Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution

Goetz Graefe and Diane L. Davison

Abstract—**Emerging database application domains demand not only high functionality but also high performance. To satisfy these two requirements, the Volcano query execution engine combines the efficient use of parallelism on a wide variety of computer architectures with an extensible set of query processing operators that can be nested into arbitrarily complex query evaluation plans. Volcano's novel *exchange* operator permits designing, developing, debugging, and tuning data manipulation operators in single-process environments but executing them in various forms of parallelism. The exchange operator shields the data manipulation operators from all parallelism issues, including process allocation, inter-process communication, and flow control. It also hides the machine's architecture, which can be a single-processor, shared-memory, or distributed-memory design. It can even be hierarchical, i.e., an interconnected group of shared-memory machines. In this paper, we detail design and implementation of the recently generalized exchange operator, justify our decision to support hierarchical architectures, and argue that the exchange operator offers a significant advantage for development and maintenance of database query processing software. We also discuss the integration of bit vector filtering into the exchange operator paradigm with only minor modifications.**

Index Terms—**Architecture-independence, distributed memory, encapsulation of parallelism, extensible database systems, hierarchical memory, iterators, query execution, shared memory.**

I. Introduction

FROM a number of research projects and their benchmarks, it is obvious that database query evaluation can benefit significantly from parallel processing [20]. Therefore, both researchers and commercial DBMS vendors are including parallelism in their system designs and implementations. However, the complexity and effort of realizing a high-functionality, high-performance parallel query processing system are severe obstacles in system implementation and have hampered development of parallel query execution facilities in both research and commercial environments.

Manuscript received November 25, 1991; revised February 17, 1993. This work was supported in part by NSF with awards IRI-8805200, IRI-8912618, and IRI-9116547, DARPA with Contract DAAB07-91-C-Q518, Texas Instruments, Digital Equipment Corp., Intel Supercomputer Systems Division, Sequent Computer Systems, ADP, the Oregon Advanced Computing Institute (OACIS), and a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland. Recommended by M. Jarke.

G. Graefe is with the Department of Computer Science, Portland State University, Portland, OR 90207.

D. L. Davison is with the Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309.

IEEE Log Number 9210759.

Much of today's research in database query processing focuses on algebras for object-oriented and extensible systems. In these systems, the ability to add new functionality in the form of operators and algorithms to an existing system is a central requirement. Making such additions as simple and painless as possible is a major goal of extensible database systems. Thus, to enable research into parallel query processing in object-oriented and extensible database systems, parallel systems should be extensible. The design and implementation of a new operator or algorithm should be affected very little or not at all by the fact that the operator will be used in a parallel environment. Our contribution to efficient extensible systems are the design and prototype implementation of a parallel query processing system called Volcano that is extensible with new query processing operators and algorithms. The key to the combination of parallelism and extensibility is that the mechanisms required for parallel query execution are orthogonal to the semantics and implementation of data manipulation operators.

Commercial DBMS vendors with existing query processing software designed for single-process execution face the extensibility problem in the opposite direction: how can parallelism be added to an existing system, requiring as little change as possible to existing query execution modules? Considering the substantial investment that commercial database management systems and their query processing components represent, a simple answer to this question is of significant interest. Our contribution to this problem is a query execution operator that can be added to existing sequential query engines and that encapsulates process control and the machine architecture. In fact, one of the relational DBMS vendors is using the Volcano design to parallelize its query execution software [14].

While parallelism is desirable, it is not clear which of the two prevalent parallel computer architectures is best suited for database systems. The strongest arguments for shared-memory systems are that they allow fast and cheap communication, synchronization, and load balancing; the arguments for distributed-memory approaches include that they are scalable and can avoid bottlenecks due to bus saturation. Therefore, both hardware architectures are commercially available, and database researchers as well as DBMS vendors are searching for software designs that can use and exploit both of them.

The Volcano project investigates issues and tradeoffs of extensibility and efficiency in database query and request processing. Volcano's execution module consists of a large,

extensible set of operators that can be nested into arbitrarily complex plans and executed in parallel on either computer architecture. Due to separation of data manipulation and control of parallelism, only one operator, called *exchange*, deals with parallelism issues; all other operators (which we call the data manipulation operators, e.g., file scan, sort, and merge-join) were designed, implemented, debugged, and tuned in an easy and familiar single-process environment and then parallelized by combining them with this new operator.

Our earlier work on parallel query execution in Volcano had relied on shared memory [28], [33]. While this earlier version of Volcano and its exchange operator was a proof-of-concept prototype for parallelizing an existing query execution engine using this new operator and for the integration of parallelism and extensibility, it is also imperative to support distributed-memory architectures in order to permit scalable high-performance query execution [20]. In this paper, we describe and justify our design goals and our reimplementation of the exchange operator such that it supports not only shared memory but also distributed and hierarchical memory designs. Moreover, we analyze the overhead in the form of control messages and extend the basic abstraction of the exchange operator to support bit vector filtering, a technique that has been used successfully to reduce network traffic in distributed-memory parallel query processing [1], [18].

The following sections are organized as follows. First, we survey related work in Section II. In Section III, we outline and justify our target architecture. Section IV provides a brief overview of Volcano. Section V introduces the operator model of parallel processing and the exchange operator that embodies this model in the Volcano system. Section VI describes Volcano's implementation of the exchange operator for shared memory. The implementation for distributed-memory and hierarchical architectures is considered in Section VII, including the control overhead required by distributed-memory operation and the augmentation of the operator model for bit vector filtering. Extensibility and the application of the exchange operator in scientific databases are discussed in Section VIII. Section IX contains a brief summary and our conclusions from this effort.

II. Related Work

There are two areas of related work, one concerning the best hardware architecture and one the best software architecture for highly parallel database systems. In this section, we summarize what issues have been explored and what approaches have been taken in earlier research projects and prototype systems.

A. Hardware Architectures for Parallel Database Systems

Many database research projects have investigated hardware architectures for parallelism in database systems. Stonebraker compares shared-nothing (distributed-memory), shared-disk (distributed-memory with multiported disks), and shared-everything (shared-memory) architectures for database use based on a number of issues including scalability, communication overhead, locking overhead, and load balancing [57]. His conclusion is 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 important benefit of scalability. Therefore, he concludes that overall shared-nothing is the preferable architecture for database system implementation.

Bhide and Stonebraker compare architectural alternatives for transaction processing [4], [5] and conclude that a shared-everything (shared-memory) design provides the best performance. To achieve high performance, reliability, and scalability, Bhide suggests considering shared-nothing (distributed-memory) machines with shared-everything parallel nodes. The same idea is mentioned in equally general terms by Pirahesh *et al.* [50] and Boral *et al.* [10], but none of these authors elaborate on the idea's generality or potential. Kitsuregawa and Ogawa have designed a new database machine called SDC [45]. Although the SDC machine uses multiple shared-memory nodes (plus custom hardware like the Omega network and a hardware sorter), the effect of the hardware design on operators other than join is not evaluated in [45].

For query processing, customized parallel hardware was investigated for numerous database machine projects but largely abandoned after Boral and DeWitt's influential analysis [7] that compared CPU and I/O speeds and their trends and concluded that I/O rather than CPU processing is most likely the bottleneck in future high-performance query execution. 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 have come to the same conclusion that parallelism is no substitute for effective storage structures and query execution algorithms [15], [48], [55]. Subsequently, both Boral and DeWitt have embarked on new database machine projects, Bubba and Gamma, that run customized software on standard processors with local disks [10], [18]. For scalability and availability, both projects use distributed-memory hardware with single-CPU nodes, and have investigated scaling questions for very large configurations.

Tandem has been using distributed-memory hardware for a long time, mostly for reliability and fault-tolerance reasons. It did not reconsider its hardware for its NonStop SQL product and its parallel query processing facilities [21]. Since Tandem uses its own hardware, it is not clear how easily the software could be moved to a group of shared-memory nodes.

The XPRS system, on the other hand, is being built on shared memory [39], [40], [58]. Its designers believe that modern bus architectures can handle up to 2000 transactions per second (TPS). Shared-memory architectures permit 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 [36]. However, we believe that attaching 250 disks to a single machine as necessary for 2000 TPS [58] requires significant special hardware, e.g., channels or I/O processors, and it is quite likely that the investment for such hardware can have greater impact on overall system performance if spent

on general-purpose CPU's or disks. Without such special hardware, the performance limit for shared-memory machines is probably much lower than 2000 TPS. Furthermore, there already are applications that require larger storage and access capacities.

Richardson *et al.* [51] present an analytical study of parallel join algorithms on a multiple shared-memory "clusters" of CPU's. They assume a group of clusters connected with a global bus with multiple microprocessors and shared memory in each cluster. Disk drives are attached to the buses within clusters. However, their analysis suggests that the best performance is obtained by using only one cluster, i.e., a shared-memory architecture. We contend 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 μs for a comparison, about 2–5 MIPS), a slow network (a bus with typically 100 Mbit/s), 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), more and faster CPU's (e.g., 1000 $\times$ 30 MIPS), and a faster network e.g., a modern hypercube or mesh with hardware routing. In such machines, multiple clusters are likely to be the better choice. On the other hand, communication between clusters will remain a significant expense. Wong and Katz developed the concept of "local sufficiency" [61] that might provide guidance in declustering and replication to reduce data movement between nodes. Other work on declustering and limiting declustering includes [13], [22], [24], [41], [42].

In a recent paper on the future of parallel database systems, DeWitt and Gray survey experimental and commercial parallel database systems and argue that shared-nothing machines will be the platforms of future parallel database systems [20]. Their argument for shared-nothing is focused on scalability, i.e., limited interference among very many processors, ignoring the significant demand for moderately parallel machines and database servers.

Finally, there are several hardware design and prototyping projects that attempt to overcome the shared-memory scaling problem, e.g., the DASH project [46], the Wisconsin Multicube [25], and the Paradigm project [11]. 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 like 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, as we have done in Volcano.

B. Software Architectures for Parallel Query Processing

There are two models that have been used to parallelize database query execution software. We call them the bracket model and the operator model. The bracket model has been used in distributed-memory systems such as Gamma [18] and Bubba [10], while the operator model has been used in Volcano and in simpler forms in R* [47] and the Tandem NonStop SQL product. We describe and discuss the bracket model in this section and the operator model in Sections V through VII.

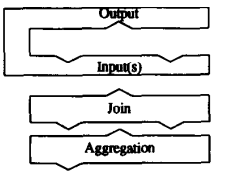

Fig. 1. Bracket model of parallelization.

The bracket model is defined by using a generic template process that can execute a variety of query processing algorithms, but only one at a time. A schematic diagram of such a template process is shown in Fig. 1 with two possible operators, join and aggregation. After an operator has been triggered by the template, it controls execution within its process. In particular, network input and output are performed on request by procedures that are part of the process template. In other words, pacing of network input and output is determined entirely by the executing 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, in particular the operator(s) that produce its input and consume its output.

While the bracket model has been used successfully in distributed-memory relational prototypes, there are two basic problems with the bracket model, which make it unsuitable for extensible database systems. First, each locus of control needs to be created, i.e., each process executing a suitable algorithm. This is typically done by a separate scheduler process. Thus, the semantics and execution strategies of an algorithm are located in two places, namely in the scheduler code and in the algorithm code to be executed within a process template. When a system using a separate scheduler process is extended with a new algorithm or parallel execution strategy, both software modules must be modified.

Second, operators are coded in such a way that network I/O is their only means of obtaining input and delivering output (with the exception of scan and store operators). The reason is that each operator is its own locus of control and network flow control must be used to coordinate multiple operators, e.g., to match two operators' speed in a producer-consumer relationship. Unfortunately, this also means that passing a data item from one operator to another always involves expensive inter-process communication (IPC) system calls, even in cases when an entire query is evaluated on a single CPU and therefore should be executed in a single process (i.e., without any IPC in a single process), or when data do not need to be repartitioned among nodes in a network. An example for the latter is the three-input join query "joinCselAselB" in the Wisconsin Benchmark [6], [8], which uses the same join attribute for both binary joins. Thus, in queries with multiple operators (meaning almost all queries), IPC and its overhead are mandatory in the bracket model rather than optional as in the operator model.

This limitation has been overcome in the operator model of parallel query evaluation, which we will discuss after arguing the case for hierarchical-memory architectures, both from a

hardware and a software perspective, and introducing the Volcano query evaluation system.

III. The Case for Hierarchical Architectures

There are two important and interesting questions about hardware architectures for parallel query processing. Although related, they are not the same. First, what is the best hardware design for database systems? Second, what is the best target architecture for portable, extensible database software?

Contrary to some researchers' belief, we have concluded that shared memory is the best approach to limited degrees of parallelism as required for database applications of moderate size. The reason is that shared memory provides a number of advantages over distributed memory, including not only easier communication and load balancing but also near-linear speedup. In a recent study of shared-memory query processing performance, we have observed a speedup of 14.9 using 16 CPU's and disks [31]. Thus, we believe that shared memory is the best paradigm for limited degrees of parallelism. Another result of this study was that careful tuning might be required to obtain the highest degree of parallelism with near-linear speedup.

On the other hand, we realize that distributed memory is required for exploiting massively parallel processing capabilities. In order to combine the advantages of shared and distributed memory, we recommend that both database hardware and software be based on a hierarchical organization of memory and communication, i.e., a set of shared-memory machines connected by some network. Within each shared-memory node, efficient shared-memory primitives can be used for communication and synchronization, and message passing is used between nodes across the network.

Fig. 2 shows a general hierarchical architecture. The important point is the combination of local buses within shared-memory parallel nodes and a global interconnection network between nodes. 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, and it can itself be organized in some hierarchical way. The local buses may be split into code and data or by address range to obtain less contention, higher bus bandwidth, and hence higher scalability limits for shared memory. Disks can be conventional disk drives or highly parallel disk arrays. Design and placement of caches, disk controllers, terminal connections, and local- and wide-area network connections are also left open. Tertiary storage devices as well as tape drives or other backup devices would probably be connected to local buses.

Independently of any actual hardware platform, we recommend to design and implement new database software as if all parallel machines were based hierarchical architectures. In other words, new software should exploit shared memory as well as support message passing across node boundaries. There are two reasons for this suggestion.

First, genuinely hierarchical computer machines are forthcoming from several hardware vendors and are likely to be

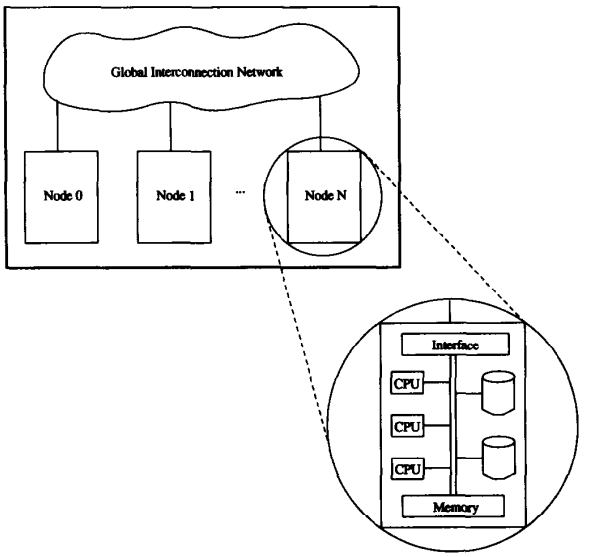

Fig. 2. A hierarchical-memory architecture and one of its nodes.

successful. Parallel database software that can be ported to these new machines and always exploits the most efficient communication primitives will obtain the highest performance. It is very important that the software adapt to changing configurations of hierarchical machines. For example, it should be possible to replace all CPU boards with upgraded models without having to replace memories and disks. Considering that new components will change communication demands, e.g., faster CPU's might require more local bus bandwidth, it is equally important that the allocation of boards to local buses can be changed. More concretely, it should be easy to reconfigure a system, both hardware and software, from 4×16 to 8×8 CPU's.

Second, most of today's parallel machines are built as one of the two extreme cases of this hierarchical design: a conventional distributed-memory machine uses single-CPU nodes, while a shared-memory machine consists of a single node. Software designed for a generic hierarchical architecture will run on either conventional design as well as on a genuinely hierarchical machine. Thus, hierarchical machines are the target architecture of choice for the commercial development of portable parallel software such as database management systems. For database research, such architecture-independent software permits exploring the tradeoffs of alternative parallel architectures with comparisons using a uniform software base.

In a sense, a software system that runs on single-processor, shared-memory, distributed-memory, and hierarchical machines can be considered architecture-independent, although there also are some hardware architectures that are based on heterogeneous sets of nodes. For the time being, we ignore these architectures for two reasons. First, the system software controlling these machines typically uses a set of main nodes (sometimes called compute nodes) and makes all other nodes perform low-level services on demand by the main nodes, e.g., I/O. We presume that database query processing software

would run entirely or almost entirely on the (uniform set of) main nodes. Second, we are not convinced that these architectures will be as successful as the designs with a large set of uniform nodes that permit using all resources for all tasks and therefore better load-balancing for all applications.

In the next section, we start presenting the Volcano query execution system that is based on the operator model of parallelization and offers the advantage of architecture-independent parallel query evaluation. A single "parallelism operator" is the only operator that needs to "understand" the underlying architecture, whereas data manipulation operators can be implemented without concern for machine architecture, process and processor boundaries, data distribution, flow control, or any other parallelism issue.

IV. Volcano System Overview and Single-Process Query Execution

In this section, we introduce the Volcano execution engine by summarizing descriptions of the system from earlier papers. The following three sections discuss Volcano's exchange operator, first in a conceptual overview, then for shared memory, and finally for hierarchical architectures.

The Volcano query execution engine consists of a file system and an extensible set of query execution operators. Volcano's file system is rather conventional. It includes modules for device, buffer, file, and B-tree management. A detailed description of the Volcano architecture can be found in an earlier paper [33].

For our discussion, the most interesting aspect of the file system internals is that the unit of I/O and buffering is not a page but a cluster of multiple pages; the cluster size is set for each file individually. Since the implementation of data streams crossing machine boundaries uses file system code for creating and scanning network packets, the ability to vary cluster sizes implies the ability to choose the sizes of network packets.

The file system routines are used by the query execution routines to evaluate complex query plans. Queries are expressed as complex algebra expressions; the operators of this algebra are query execution algorithms. The set of query execution routines is extensible but already includes file and B-tree scans and maintenance, sorting, and sort- and hash-based implementations of join, semi-join, outer join, union, intersection, difference, aggregation, duplicate removal, and relational division, plus some operations we are exploring in connection with research into algebraic optimization of computations over scientific databases [60].

All operators in Volcano are implemented in the iterator paradigm, also called lazy evaluation, demand-driven dataflow, or synchronous pipelines. This means that each operator is realized through three functions, called the *open*, *next*, and *close* procedures in Volcano because their semantics are similar to the *open*, *next*, and *close* procedures in conventional file scans. Because all operators provide the same iterator protocol and rely on this protocol for their inputs, operators can be nested into arbitrarily complex query plans. Each operator in a multioperator plan is scheduled by its consumer and schedules its producer or producers by means of calls to the three iterator procedures, making query evaluation in a single process self-scheduling with minimal overhead and therefore very efficient. An operator does not need to know what kind of operator produces its input, and whether its input comes from a complex query tree or from a simple file scan. In a sense, each operator in a complex plan acts like an object since it encapsulates its query processing algorithm and state and requests data from the "operator objects" representing its input by sending them *next* "messages," without any specific knowledge of the algorithm of the input operators. 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 extensible systems, e.g., Exodus [52], Genesis [2], [3], and Starburst [37], [38].

In order to ensure extensibility at the instance level, set processing and instance (record) interpretation are very cleanly and consistently separated. All operations on records, e.g., comparison, moving, and hashing, are performed by support functions that are passed to the operators using function entry points, e.g., functions for predicate evaluation and for displaying database items of the query output [33]. To support both compiled and interpreted query execution, there is an argument associated with each support function. For interpreted query processing, a generic predicate interpreter is passed as support function and the code to be interpreted is passed as an argument. For compiled query processing, the argument can either be ignored or used for constants, e.g., for a value with which to compare database values.

The uniform iterator interface allows combining operators into arbitrarily complex query evaluation plans. Furthermore, the operator set can easily be extended and new operators be integrated with existing ones. In fact, Volcano grew over time and new operators were added repeatedly, e.g., for the studies reported in [26], [43]. Volcano's extensibility at the operator level proved particularly valuable when porting Volcano to a (shared-memory) parallel architecture, and led to the "operator model" of parallelizing a query execution system [28]. The new parallelism operator, called exchange in Volcano, is the focus of the next three sections. We first discuss the idea and the semantics of the exchange operator, then describe its implementation for shared memory, and finally generalize this discussion to distributed-memory and hierarchical architectures.

V. Conceptual Overview of Volcano's Exchange Operator

The exchange operator facilitates parallelism by creating processes and process boundaries and by managing data flow and data redistribution between producer and consumer processes. Before discussing any details of the exchange operator, it is important to point out that it is an example of an unusual kind of query evaluation operator and is very different from most database query execution operators. Whereas most database operators implement a logical operation, e.g., the physical operator merge-join implements the logical operator

TABLE I
CLASSIFICATION OF QUERY EVALUATION OPERATORS

Operator Group	Logical Operator or Enforced Property	Volcano Execution Operator
Implementations of Logical Operators	Retrieval Operators	File Scan
		Index Scan
	Select, Project	Filter
	Binary Operations	Nested Loops
		Merge-Join
		Hybrid Hash Join
	Aggregation and Duplicate Removal	Sorting
		Unary Hybrid Hashing
	Relational Division	Naive Division
		Hash-Division
Control Operators (Enforcers)	Sort Order	Sorting
	Plan Robustness	Choose-Plan
	Compression, Decompression	Filter
	Parallelism, Data Partitioning	Exchange

join, the exchange operator provides control beyond that provided by the data manipulation operators and the iterator paradigm. In a nutshell, because all data manipulation operators are implemented as iterators for single-process query evaluation, the exchange operator adds and encapsulates all issues that pertain to parallelism, in particular process creation and termination, data transfer between processes, flow control, and data redistribution.

This kind of operator provides control rather than data manipulation; thus, we call them *control operators*. There are additional control operators in Volcano; the exchange operator is the control operator that facilitates parallelism. If there is an exchange operator in a query evaluation plan (tree), the entire subtree (subplan) below the exchange, down to the next exchange operator if there is one, is executed by newly created producer processes. Thus, the exchange operator permits concurrent execution of complex query evaluation plans in multiple cooperating processes.

Although the ultimate goal of a query evaluation plan is retrieving, selecting, manipulating, and inferring from data stored in a database, 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. We consider it an important advantage of the Volcano design that it separates data manipulation from process control and inter-process communication, because this separation permits design and implementation of new data manipulation algorithms such as N-ary hybrid hash [34] without regard to the execution environment.

Table I summarizes, in a simplified form, Volcano's execution operators for relational query evaluation. Of the two principal groups, one directly implements operators of the logical (relational) algebra, while the other operator group provides query processing control. In the notation of the Volcano Optimizer Generator [30], [32], these operators are called *enforcers* since they enforce physical properties useful or desirable for further query processing. The choose-plan operator enforces plan robustness for unpredictable run-time parameters and resource situations [12], [27], the filter operator (a simple but flexible single-input iterator) not only implements selection and projection (without duplicate removal) but also compresses and decompresses data [29], [34], [54], and the exchange operator ensures that intermediate results be partitioned correctly for the next data manipulation operation.

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, both within and outside the Volcano project. 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. Mapping a sequential plan to a parallel plan by inserting exchange operators into the plan 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 [16].

In spite of the fact that it facilitates parallelism, the exchange operator is an iterator with *open*, *next*, and *close* procedures; therefore, it can be inserted at any one place or at multiple places in a complex query tree. Fig. 3 shows a query execution plan for a three-input join query in a single-process environment using file scans at the leaves, joins for matching, and a print operator for data output. Fig. 4 shows a parallel plan for the same query. The previous plan was parallelized simply by adding exchange operators. Therefore, we call it the operator model of parallelization [28]. The operator model is very different from the bracket model used in the Gamma and Bubba designs and illustrated in Fig. 1. The placement

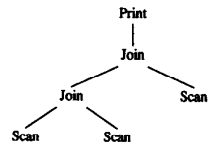

Fig. 3. A query plan for single-process execution.

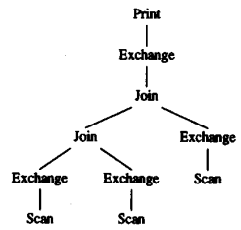

Fig. 4. Operator model of parallelization.

and arguments of exchange operators are chosen based on what forms and degrees of parallelism are desired and based on where data must be repartitioned for correct execution, not by constraints inherent in the exchange operator itself. Exchange operators are placed into query evaluation plans at those points where the optimizer or the experimenter want inter-operator parallelism (pipelining) and where data need to be repartitioned (redistributed) for the data manipulation operators to work correctly.

Fig. 5 shows the processes that could have been created by the exchange operators with suitable argument settings in the query plan of Fig. 4. The design of the exchange operator permits both vertical inter-operator parallelism (program parallelism based on pipelining) and horizontal intra-operator parallelism (data parallelism based on partitioning of datasets into disjoint subsets). The join operators are executed by three processes while the file scan operators are executed by one or two processes each, typically scanning file partitions on different devices. To obtain this grouping of processes, the "degree of parallelism" arguments for the exchange operators have to be set to 2 or 3, respectively, and partitioning functions must be provided for the exchange operators that transfer file scan output to the join processes. The exchange operators are part of both sides of the process boundaries; for example, the topmost exchange operator is part of four processes, namely the print process and all three join processes. The file scan processes can either partition or broadcast their data to all join processes depending on argument settings for the exchange operators.

The exchange operator is the only operator in Volcano that facilitates process boundaries; all other operators are iterators designed and implemented in sequential environments, including the other control operators shown in Table I. Thus, the degree of parallelism set in a given exchange operator as well as the processes created to achieve this parallelism cover the entire group of operators below the exchange operator. The only means to execute some subplan with a different degree of parallelism is to insert another exchange operator into the query evaluation plan.

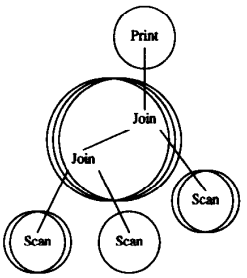

Fig. 5. Vertical and horizontal parallelism.

The exchange operator can be combined freely with all other operators in Volcano, which addresses the two extensibility problems discussed in the Introduction. It can be used to parallelize an existing set of operators, and it permits development of new operators without concern for parallelism. Both directions have been validated within the Volcano project. About 18 months after the project began, we parallelized the then-existing sequential Volcano code by adding the first version of the exchange operator, and we have implemented, debugged, and tuned new algorithms in a single-process environment and parallelized them afterwards by combining them with the exchange operator. In the next two sections, we describe in detail what happens when an exchange operator connects two data manipulation operators.

VI. The Exchange Operator in Shared Memory

The first function of exchange is to provide vertical inter-operator parallelism or pipelining. The *open* procedure allocates a new process after creating a data structure in shared memory called a port for synchronization and data exchange. Thus, when the *open-exchange* completes, two processes participate in executing a query. Processes are allocated from a pool of waiting server processes started earlier, also called "primed" processes.

The older process serves as the consumer and the newly allocated process as the producer in Volcano. The exchange operator in the consumer process acts as a normal iterator; the only difference from other iterators is that it receives its input via inter-process communication rather than iterator (procedure) calls. After allocating the new process, *open_exchange* in the consumer is done. *Next_exchange* waits for data to arrive via the port and returns them a record at a time. *Close_exchange* informs the producer that it can close, waits for an acknowledgment, and returns.

In the producer process, the exchange operator becomes the driver for the query tree below the exchange operator using the *open, next,* and *close* procedures on its input. In order to reduce semaphore interactions and context switches between producer and consumer processes, the output of *next* is collected in packets, which are arrays of pointers to records in the shared buffer. The packet size is an argument for the exchange iterator and can be set between 1 and 32 000 records. When a packet is filled, it is passed to the consumer process. Records in packets are fixed in the shared buffer and must be unfixed by a consuming operator. When its input is exhausted, the exchange

operator in the producer process marks the last packet with an end-of-stream tag before passing it to the consumer, and waits for the consumer to permit process termination. This final delay is necessary because of an idiosyncrasy of the record buffering scheme used in Volcano [28].

While all interaction between operators is based on demand-driven dataflow (iterators, lazy evaluation), the producer-consumer relationship crossing process boundaries inside of exchange uses data-driven dataflow (eager evaluation). A run-time switch of exchange allows augmenting data-driven dataflow with flow control or back pressure using an additional semaphore. If the producer is significantly faster than the consumer, multiple packets queue up in the port. Since all records in the packets are pinned in the buffer, the producer may pin a significant portion of the buffer, thus impeding overall system performance. An argument to the exchange operator determines the initial value of the flow control semaphore, i.e., how many packets the producers may get ahead of the consumers.

Vertical parallelism immediately permits another form of parallelism, called horizontal inter-operator or bushy parallelism. For binary operators, executing both inputs in separate processes results in two input processes proceeding independently in parallel. However, since it is extremely hard to ensure that both producers deliver their output at the right time and the right rate, bushy parallelism is not used in most systems.

The second function of the exchange operator is to permit intra-operator parallelism based on partitioning. The exchange operator in Volcano has two arguments for determining the degree of parallelism for a process group, namely a value that can be set by the optimizer when generating a query execution plan and a support function that can determine the degree of parallelism dynamically at run-time.

Intra-operator parallelism requires data partitioning. Partitioning of stored datasets is achieved by using multiple files, preferably on different devices. Partitioning of intermediate results is implemented by including multiple queues in a port, one for each consumer. The producers use a support function to decide into which of the queues an output record belongs (actually, into which of the packets being filled by the producer). A support function allows implementation of round-robin-, key-range-, and hash-partitioning, i.e., at least all the capabilities realized with split tables in Gamma [18].

In summary, the operator model of parallel query execution as realized in the Volcano execution engine separates data manipulation from parallelism and encapsulates all issues of process control, data transfer, flow control, partitioning, etc. From a software engineering point of view, this separation and its resulting modularity and extensibility are the essential contributions of the Volcano research into parallelism. However, in order to make these advantages available in very high-performance and scalable query processing engines, the limitation to shared memory found in the design described so far had to be broken.

VII. Exchange for Distributed and Hierarchical Memory

After the shared-memory version of Volcano's exchange operator had shown the validity of the operator model of parallel query execution, a distributed-memory version was a natural extension. In this section, we describe this extension in more detail. The important aspects of this extension are that the new exchange operator preserves the extensibility features of the earlier exchange design and, at the same time, permits parallel query execution on essentially all multiple-instruction-multiple-data parallel machines. We first discuss the goals and the implementation of exchange for hierarchical architectures, then assess the required message traffic for startup and shutdown, and finally consider the design for an important future extension of the exchange operator. We omit discussion of the special operating modes described in [28], [33], e.g., broadcasting and separating input streams by producers; suffice it to say that they are supported on shared memory as well as on distributed and hierarchical memory.

The goals for the design and implementation of Volcano's distributed-memory exchange operator were the following. First, the distributed-memory software architecture should support conventional distributed-memory machines like hypercubes as well as shared-memory machines and hierarchical machines with shared-memory parallel nodes. Second, encapsulation should be preserved, i.e., data manipulation operators should not be concerned with parallelism such that these operators can be designed, implemented, debugged, and tuned in a sequential environment. Third, the exchange operator should provide only the mechanisms for parallel query execution, leaving policy decisions to a query optimizer. Fourth, data transfer should be as fast as possible. In particular, shared-memory should be used within each node, and message passing should be used only across node boundaries. Fifth, to ensure that extending Volcano with a new operator requires only implementation of this operator, no separate scheduler process should be required. Scheduling functions should be secondary tasks of processes that participate in executing the query for data manipulation. Sixth, process and communication setup times should be short to allow speedup even for queries on small datasets, and scheduling requirements should be kept to a minimum. Seventh, the code should be portable to environments other than the prototype development environment of UNIX workstations. These goals will be discussed throughout the following subsections.

A. Design and Implementation

The easiest approach to providing a uniform abstraction of parallel query processing in shared- and distributed-memory machines is to base the exchange operator entirely on message passing, which can be supported transparently on both shared- and distributed-memory machines. However, instead of designing an entirely new exchange operator, we chose to separate shared-memory data transfer within a node and distributed-memory data transfer between nodes. The new exchange operator uses shared memory buffers for data transfer within each shared-memory node (as discussed in Section V) and message passing primitives between nodes. In a sense, the same functionality is implemented twice; however, the cost of message passing can be so much higher that this duplication seemed warranted. While some hardware and software systems

exist in which message passing is much less expensive, our portability goal did not permit restricting effective use of parallelism in Volcano to those systems.

The principal design for the new exchange operator follows the hierarchy of shared and distributed memory, and exploits it by using two levels in all global control messages. The lower level is represented by shared-memory communication among processes within one shared-memory node. One of these processes is always a local master, which includes the responsibilities of master processes in process groups as discussed earlier for shared memory plus global communication using message passing between local masters. The upper level is represented by the control communication of local masters with one global master. In order to broadcast a control message such as triggering processes, the originator of this message (being the global master here) sends one message to all local masters. In machines with very many nodes, the performance of this step could be improved by considering all nodes as corners in a binary hypercube and by propagating a broadcast message dimension by dimension, reducing the broadcast delay in point-to-point networks from $O(N)$ to $O(\log N)$. After a broadcast message is propagated to all local masters, they send it to all their local participant processes using efficient shared-memory primitives. A global gathering operation performs the same steps in the opposite direction, i.e., it also exploits the two levels. Note that this discussion only applies to control messages; data packets can be sent from any producer to any consumer without involvement of the local masters. Moreover, data packets within one node are forwarded using efficient shared-memory primitives, and message passing is used only between nodes. Note further that in either extreme cases, i.e., a single shared-memory machine or a conventional distributed-memory machine, this design results in exactly the expected behavior.

One of the prerequisites for query processing using distributed memory is the ability to ship plans and programs across the network. For Volcano query evaluation plans, the three *open*, *next*, and *close* procedures required for each operator were extended by three more procedures, called *size*, *pack*, and *unpack*. They are used to format a tree-shaped query evaluation plan into a network packet and reassemble the plan at the receiving site. For the support functions, the provisions for both compiled and interpreted query processing turned out to be very useful. For distributed-memory query execution, Volcano uses interpreted support functions. The predicate interpreter is linked into all primed processes, and each operator's *pack* routine includes interpretable code of the operator's support functions in the network packet.

Fig. 6 shows a simple query evaluation plan with three data manipulation operators, called P, T, and S. Any actual operators could be in their places, e.g., print, aggregation, and scan. We restricted ourselves to single-input operators in Fig. 6, because a complex tree would make the example harder to follow. Any query evaluation plan, e.g., the one shown in Fig. 4, can be executed on a distributed-memory or hierarchical machine.

Fig. 7 shows a possible parallelization on four nodes for the plan in the previous figure. Other parallelizations would

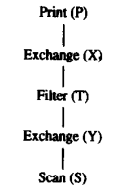

Fig. 6. A simple query evaluation plan.

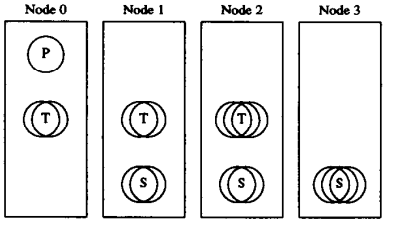

Fig. 7. Nodes, processes, and operators.

have been possible with Volcano's mechanisms, but this will serve as our example. The four rectangles represent nodes; they could be autonomous computers in a distributed system or nodes in a distributed-memory parallel machine. The circles represent processes; let them be called $O_{i,j}$ where O is one of P, T, and S, i is the node number, and j is the process number for the respective operator on that node. For example, the bottom leftmost and rightmost processes in Fig. 7 are called $S_{1,0}$ and $S_{3,2}$, respectively.

The processes called $O_{i,0}$ perform the role of local masters. While all processes can send and receive data, only local master processes exchange control messages across node boundaries. In addition to being local masters on their respective nodes, processes $P_{0,0}, T_{0,0}$, and $S_{1,0}$ are also global masters within their process groups. During query initialization and shutdown, they facilitate inter-node synchronization and exchange of control information when necessary.

Between initialization and shutdown, all processes execute entirely independently from one another. Their progress is only hampered if either no input data are available from the appropriate producer processes, or if flow control stops a group of producer processes that threaten to overrun their consumers. The roles of local and global masters are not distinguished during data production between initialization and shutdown; in particular, any producer process can send network packets with data to any consumer process as dictated by the partitioning function.

For data transfer between nodes, i.e., across the network, both the producer part and the consumer part of an exchange operator create files that only exist in the buffer. The producer "appends" records to the file, and sends an entire cluster across the network when it is full. The consumer reads an entire cluster directly into the buffer pool and uses the normal file scan mechanisms to extract record after record. The important point is that the Volcano software does not require copying on the consumer side, making data transfer as fast as possible. Moreover, since the cluster size can be chosen freely, the

Volcano software supports any packet size. Finally, we are considering extending the exchange operator with mechanisms for compression and decompression of network data.

B. Message Traffic and Overhead

In order to ensure that the exchange operator does not encapsulate parallelism at the expense of added overhead and control messages, the required control messages were kept to a minimum. Moreover, they are independent of the amount of data being processed and of the number of processes participating in a process group; they only depend on the number of shared-memory nodes that participate in the producer and consumer process groups. Fig. 8 shows the messages required between nodes to initiate the processes shown in the previous figure as well as the appropriate data paths. Of course, some of the details here are particular to our choice to build the first prototype of distributed-memory Volcano on top of the UNIX operating system using TCP, and would change if a different operating system or communication protocol were used. The important point is that the number of control messages is quite small, and increases only with the number of nodes executing an operator, not with the number of nodes or processes in the machine.

Notice that all messages in Fig. 8 are exchanged between local masters, i.e., processes numbered $O_{i,0}$, and that all message traffic is encapsulated in the two exchange operators. In other words, all the message between processes in Fig. 8 are really within the exchange operators that straddle the respective process boundaries. First, after $P_{0,0}$ has initiated $T_{0,0}$, $T_{0,0}$ sends a request message to waiting primed processes at nodes 1 and 2 using known ports, shown by solid arrows. This request message includes a packed query evaluation plan and information on how to reach $P_{0,0}$. On each receiving node, one process in the process pool receives the request packet, assumes the role of local master $T_{i,0}$ on node i, and allocates the other local processes, e.g., $T_{1,1}$. Second, the producer processes $T_{i,j}$ connect to the consumer process $P_{0,0}$ using the UNIX connect system call, as indicated by dashed arrows. These connect calls establish all data paths from T operators to the P operator. The need for this handshake between producers and consumers makes UNIX very expensive for distributed-memory query processing; some operating systems for distributed-memory machines do not require it. At this point, the X operator is ready to transfer data, and the X operator in each $T_{i,j}$ process invokes the *open* procedure for its input, i.e., the operator T. Third, when the T operators open their input operator Y, all Y operators in all $T_{i,j}$ processes open input sockets for receiving data from S processes. Information about input sockets is first collected locally by the local masters, and then forwarded to the global master, as indicated by dotted arrows. Fourth, $T_{0,0}$ initiates $S_{1,0}$ which in turn sends a request packet to nodes 2 and 3 to create the local masters $S_{i,0}$ there. While all initiation messages contain information about the consumers' input sockets, query evaluation plans are only included in packets to those nodes in which the plan is not available yet, node 3 in the example. Fifth, all producers S connect to all consumers T to establish

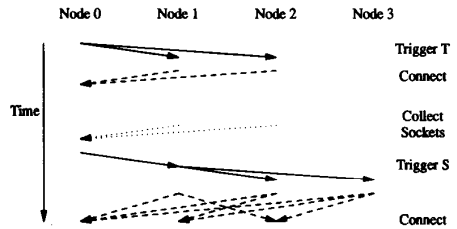

Fig. 8. Control messages between nodes.

all required data paths. At this point, setup is completed and all processes are ready to produce and transfer data.

Processes now proceed on their own, trying to produce data as fast as possible and to ship them to their consumer processes. In other words, data-driven dataflow is used between processes both within and across node boundaries. Flow control within each node is implemented as described earlier. For flow control across node boundaries, the standard UNIX and TCP/IP mechanisms are used, namely a limited buffer associated with each connection.

In general, the mechanisms used for data transfer between nodes are organized similarly to those within nodes. There is only one difference, dictated directly by the accessibility of memory. For data exchange across node boundaries, there are no packets with pointers to records in a shared buffer pool. Instead, each sender maintains a set of output clusters in the buffer, one for each remote consumer. Each time a cluster fills up, it is sent across the network. The receiver also maintains a set of input clusters in the buffer into which the network packet is read as a new file cluster. Note that this organization of clusters requires only one copy step beyond the copy implicit in the send and receive network operations, namely to assemble clusters for individual consumers.

After the initial setup, the only messages exchanged are data packets, plus some low-level acknowledgments as required by the networking software to ensure reliable communication. End-of-stream is indicated by a special field included in data packets; therefore, consumers can count when all their inputs streams are exhausted and propagate end-of-stream notices up the query tree. Producer processes automatically terminate and return to the process pool after flushing all their output packets across the network and waiting for the local master consumer, if one exists, to permit termination (as described earlier for shared memory). Thus, query shutdown does not require any further control messages.

The number of control messages in Fig. 8 is minimal for the environment where ports are created dynamically and therefore cannot be compiled in the query evaluation plan distributed to the nodes. For our example with TCP, the following control messages were required between nodes. Two messages resulted from $T_{0,0}$, the T global master, triggering the local masters on nodes 1 and 2. The five remote T processes connecting to $P_{0,0}$ caused five connect messages. Two messages were due to the collection of sockets by $T_{0,0}$ to enable the S processes to connect later to the T processes. Since the S global master, $S_{0,0}$, is remote from $T_{0,0}$, the triggering of the S process group resulted in three messages

rather than the two required to trigger the S process group. Finally, the connection of the seven S processes to the seven T processes resulted in 39 connect messages. Therefore, for our example using TCP, a total of 51 control messages was required. Our design required only five control messages with the other 46 messages being specific to the underlying communication protocol. The additional messages required by TCP are the collection of sockets (2 messages) and the explicit connects (44 messages). Thus, the design goal of encapsulation was attained, but not at the expense of additional control messages that would not have been required in alternative designs.

In summary, the recent extensions to Volcano's exchange operator provide the mechanisms for parallel query execution in shared-memory, distributed-memory, and hierarchical machines. The extended exchange operator retains all encapsulation properties designed and implemented in the shared-memory version. It effectively shields the data manipulation operators from all parallelism issues. Beyond process management, data transfer, and flow control, the new exchange operator also hides the machine architecture from the data manipulation operators. We believe that this separation of data manipulation and parallelism contributes significantly to Volcano's extensibility and portability and will allow rapid development and parallelization of additional Volcano operators on more systems and architectures. As current shared-memory and distributed-memory machines are subsumed by the hierarchical architecture, the model of parallel query processing in Volcano subsumes those of Bubba [9], [10], [44], Gamma [16], [18], Tandem's NonStop SQL [21], and the Teradata database machine [48], [49], [59] (distributed memory) as well as that of XPRS [39], [40], [58] (shared memory). The versatility of the Volcano software to run on single-processor, shared-memory, distributed-memory, and hierarchical architectures makes it unique among experimental parallel database platforms.

C. Bit Vector Filtering

The exchange operator was originally designed as an encapsulation and abstraction for root-to-leaf initialization and control flow, demand-driven dataflow within processes, data-driven dataflow and flow control between processes, partitioning of data on the producer side of a process boundary, and gathering data on the consumer side of a process boundary. The last task might include merging multiple sorted input streams, which is the dual to partitioning on the producer side (see also earlier discussions of the duality of merging and partitioning for sequential algorithms [34], [35]). One of the capabilities missing from the original exchange design was the ability to pass control information up and down a query plan and across process and processor boundaries. An example for the need of this capability in distributed-memory machines is bit vector filtering, a technique that has been used very effectively to reduce network traffic in distributed-memory query processing systems such as Gamma [16], [17], [23], [53]. In essence, bit vector filtering is a probabilistic semi-join that eliminates many items in the second join input before they are shipped across a network and processed in the join.

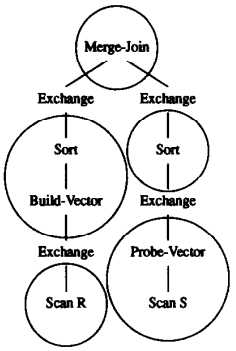

Fig. 9. Parallel query plan with bit vector filtering.

Bit vector filtering consists of two steps, namely building a bit vector with one input and probing the bit vector with the other input. Join keys of items from the input with fewer unique join keys are hashed to a bit in the bit vector, which is raised (set to "1"). After all items from the first input have been processed, the bit vector is shipped to the site originating the second input. Instead of shipping all items, this site ships only items for which the join key hashes to a raised position in the bit vector and discards all other items. This description can easily be generalized to multiple join sites to multiple origination sites for each input, and to setting multiple bits for each item in the first input.

Although a bit vector typically permits some false passes, i.e., some items are shipped and joined even though they have no match in the first join input, the savings in network data transfers are frequently much larger than the added expense of shipping bit vectors across the network. In spite of being an effective performance enhancement in distributed-memory parallel query processing, bit vector filtering cannot be realized in the operator model of parallel query processing as discussed so far.

Consider a parallel merge-join that matches two sets, R and S. Fig. 9 shows a highly parallel query evaluation plan for this query including both pipelining and bushy parallelism. We presume for the time being that partitioning is not used, i.e., no operator shown in Fig. 9 is executed by two or more processes. Building and probing a bit vector can readily be implemented in the Volcano system by using Volcano's filter operator, which is a single-input single-output operator that can perform selections such as normal predicates and testing a bit vector, computations on individual data items such as decompression and relational projection (without duplicate removal), and side-effects such as updates, printing, and creating a bit vector [33].

The difficulty of integrating bit vector filtering into the operator model of parallel query processing lies in moving the bit vector from the build-vector operator to the probe-vector operator without violating the encapsulation objective. Earlier system designs such as Gamma use a separate scheduler process for each query, which can perform this task, but at the expense of poor encapsulation.

The situation in Fig. 9 is deliberately chosen to be the worst case; with hash join, the scenario would be straightforward for

two reasons. First, hash join first consumes its first input before starting to consume the second input. This division into two phases makes it much easier to apply bit vector filtering, which requires that one input be processed completely for building the filter before probing the filter with the second input is begun. Second, as a consequence of the first reason, the bit vector can be created at the join site by the hash join operator and process, and must only be passed down from the join through the exchange to the probe-vector operator. Passing a bit vector down a query tree is not difficult in Volcano, because this is exactly the function of the bindings parameter passed from one *open* procedure to the next when a query evaluation plan in activated. The original intent of introducing the bindings parameter was to permit delayed optimization decisions at run-time using Volcano's choose-plan operator [27], but it is equally useful for passing a bit vector down from a join to a probe-vector operator. Since the exchange operator passes and ships this parameter (which must include an encoding of its size) from a master consumer process to the producer process group, passing a bit vector from a join operator to the probe-vector operator can be achieved without violating or changing the exchange operator's design.

In a query evaluation plan based on merge-join, however, as in Fig. 9, the two subtrees execute concurrently in bushy parallelism, which makes it impossible to ensure that the bit vector is built completely before being probed with the second input (ignoring temporarily the issue of moving the bit vector from the build-vector operator to the probe-vector operator). Thus, the exchange operator between the merge-join and the left sort must perform a *synchronized open*. In other words, the *open-exchange* procedure invoked by the merge-join must not complete until the *open-sort* procedure has completed and returned to the driver component of the exchange operator between the merge-join and the left sort. While the synchronized *open* eliminates bushy parallelism between the two sorts in Fig. 9 except during the two final merges, it is the goal of combining merge-join with bit vector filtering that requires this restriction. The restriction is imposed neither by the exchange operator nor by the operator model of parallel query execution; the exchange operator only provides the mechanisms for a variety of parallel query execution strategies.

A synchronized *open* is a new requirement for the exchange operator, which will increase the message traffic between producers and consumers in a distributed-memory architecture. The required messages are exactly those discussed earlier for a global gathering operation among all producers, a message from the global master producer to the global master consumer, and a global broadcast among all consumers. Thus, the additional synchronization point, which is required by the logic of bit vector filtering, can be implemented with minimal overhead and cost.

In addition to the synchronization problem, the bit vector must be moved from the build-input operator to the probe-input operator. In order to retain the encapsulation properties, the bit vector should be passed only along the edges of the tree representing the query processing plan. Thus, the bit vector must first be passed from the build-vector operator to the merge-join and then from the merge-join to the probe-vector operator. The latter part can be handled precisely as for hash join. Passing the filter up the left side of Fig. 9 requires augmentation of the Volcano iterator protocol and some modifications of the exchange operator.

The bit vector must be created on the input side of the sort, because the filter is usable only after all data items have passed through the operator building the filter and the sort output is pipelined directly into the merge-join. In other words, the newly created bit vector is passed from the build-vector operator to its consumer when the build-vector operator *closes*. In order to facilitate this, we augmented the interface to the *close* iterator procedure with bindings being passed up the query evaluation plan. In fact, we generalized it further to permit bindings to be passed up and down the edges of a query evaluation tree both during the activation phase (call and return of *open*) and deactivation phase (call and return of *close*) of query execution. Thus, we can ensure that when the sort operator closes its input, it will receive an initialized bit vector from the build-vector operator.

Since a sorting operation cannot produce any output before inspecting the last input item, Volcano's sort iterator performs all its work except for the last merge step as part of its *open* procedure and the *open-sort* procedure *closes* the sort's input iterator. Thus, the bit vector returned by the build-vector operator to the left sort in Fig. 9 can be passed to the exchange operator in the return of the *open-sort* procedure and then further up the query evaluation plan to the merge-join as part of the synchronized *open*. Since we discussed earlier how a join operator can pass a bit vector down to the probe-vector operator, we have discussed all mechanism required for build vector filtering as shown in Fig. 9, under the assumption that no intra-operator parallelism is used.

If intra-operator parallelism and partitioning are used, there are several bit vectors that are created and used in the parallel processes. When bit vectors are passed up from the left sort through the exchange operator to the merge-join in Fig. 9, these bit vectors need to be superimposed into a single vector using bit-wise "or" operations, and the resulting bit vector is passed to each of the merge-joins. In order to support operations on bit vectors such as superimposing or concatenation, the definition and implementation of the exchange operator requires additional support functions, i.e., functions invoked by the exchange operator at suitable times to modify the bindings parameters (e.g., the bit vectors) passed up or down in a query plan. Moreover, since the bindings parameters are created and used in multiple parallel processes and possibly in multiple nodes of a distributed-memory machine, these operations require that the bindings parameter be passed to and from local and global masters. In other words, in order to merge multiple bit vectors created at multiple sites, a global gathering operation must be performed. Fortunately, the bit vectors can be included in the network packets already mentioned to ensure synchronized *open*; thus, no additional network overhead is required.

In summary, extending the exchange operator to support bit vector filtering is relatively straightforward, making this effective and inexpensive technique available not only to

relational but also extensible database query processing. While the Volcano iterator interface and the operator model of parallel query processing requires some modifications such as the synchronized *open*, the basic paradigm remains unchanged.

VIII. Extensibility

One reason for using the operator model of parallel query execution in Volcano was its extensibility, i.e., the freedom to combine the exchange operator with any existing or new data manipulation operator that provides and uses the uniform iterator interface. In order to validate Volcano's extensibility, we integrated numerical computations into the Volcano framework, which are normally not found in database management systems. For scientific database systems to become accepted, they must offer at least competitive performance to file systems currently used in scientific computing. In other words, the additional software layers for database management must be counterbalanced by database performance techniques normally not used in scientific computations. We believe that automatic optimization and parallelization are prime candidates for such techniques, and are applying automatic optimization and parallelization to scientific computations [60]. Our approach is to specify a logical algebra that includes both numerical, scientific operators and database pattern matching operators, to define and implement a set of algorithms in the Volcano framework that can execute the operators of the logical algebra, to create an optimizer with the Volcano Optimizer Generator that maps an expression over the logical algebra into the optimal execution plan, and to use the exchange operator to parallelize execution plans over scientific databases [30], [32], [60].

We have already added two scientific operators to Volcano, which we call the interpolation and sampling operators. The interpolation operator is capable of performing digital filtering, data reduction, and data interpolation and extrapolation on sequences of numerical data, e.g., time series. The interpolation operator has verified the feasibility of integrating such operations into an extensible database query execution engine, while also providing some initial insights into the different means and needs of parallel computation in traditional and scientific database systems.

We explain our research into parallel execution in scientific databases by means of an example execution plan. This is a simplified model of a high altitude cloud model using both ground measurements of temperature, pressure, etc., and spacecraft data on high altitude water densities. These two data sets are the inputs into the calculation. Satellite data is arriving in real-time, and is driving the entire computation. Ground observation data is available directly in a database file. Neither data set provides complete coverage of the atmosphere, thus interpolation operations are applied to fill in gaps in the data. Interpolation can be accomplished by the existing Volcano interpolation operator. From these sets of data, observations near to each other are combined; for example, observations within 10 km might be used together in the final calculations. Other uses for matching data from multiple sources in scientific data management include combining raw data with calibration data or matching current data with historical data

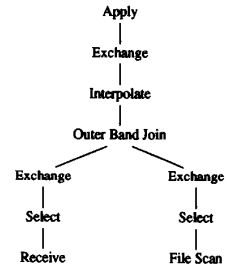

Fig. 10. Computation with exchange operators.

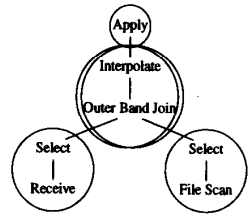

Fig. 11. Processes in the parallel computation.

(e.g., similar weather patterns). Matching of data values that fall within a range of each other is called a band join, and has been studied in [19]. With the matching data, a selection is performed to isolate the area of interest. Finally, the cloud cover calculations are applied, the data is graphed, etc. Such calculations can be performed by the existing Volcano filter operator using its "apply" support function.

To exploit parallel processing capabilities, exchange operators are inserted into the computation, for example below the join and above the interpolator as shown in Fig. 10. These operators perform process management and data transfer between processes, including flow control. In Fig. 11, the exchange operators are used to exploit parallelism: one process for each of the input selection subtrees, two processes working together to perform the join and the interpolation (each on half of the data), and one process performing the final calculation and display.

Our experience so far has shown that the iterator model is quite flexible and useful for scientific computations, in particular in single-dimensional arrays and time series, and that the exchange operator is perfectly suitable for parallel execution of scientific operators (iterators). At the same time, we have identified the need for multicast in some computations. Volcano's exchange operator currently does not support multicast, but we had already planned on adding it for other database operations, e.g., symmetric fragment-and-replicate joins [56] and specialized versions of parallel hash-division [26]. A second new requirement for the exchange operator is data parallelism analogous to vertical partitioning in the database world. For example, in order to perform digital filtering of multiple channels of a time series in parallel, each element in the time series is divided and the individual measurements are sent to different processes, which requires another minor extension of the operator model of parallel query execution and the existing Volcano exchange code. Thus,

we believe that the exchange operator as explored in Volcano is very extensible and that it is a good basis for parallel query execution in extensible and scientific as well as object-oriented database systems. Since it is much easier to develop and maintain numerical scientific as well as nonnumerical database operations in single-process environments, both because it is easier to grasp the interactions of program components and because more and better software tools are available, the separation of data manipulation and parallel execution concerns offers a significant advantage for database query processing software for conventional and new application domains.

IX. Summary and Conclusions

In this paper, we have described extensions of earlier work on parallel query evaluation and on encapsulation of parallelism in a novel operator, called the exchange operator in Volcano. In a previous paper [28], we described the shared-memory version; here we detailed the design and implementation for distributed-memory parallel machines. Encapsulation of parallelism in a single operator, the important and new property of the operator model introduced in [28], has been maintained in the extension from the shared-memory version to the distributed-memory version of the exchange operator. The data manipulation operators did not require any modifications, neither when extending Volcano from single-process to shared-memory parallel query processing nor when extending it further to distributed-memory architectures. One and the same implementation of these operators can be used very efficiently in single-process, single-machine, and multi-machine query evaluation. Thus, the exchange operator makes the design and implementation of all other query processing algorithms and operators architecture-independent. Due to the uniform iterator interface, the exchange operator could even parallelize new operators unknown at this time, e.g., a particularly efficient join algorithm for three inputs or a new operator for scientific databases.

Volcano's design and implementation also supports a hierarchical combination of the two conventional architectures, i.e., a loosely-coupled group of shared-memory machines. Machines of this general architecture have been designed and are becoming commercially available. We plan on porting Volcano to these new architectures as soon as such machines become available to us. Conventional parallel machines, both shared-memory and distributed-memory, are extreme cases of the hierarchical design. A conventional shared-memory machine represents a single node in this new architecture, while a distributed-memory machine uses nodes with single CPU's. Accordingly, software running on the hierarchical design runs, without modification, on both conventional designs.

The hierarchical design offers significant advantages of both conventional designs, and therefore over both conventional designs. For small degrees of parallelism, using shared memory allows fast synchronization and communication. For large degrees of parallelism, distributed-memory designs offer scalability to very large machines without the danger of bus saturation inherent in shared-memory architectures. Both of these advantages are significant not only for database query evaluation but also for all other applications, numerical and nonnumerical ones alike. However, different applications have different communication-to-computation ratios, and the maximal degree of parallelism to which shared-memory machines can be scaled varies for different applications. Furthermore, as different hardware components (CPU's, buses, memories, secondary storage devices) become faster at different paces, the computation-to-communication time ratio changes. Thus, it is essential for successful future architectures to be modular, meaning that CPUs, disks, etc., can be replaced individually and reconnected using buses and network interconnections to optimally exploit the hardware for an application. For database software, it is essential to be flexible enough to adapt to new architectures and architecture modifications.

Since parallel architectures using hierarchical designs, i.e., shared-memory parallel nodes connected by high-speed networks, are forthcoming from several hardware vendors, it is important that the new version of Volcano's exchange operator has demonstrated the feasibility of combining extensibility, parallelism, and portability for database query processing on forthcoming computer architectures by encapsulating all parallelism issues as well as the underlying hardware architecture. For commercial DBMS vendors who plan to support parallelism on shared- and distributed-memory architectures and to capitalize on hierarchical architectures in the near future, the exchange operator offers the important advantage of parallelizing existing query processing code on a variety of computer architectures. For extensible database systems, including scientific and object-oriented database systems, the exchange operator presents an effective means for exploiting parallelism and parallel architectures, thus permitting database system designers to combine extensibility and high functionality with parallelism and high performance.

Acknowledgment

We appreciate the reviewers' and the editor's excellent suggestions to earlier versions of this paper.

References

[1] E. Babb, "Implementing a relational database by means of specialized hardware," *ACM Trans. Database Syst.*, vol. 4, no. 1, p. 1, Mar. 1979.

[2] 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. Softw. Eng.*, vol. 14, no. 11, p. 1711, Nov. 1988.

[3] D. S. Batory, T. Y. Leung, and T. E. Wise, "Implementation concepts for an extensible data model and data language," *ACM Trans. Database Syst.*, vol. 13, no. 3, p. 231, Sept. 1988.

[4] A. Bhide, "An analysis of three transaction processing architectures," in *Proc. Int. Conf. Very Large Data Bases*, Los Angeles, CA, Aug. 1988, p. 339.

[5] A. Bhide and M. Stonebraker, "A performance comparison of two architectures for fast transaction processing," in *Proc. IEEE Conf. Data Eng.*, Los Angeles, CA, Feb. 1988, p. 536.

[6] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking database systems: A systematic approach," in *Proc. Int. Conf. Very Large Data Bases*, Florence, Italy, Oct.–Nov. 1983, p. 8.

[7] H. Boral and D. J. DeWitt, "Database machines: An idea whose time has passed? A critique of the future of database machines," in *Proc. Int. Workshop Database Machines*, Munich, 1983, p. 166. Reprinted in A. R. Hurson, L. L. Miller, and S. H. Pakzad, *Parallel Architectures*

for Database Systems, IEEE Computer Society Press, Washington, DC, 1989.

[8] H. Boral and D. J. DeWitt, A "Methodology for database system performance evaluation," *Proc. ACM SIGMOD Conf.,* Boston, MA, June 1984, 176.

[9] H. Boral, "Parallelism in Bubba," in *Proc. Int. Symp. Databases in Parallel and Distributed Syst.,* Austin, TX, Dec. 1988, p. 68.

[10] 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. Knowledge Data Eng.,* vol. 2, no. 1, p. 4, Mar. 1990.

[11] D. R. Cheriton, H. A. Goosen, and P. D. Boyle, "Paradigm: A highly scalable shared-memory multicomputer," *IEEE Comput.,* vol. 24, no. 2, p. 33, Feb. 1991.

[12] R. L. Cole and G. Graefe, "A dynamic plan optimizer," submitted for publication, Feb. 1993.

[13] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in *Proc. ACM SIGMOD Conf.,* Chicago, IL, June 1988, p. 99.

[14] W. Davison, "Parallel index building in Informix OnLine 6.0," in *Proc. ACM SIGMOD Conf.,* San Diego, CA, June 1992, p. 103.

[15] D. J. DeWitt and P. B. Hawthorn, "A performance evaluation of database machine architectures," in *Proc. Int. Conf. Very Large Data Bases,* Cannes, France, Sept. 1981, p. 199.

[16] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, "GAMMA—A high performance dataflow database machine," in *Proc. Int. Conf. Very Large Data Bases,* Kyoto, Japan, Aug. 1986, p. 228. Reprinted in M. Stonebraker, *Readings in Database Systems.* San Mateo, CA: Morgan-Kaufman, 1988.

[17] D. J. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A performance analysis of the GAMMA Database Machine," in *Proc. ACM SIGMOD Conf.,* Chicago, IL, June 1988, p. 350.

[18] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The Gamma Database Machine Project," *IEEE Trans. Knowledge Data Eng.,* vol. 2, no. 1, p. 44, Mar. 1990.

[19] D. J. DeWitt, J. E. Naughton, and D. A. Schneider, "An evaluation of non-equijoin algorithms," in *Proc. Int. Conf. Very Large Data Bases,* Barcelona, Spain, 1991, p. 443.

[20] D. J. DeWitt and J. Gray, "Parallel database systems: The future of high-performance database systems," *Commun. ACM,* vol. 35, no. 6, p. 85, June 1992.

[21] S. Englert, J. Gray, R. Kocher, and P. Shah, "A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases," Tandem Computer Syst. Tech. Rep. 89.4, May 1989.

[22] M. T. Fang, R. C. T. Lee, and C. C. Chang, "The idea of declustering and its applications," in *Proc. Int. Conf. Very Large Data Bases,* Kyoto, Japan, Aug. 1986, p. 181.

[23] R. H. Gerber, "Dataflow query processing using multiprocessor hash-partitioned algorithms," Ph.D. dissertation, Univ. of Wisconsin–Madison, Oct. 1986.

[24] S. Ghandeharizadeh and D. J. DeWitt, "Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines," in *Proc. Int. Conf. Very Large Data Bases,* Brisbane, Australia, 1990, p. 481.

[25] J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor," CS Tech. Rep. 766, Univ. of Wisconsin–Madison, Apr. 1988.

[26] G. Graefe, "Relational division: Four algorithms and their performance," in *Proc. IEEE Conf. Data Eng.,* Los Angeles, CA, Feb. 1989, p. 94.

[27] G. Graefe and K. Ward, "Dynamic query evaluation plans," in *Proc. ACM SIGMOD Conf.,* Portland, OR, May–June 1989, p. 358.

[28] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," in *Proc. ACM SIGMOD Conf.,* Atlantic City, NJ, May 1990, p. 102.

[29] G. Graefe and L. D. Shapiro, "Data compression and database performance," in *Proc. ACM/IEEE-CS Symp. Appl. Comput.,* Kansas City, MO, Apr. 1991.

[30] 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, Eds. San Mateo, CA: Morgan-Kaufman, 1993.

[31] G. Graefe and S. S. Thakkar, "Tuning a parallel database algorithm on a shared-memory multiprocessor," *Software—Practice and Experience,* vol. 22, no. 7, p. 495, July 1992.

[32] G. Graefe and W. J. McKenna, "The Volcano Optimizer Generator: Extensibility and efficient search," in *Proc. IEEE Conf. Data Eng.,* Vienna, Austria, Apr. 1993, p. 209.

[33] G. Graefe, "Volcano, An extensible and parallel dataflow query processing system," *IEEE Trans. Knowledge Data Eng.,* vol. 5, no. 5, Oct. 1993.

[34] ——, "Query evaluation techniques for large databases," *ACM Comput. Surveys,* vol. 25, no. 2, p. 73, June 1993.

[35] G. Graefe, A. Linville, and L. D. Shapiro, "Sort versus hash revisited," *IEEE Trans. Knowledge Data Eng.,* to be published, 1993.

[36] J. Gray, "A census of Tandem system availability between 1985 and 1990," Tandem Computers Tech. Rep. 90.1, Jan. 1990.

[37] L. Haas, J. C. Freytag, G. Lohman, and H. Pirahesh, "Extensible query processing in Starburst," in *Proc. ACM SIGMOD Conf.,* Portland, OR, May–June 1989, p. 377.

[38] 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. Knowledge Data Eng.,*vol. 2, no. 1, p. 143, Mar. 1990.

[39] W. Hong and M. Stonebraker, "Optimization of parallel query execution plans in XPRS," in *Proc. Int. Conf. Parallel and Distributed Inform. Syst.,* Miami Beach, FL, Dec. 1991.

[40] W. Hong, "Exploiting inter-operation parallelism in XPRS," in *Proc. ACM SIGMOD Conf.,* San Diego, CA, June 1992, p. 19.

[41] H. I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in *Proc. IEEE Conf. Data Eng.,* Los Angeles, CA, Feb. 1990, p. 456.

[42] K. A. Hua and C. Lee, "An adaptive data placement scheme for parallel database computer systems," in *Proc. Int. Conf. Very Large Data Bases,* Brisbane, Australia, Aug. 1990, p. 493.

[43] T. Keller, G. Graefe, and D. Maier, "Efficient assembly of complex objects," in *Proc. ACM SIGMOD Conf.,* Denver, CO, May 1991, p. 148.

[44] S. Khoshafian and P. Valduriez, "Parallel execution strategies for declustered databases," in *Proc. 5th Int. Workshop Database Machines,* Karuizawa, Japan, Oct. 1987.

[45] M. Kitsuregawa and Y. Ogawa, "Bucket spreading parallel hash: A new, robust, parallel hash join method for skew in the super database computer (SDC)," in *Proc. Int. Conf. Very Large Data Bases,* Brisbane, Australia, Aug. 1990, p. 210.

[46] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Henessy, M. Horowitz, and M. S. Lam, "The Stanford Dash Multiprocessor," *IEEE Comput.,* vol. 25, no. 3, p. 63, Mar. 1992.

[47] B. Lindsay, L. Haas, C. Mohan, P. Wilms, and R. Yost, "Computation and communication in R*," *ACM Trans. Comput. Syst.,* vol. 2, no. 1, p. 24, Feb. 1984.

[48] P. M. Neches, "Hardware support for advanced data management systems," *IEEE Comput.,* vol. 17, no. 11, p. 29, Nov. 1984.

[49] P. M. Neches, "The Ynet: An interconnect structure for a highly concurrent data base computer system," in *Proc. 2nd Symp. Frontiers of Massively Parallel Computation,* Fairfax, Oct. 1988.

[50] H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger, "Parallelism in relational data base systems: Architectural issues and design approaches," in *Proc. Int. Symp Databases in Parallel and Distributed Syst.,* Dublin, Ireland, July 1990, p. 4.

[51] J. P. Richardson, H. Lu, and K. Mikkilineni, "Design and evaluation of parallel pipelined join algorithms," in *Proc. ACM SIGMOD Conf.,* San Francisco, CA, May 1987, p. 399.

[52] J. E. Richardson and M. J. Carey, "Programming constructs for database system implementation in EXODUS," in *Proc. ACM SIGMOD Conf.,* San Francisco, CA, May 1987, p. 208.

[53] D. Schneider and D. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proc. ACM SIGMOD Conf.,* Portland, OR, May–June 1989, p. 110.

[54] L. D. Shapiro, S. Ni, and G. Graefe, "Full-time data compression: An ADT for database performance," submitted for publication, 1993.

[55] J. Shemer and P. M. Neches, "The genesis of a database computer," *IEEE Comput.,* vol. 17, no. 11, p. 42, Nov. 1984.

[56] J. W. Stamos and H. C. Young, "A symmetric fragment and replicate algorithm for distributed joins," IBM Tech. Rep. RJ7188, San Jose, CA, Dec. 5, 1989.

[57] M. Stonebraker, "The case for shared-nothing," *IEEE Database Eng.,* vol. 9, no. 1, Mar. 1986.

[58] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The design of XPRS," in *Proc. Int. Conf. Very Large Data Bases,* Los Angeles, CA, Aug. 1988, p. 318.

[59] Teradata, *DBC/1012 Data Base Computer, Concepts and Facilities,* Teradata Corp., Los Angeles, CA, 1983.

[60] R. H. Wolniewicz and G. Graefe, "Algebraic optimization of computations over scientific databases," in *Proc. Int. Conf. Very Large Data Bases,* Dublin, Ireland, Aug. 1993, p. 13.

[61] E. Wong and R. H. Katz, "Distributing a database for parallelism," in *Proc. ACM SIGMOD Conf.,* San Jose, CA, May 1983, p. 23.

Goetz Graefe was an undergraduate student in business administration and computer science in Germany before receiving M.S. and Ph.D. degrees in computer science in 1984 and 1987 from the University of Wisconsin–Madison. His thesis work was the EXODUS Optimizer Generator.

He has served on the faculties of the Oregon Graduate Institute and the University of Colorado at Boulder, and is the principal investigator of both the Volcano project on extensible query processing and, with David Maier, the REVELATION project on query processing and performance in object-oriented database systems. Since 1992, he has been an Associate Professor of Computer Science at Portland State University, Portland, OR. He is currently working on extensions to Volcano including a new optimizer generator, request processing in object-oriented and scientific database systems, optimization and execution of very complex database queries, and physical database design.

Diane L. Davison received the B.S. degree in computer science from the University of Kansas in 1980, and the M.S. degree in computer science from the University of Colorado at Boulder in 1991.

She is currently pursuing the Ph.D. degree at the University of Colorado at Boulder. Her research interests include parallel database query execution and dynamic resource allocation.