124
Figure 44. Effect of Compression on Hybrid Hash Join Performance 130
Figure 45. Optimization Options for Queries with Run-Time Bindings 143

List of Tables

Table 1. Examples of Iterator Functions 11
Table 2. A Taxonomy of Database Sorting Algorithms 18
Table 3. Variables, Their Meaning and Units 18
Table 4. Effect of Cluster Size Optimizations 24
Table 5. Classification of Some Index Structures 38
Table 6. Physical Database Design Issues 47
Table 7. Classification of Relational Division Algorithms 71
Table 8a. Duality of Sort- and Hash-Based Algorithms 75
Table 8b. Duality of Sort- and Hash-Based Algorithms 76
Table 9. Item and File Values in Merging 81
Table 10. Item and File Values in Partitioning 82
Table 11. Comparison Summary 87
Table 12. Query Processing: Phases and Concerns 137
Table 13. Estimation Error Factors for Very Complex Queries 138
Table 14. Directions 142