Architecture-Independent Parallel Query Evaluation In Volcano

Goetz Graefe
Diane Davison

CU-CS-500-90

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Architecture-Independent Parallel Query Evaluation in Volcano

Goetz Graefe, Diane Davison

CU-CS-500-90 November 1990

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430

(303) 492-7514
(303) 492-2844 Fax
graefe@boulder.colorado.edu

Architecture-Independent Parallel Query Evaluation in Volcano

Goetz Graefe, Diane Davison
University of Colorado at Boulder

Abstract

To investigate the tradeoffs of shared-memory and distributed-memory parallel computer architectures for database query evaluation, we have designed and implemented a query execution system called Volcano. Its *exchange* operator shields the data manipulation operators from all parallelism issues, thus encapsulating parallelism. It has recently been extended to support not only shared-memory but also distributed-memory and hierarchical architectures, i.e., a closely-tied group of shared-memory machines. In this report, we detail design and implementation of the operator, report on performance observations, and argue that the parallel database server of the future should be a modular hierarchical design.

1. Introduction

From a large number of research projects and their benchmarks, e.g. [4, 5, 13-15, 19, 30, 31, 33, 39, 43, 45, 46, 49, 51, 53, 59, 60, 64, 68], it is obvious that database query evaluation can benefit significantly from parallel processing. However, it is not clear which of two 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 systems is that they are scalable and can avoid bottlenecks due to bus saturation. What, then, is the best architecture to use, or what circumstances determine the best architecture?

In order to investigate this question experimentally, we have designed and implemented a query evaluation system called Volcano that allows parallel execution of a large set of operators on either architecture [32]. Due to separation of data manipulation and control of parallelism, only one, novel operator deals with parallelism issues; all other operators (which could be called the "work" operators, e.g., file scan, sort, and 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.

All operators 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 *open*, *next*, and *close*. Iterators can be nested in complex trees or plans in which each operator is scheduled by its consumer and schedules its producer or producers by means of calls to these procedures, making query evaluation in a single process self-scheduling with minimal overhead and therefore very efficient.

The module responsible for parallel execution and synchronization is called the *exchange* iterator in Volcano. Notice that it 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. Figure 1 shows a complex query execution plan that includes data processing operators, i.e., file scans and joins, and exchange operators. The exchange operator can be freely combined with all other operators, and it can be used multiple times in a query evaluation plan.

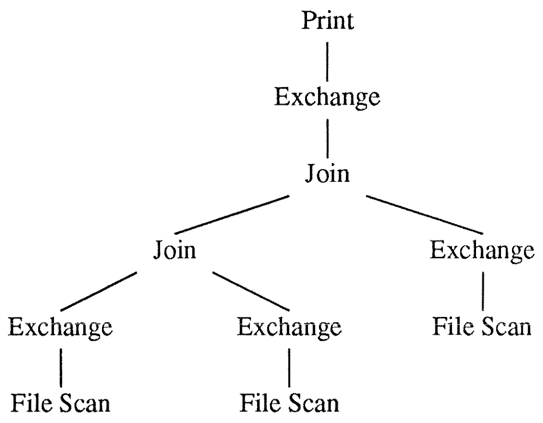

Figure 1. Operator Model of Parallelization.

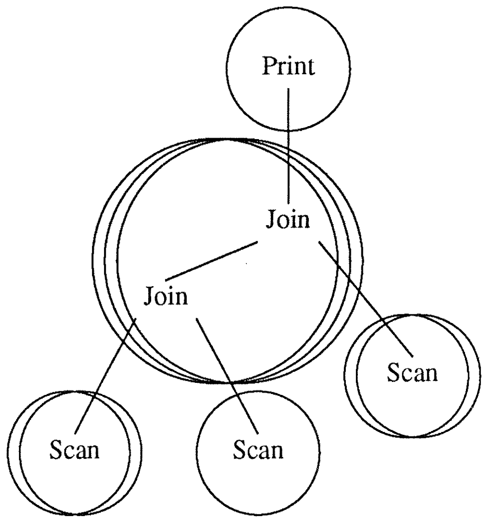

Figure 2. Vertical and Horizontal Parallelism.

Figure 2 shows the processes that could have been created by the exchange operators with suitable argument settings in the query plan of the previous figure. 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 operate on both sides of the process boundaries; for example, the top-most exchange operator is part of four processes, namely the print process and all three join processes. The file scan processes can transfer data to all join processes; however, data transfer between the join operators occurs only within each of the join processes. Unfortunately, this restriction renders this parallelization infeasible if the two joins are on different attributes and partitioning-based parallel join methods are used (rather than fragment-and-replicate methods [4, 16, 18, 61]). For this case, a variant of exchange is supported in Volcano's exchange operator called *interchange* which would be placed between the two joins. This variant is described in [29, 32].

The implementation of Volcano proceeded in three phases. First, only single-process query evaluation was supported. Second, parallel query processing on shared-memory systems was supported using the exchange operator as described in [29, 32]. This version of Volcano and its exchange operator was a proof-of-concept prototype for the operator model of parallel query evaluation and for the integration of parallelism and extensibility. Third, the exchange operator was modified further to support both shared-memory and distributed-memory machines. The third phase, its motivation, design, implementation, and conclusions, are the subject of this report.

When extending the exchange operator to support query processing on distributed-memory machines, we did not want to give up the advantages of shared memory, namely fast communication and synchronization. A recent investigation demonstrated that shared-memory architectures can deliver near-linear speed-up for limited degrees of parallelism; we observed a speed-up of 14.9 with 16 CPUs for parallel sorting in Volcano [30]. To combine the best of both worlds, we have built our software such that it runs on a closely-tied group, e.g., a hypercube or mesh architecture, of shared-memory parallel machines. We can now investigate query processing on hierarchical architectures and heuristics of how CPU and I/O power as well as memory can best be placed and exploited in such machines.

Figure 3 shows a generic hierarchical architecture. The important point is the combination of local busses within shared-memory parallel machines and a global interconnection network between machines. The diagram is only a very general outline of such an architecture; many details are deliberately left out and unspecified. The network could be implemented using a bus (such as an ethernet [50]), ring, hypercube, mesh, or point-to-point connections. The local busses may or may not be split into code and data or by address range to achieve higher bus bandwidth, less bus contention, and hence higher scalability limits for the use of shared memory. Design and placement of caches, disk controllers, terminal connections, and local- and wide-area network connections are also left open. Tape drives or other backup devices would probably be connected to local busses.

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

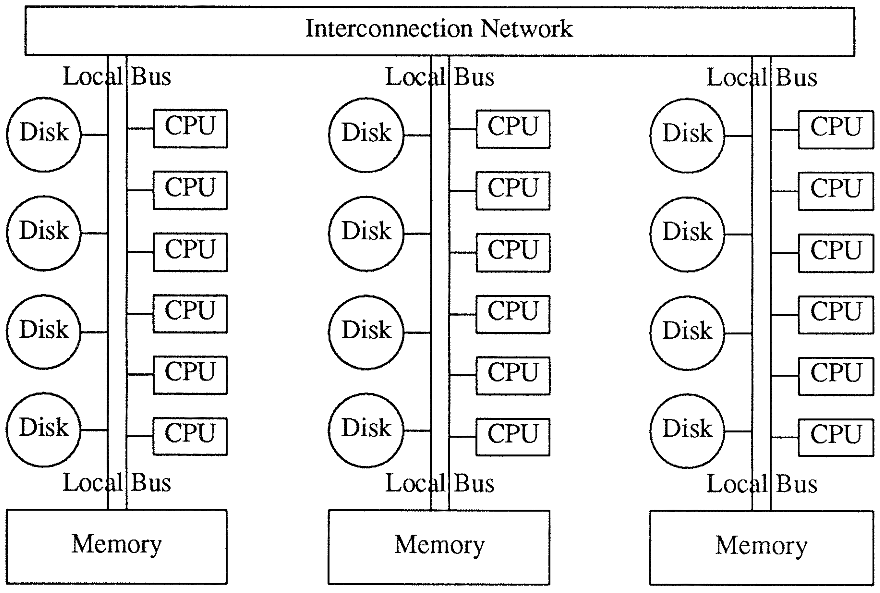

Figure 3. A Hierarchical-Memory Architecture.

machine with 4×16 CPUs into one with 8×8 CPUs.

This architecture may also be exploited for reliability and availability. Tandem's mirroring architecture could be recreated by pairing shared-memory machines and their storage devices. Long-distance logging, task migration, and recovery could also be designed and implemented [10, 48]. High availability also requires that components can be replaced while the rest of the machine is still operating.

Most of today's parallel machines are built as one of the two extreme cases of this hierarchical design: a distributed-memory machine uses single-CPU nodes, while a shared-memory machine consists of a single node. Software designed for this hierarchical architecture will run on either conventional design as well as a genuinely hierarchical machine, and will allow exploring tradeoffs in the range of alternatives in between. Thus, the operator model of parallelization also offers the advantage of architecture- and topology-independent parallel query evaluation.

The remainder of this report is organized as follows. After a survey of related research in Section 2, Section 3 provides a brief overview of Volcano. Volcano's exchange operator is described in Section 4, both for shared memory and distributed memory. Section 5 contains performance measurements for Volcano to support the arguments for both shared-memory- and distributed-memory query evaluation. Section 6 contains a brief summary and our conclusions from this effort.

2. Related Work

There are several hardware designs that attempt to overcome the shared-memory scaling problem, e.g., the DASH project [2, 23, 44], and the Wisconsin Multicube [24, 25]. 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.

Many database research projects have investigated hardware architectures for parallelism in database systems. Stonebraker compares shared-nothing (distributed-memory), shared-disk (distributed-memory with multi-ported disks), and shared-everything (shared-memory) architectures for database use [63] based on a number of issues including scalability, communication overhead, locking overhead, and load balancing. 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 big benefit of large 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 [6, 7] and conclude that a shared-everything (shared-memory) design provides 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 by Pirahesh et al. [55], but neither of these authors elaborate on the idea's generality or potential.

For query processing, customized parallel hardware was investigated for numerous database machine projects but largely abandoned after Boral and DeWitt's influential analysis [8] that compares CPU and I/O speeds and their trends and concludes that I/O is most likely the bottleneck in future high-performance query execution, not processing. Therefore, they recommend moving from research on custom processors to methods 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 came to the same conclusion that parallelism is no substitute for effective storage structures and query execution algorithms [12, 62]. Subsequently, both Boral and DeWitt embarked on new database machine projects, Bubba and Gamma, that ran customized software on standard processors with local disks [1, 9, 13-]. For scalability and availability, both projects used distributed-memory hardware with single-CPU nodes, and investigated scaling questions for very large configurations [20, 21].

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 [15, 68]. Since Tandem uses its own hardware, it is not clear how easily the product could be moved to a hierarchical architecture.

The XPRS system, on the other hand, is being built on shared memory [64, 65]. Its designers believe that modern bus architectures can handle up to 2,000 transactions per second (TPS). They provide automatic load balancing and faster communication than shared-nothing machines and are equally reliable and available for most errors, i.e., media failures, software, and operator errors [34, 35]. However, we believe that attaching 250 disks to a single machine as necessary for 2,000 TPS [64] requires significant special hardware, e.g., channels or I/O processors, and it is quite likely that the investment for such hardware can have greater impact on overall system performance if spent on general-purpose CPUs or disks. Without such special hardware, the performance limit for shared-memory machines is probably much lower than 2,000 TPS. Furthermore, there already are applications that require larger storage and access capacities, e.g., the Japanese national social insurance database planned for about 1,000 disks of 400 MB each [52].

Richardson et al. [57] performed an analytical study of parallel join algorithms on a hierarchical architecture. They assumed a group of "clusters" connected with a global bus with multiple microprocessors and shared memory in each cluster. Disk drives were attached to the busses within clusters. However, their analysis suggested that the best performance are 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 CPUs (e.g., 5 μsec for a comparison, about 2–5 MIPS), a slow network (a bus with typically 100 Mbit/sec), and a modest number of CPUs in the entire system (128). It would be very interesting to see the analysis with larger relations (e.g., 1–10 GB), more and faster CPUs (e.g., $1{,}000 \times 30$ 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" [67] that might provide guidance in declustering and replication to reduce data movement between nodes. Other work on declustering and limiting declustering includes [11, 17, 22, 37, 38].

Kitsuregawa and Ogawa are have designed a new database machine called SDC [42]. Although the SDC machine uses a hierarchical architecture (plus some custom hardware like the Omega network and a hardware sorter), the effect of the hardware design is not evaluated in [42].

3. Volcano System Design

In this section, we provide an overview of the modules in Volcano. Volcano's file system is rather conventional. It includes modules for device, buffer, file, and B^+-tree management. For a detailed discussion, we refer to [26].

The file system routines are used by the query processing routines to evaluate complex query plans. Queries are expressed as complex algebra expressions; the operators of this algebra are query processing algorithms. All algebra operators are implemented as *iterators*, i.e., they support a simple *open-next-close* protocol similar to conventional file scans.

Associated with each algorithm is a *state record.* The arguments for the algorithms are kept in the state record. In queries involving more than one operator (i.e., almost all queries), state records are linked together by means of *input* pointers. The input pointers are also kept in the state records. They are pointers to a *QEP* structure which in turn points to the three procedures implementing the operator (*open*, *next*, and *close*) and to a state record. All state information for an iterator is kept in its state record; thus, an algorithm may be used multiple times in a query by including more than one state record in the query. 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.

Figure 4 shows the state record for the *filter* iterator which is a fairly general single-input operator useful for printing, updates, projection, and others purposes. The upper rectangle represents a QEP structure; holding a pointer to it permits calling the iterator functions and passing them the state record address as argument. The lower rectangle represents the state record with sections for arguments, input, and local state. The input arrow points to another QEP structure to allow any of the filter functions to invoke the operator that produces the input stream for the filter operator.

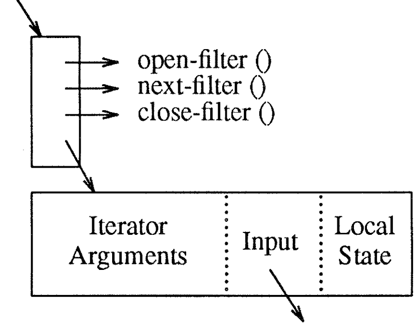

Figure 4. Data Structures for a Filter Operator in a Query Plan.

Calling *open* for the top-most operator results in instantiations for the associated state record, e.g., allocation of a hash table, and in *open* calls for all inputs. In this way, all iterators in a query are initiated recursively. In order to process the query, *next* for the top-most operator is called repeatedly until it fails with an *end-of-stream* indicator. Finally, the *close* call recursively "shuts down" all iterators in the query. This model of query execution matches very closely the one being included in the E programming language design [56] and the algebraic query evaluation system of the Starburst extensible relational database system [36].

The tree-structured query evaluation plan is used to execute queries by demand-driven dataflow. The return value of *next* is, besides a status value, a structure called *NEXT_RECORD* that consists of a record identifier and a record address in the buffer pool. This record is pinned (fixed) in the buffer. The protocol about fixing and unfixing records is as follows. Each record pinned in the buffer is *owned* by exactly one operator at any point in time. After receiving a record, the operator can hold on to it for a while, e.g., in a hash table, unfix it, e.g., when a predicate fails, or pass it on to the next operator. Complex operations like join that create new records have to fix them in the buffer before passing them on, and have to unfix input records.

For intermediate results, Volcano uses *virtual devices*. Pages of such a device exist only in the buffer, and are discarded when unfixed. Using this mechanism allows assigning unique RID's to intermediate result records, and allows managing such records in all operators as if they resided on a real (disk) device. The operators are not affected by the use of virtual devices, and can be programmed as if all input comes from a disk-resident file and output is written to a disk file.

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. 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 [27, 40, 41]. This 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 [29]. The new "parallelism" operator, called *exchange* in Volcano, is the focus of the next section.

4. Implementation of Parallelism in Volcano

In this section, we describe how Volcano's exchange operator works. The other operators, e.g., for scans, sort, join, etc., are described elsewhere [27, 28, 31, 32, 40]. The shared-memory exchange operator is discussed in [29, 32]. In this report, we detail how the exchange operator was ported to distributed-memory and hierarchical architectures.

During the port to distributed and hierarchical memory, all encapsulation properties of the exchange operator were maintained, i.e., it continues to shield the other operators from all parallelism issues, e.g., process management, data transfer, and flow control. The important property of the operator model of parallel query evaluation and of Volcano's exchange operator is that all other operators can be implemented without regard to parallelism or a specific parallel hardware architecture. The "work" operators are designed, implemented, debugged, and tuned in an easy and familiar single-process sequential environment.

Before a query can be started, query-independent server processes are created on all participating nodes and can be allocated as the need arises. We first describe the mechanisms for shared memory, and then proceed to distributed memory.

4.1. Shared Memory

The first function of exchange is to provide *vertical parallelism* or pipelining between processes. The *open* procedure allocates a new process after creating a data structure in shared memory called a *port* for synchronization and data exchange. The exchange operator then takes different paths in the old and new processes.

The old process serves as the *consumer* and the new 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 acknowledgement, and returns.

In the producer process, the exchange operator becomes the *driver* for the query tree below the exchange operator using *open*, *next*, and *close* on its input. The output of *next* is collected in *packets*, which are arrays of *NEXT_RECORD* structures. The packet size is an argument in the exchange iterator's state record, and can be set between 1 and 2^{15} records. When a packet is filled, it is inserted into a linked list anchored in the *port* and a semaphore is used to inform the consumer about the new packet. 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 and passes it to the consumer.

The alert reader has noticed that the exchange module uses a different dataflow paradigm than all other operators. While all other modules are based on demand-driven dataflow (iterators, lazy evaluation), the producer-consumer relationship 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, the producer may pin a significant portion of the buffer, thus impeding overall system performance. If flow control is enabled, after a producer has inserted a new packet into the port, it must request the flow control semaphore. After a consumer has removed a packet from the port, it releases the flow control semaphore. The initial value of the flow control semaphore determines how many packets the producers may get ahead of the consumers.

Notice that flow control and demand-driven dataflow are not the same. One significant difference is that flow control allows some "slack" in the synchronization of producer and consumer and therefore truly overlapped execution, while demand-driven dataflow is a rather rigid structure of request and delivery in which the consumer waits while the producer works on its next output. The second significant difference is that data-driven dataflow is easier to combine efficiently with horizontal parallelism and partitioning.

The second function of the exchange operator is to provide *intra-operator* or *horizontal* parallelism. Intra-operator parallelism requires data partitioning. Partitioning of stored datasets is achieved by using multiple files, preferably on different devices [54, 58]. Partitioning of intermediate results is implemented by including multiple queues in a port. If there are multiple consumer processes, each uses its own input queue. The producers use a support function to decide into which of the queues an output record belongs (or actually, into which of the packets being filled by the producer). Using a support function allows implementing round-robin-, key-range-, or hash-partitioning.

If an operator or an operator subtree is executed in parallel by a *group* of processes, one of them is designated the *master*. When a query tree is *open*ed, only one process is running, which is naturally the master. When a master allocates a new process in a producer-consumer relationship, the new process becomes the master within its group. The first action of the master producer is to determine how many slaves are needed by calling an appropriate support function. If the producer operation is to run in parallel, the master producer allocates the other producer processes.

After all producer processes are allocated, they run without further synchronization among themselves, with two exceptions. First, when accessing a shared data structure, e.g., the port to the consumers or a buffer table, short-term locks must be acquired for the duration of one linked-list insertion. Second, when a producer group is also a consumer group, i.e., there are at least two exchange operators and three process groups involved in a vertical pipeline, the processes that are both consumers and producers synchronize twice. During the (very short) interval between synchronizations, the master of this group creates a port which serves all processes in its group.

When a *close* request is propagated down the tree and reaches the first exchange operator, the master consumer's *close_exchange* procedure informs all producer processes that they are allowed to close down using the semaphore mentioned above in the discussion on vertical parallelism. If the producer processes are also consumers, the master of the process group informs its producers, etc. In this way, all operators are shut down in an orderly fashion, and the entire query evaluation is self-scheduling.

4.2. Distributed Memory

After the shared-memory version of Volcano's exchange operator had proven the validity of the operator model of parallel query execution, a distributed-memory version was the natural extension. In this section, we describe this extension in more detail. 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 a hypercube as well as shared-memory machines and hierarchical machines with shared-memory parallel nodes. Second, encapsulation should be preserved, i.e., work operators should not be concerned with parallelism such that these operators can be designed, implemented, debugged, and tuned without regard to parallelism in any form. Also, no separate scheduler process should be required, or even a scheduler node as in Gamma [13]. Third, the exchange operator should only provide the mechanisms for parallel query execution, leaving policy decisions to the query optimizer. Fourth, data transfer should be as fast as possible, in particular, shared-memory techniques should be used within each node. Message passing should be used only across node boundaries. Fifth, process and communication setup times should be short to allow speedup even for queries on small datasets. Thus, Volcano employs a pool of server processes at each node, also called primed processes [34]. Sixth, the code should be portable to environments other than the prototype development environment of UNIX workstations.

One of the preconditions for distributed-memory query processing is the ability to ship plans and programs across the network. For the query plans, the *open-next-close* triple of functions required for each operator was extended by two more functions called *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 part of the server (or primed) processes, and each *pack* routine must include interpretable code in the network packet.

Figure 5 shows a simple query evaluation plan with three operators called P, T, and S. These operators could be any work operators, e.g., scan, aggregation, print, etc. We restricted ourselves to single-input operators in Figure 5 because a complex tree would have made the example harder to follow. Any query

$$\begin{tikzcd}
\text{Print (P)} \arrow[d, dash] \\
\text{Exchange (X)} \arrow[d, dash] \\
\text{Filter (T)} \arrow[d, dash] \\
\text{Exchange (Y)} \arrow[d, dash] \\
\text{Scan (S)}
\end{tikzcd}$$

Figure 5. Simple Query Plan.

evaluation plan, e.g., the one shows in the introduction, could be executed on a distributed-memory or hierarchical machine.

Figure 6 shows a possible parallelization on four nodes for the plan in the previous figure. Again, as for the shared-memory case, other parallelizations would have been possible with Volcano's mechanisms, but this will serve as our example. The four rectangles represent nodes; they could be autonomous comput-

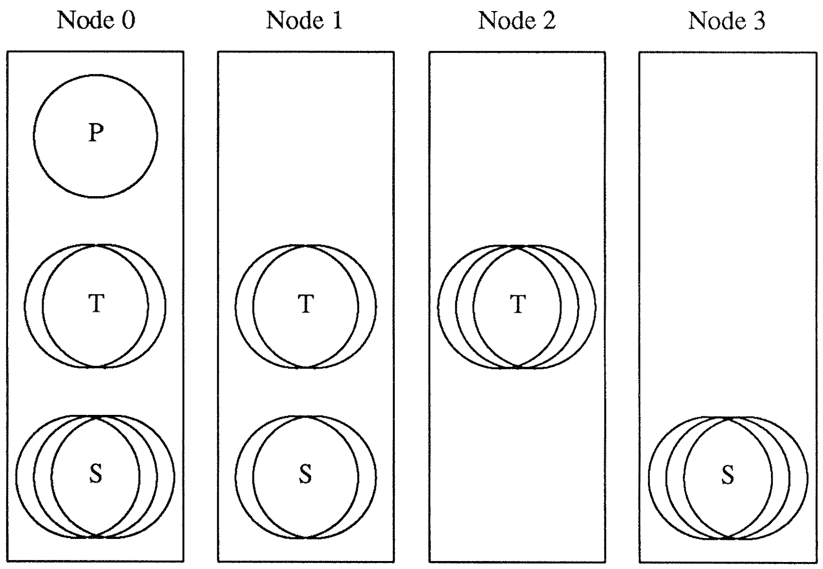

Figure 6. Nodes, Processes, and Operators.

ers 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 left-most and right-most processes in Figure 6 are called $S_{0,0}$ and $S_{3,2}$

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. The processes $O_{0,0}$ are not only the local masters on node 0 but also the global masters that facilitate glocal synchronization where necessary. In a way, they perform, beyond their data manipulation tasks, the role of the scheduler process required for example in Gamma [13, 14, 20].

Figure 7 shows the messages required between nodes to initiate the process structure 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, and would change if a different operating system were available.

First, after $P_{0,0}$ has initiated $T_{0,0}$, $T_{0,0}$ sends a request message to known ports at nodes 1 and 2, indicated 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., $T_{1,1}$.

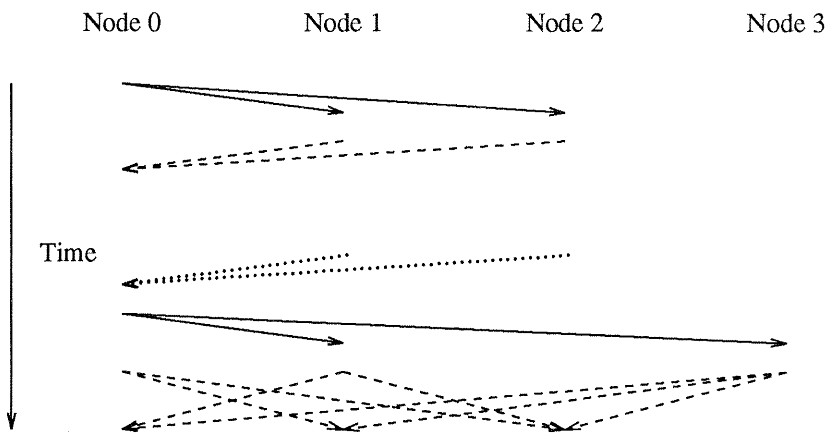

Figure 7. Control Messages.

Second, the producer processes $T_{i,j}$ (actually, this is the X operator in the $T_{i,j}$ processes) 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. At this point, the X operator is ready to transfer data.

Third, when the T operators open their input operator Y, all Y operators in all $T_{i,j}$ processes synchronize to ensure that all processes have reached this point. They first synchronize locally, led by each local master, and then the local masters synchronize with the global master, indicated by dotted arrows.

Fourth, $T_{0,0}$ initiates $S_{0,0}$. $S_{0,0}$ sends request packets to the local masters for the $S_{i,j}$ processes. 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 all required data paths. At this point, setup is completed and all processes are ready to produce and transfer data.

Processes now act 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. Process performance is limited by four factors, namely processor performance, I/O performance, speed of input from other processes, and flow control. Flow control is implemented within each node as described above and in [29, 32]. 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 across node boundaries are organized similarly to those within nodes. There is only one main difference, dictated directly by the accessibility of memory. For data exchange across node boundaries, there are no packets with pointers to records in the shared buffer pool. Instead, each sender maintains a set of virtual files, one for each remote consumer. Each time a packet fills up, it is sent across the network. The receiver also maintains a virtual file into which the network packet is read. Note that the organization of virtual files requires only one copy step beyond the copy implicit in the send and receive network operations, namely to assemble packets for individual consumers.

Shutting all processes down is quite simple. Within each node, the mechanisms described above for shared memory are used. Among nodes, each process flushes all its output data to the appropriate recipients and then exits, i.e., the process returns to the pool of waiting server processes. The waiting and synchronization required for shared memory is not needed for distributed memory because processes do not share records in the buffer pool, so there is no need to delay closing files until all consumers have unpinned their records in the files.

At the current time, the exchange operator is operational in Volcano as described so far. The next step is to include all special modes of operation, e.g., merging multiple sorted streams or replication [29, 32], that currently work only on shared-memory machines. Following that, we plan on investigating suitable mechanisms for error and exception handling in extensible multi-process multi-processor query execution.

In summary, the recent extensions to Volcano's exchange operator provide the mechanisms for parallel query processing in shared-memory, distributed-memory, and hierarchical machines. The modified exchange operator retains all encapsulation properties designed and implemented in the shared-memory version. It effectively shields the work operators from all parallelism issues. Beyond process management, data transfer, and flow control, the new exchange operator also hides the machine architecture from the work 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 more operators on more systems in the near future. The first prototype implementation relies on UNIX and TCP/IP because this was the most convenient development environment for us; we hope to port the system and its new exchange operator to other distributed-memory parallel machines in the near future.

5. Performance Observations

In this section, we report on experimental performance measurements and argue that hierarchical architectures are indeed promising for database query evaluation.

From a number of projects and reports, it is well known that both data and program parallelism (partitioning and pipelining) can provide significant speedup. Frequently, linear speedup could be obtained, e.g. [13, 15].

Figure 8, taken from a tuning study reported in [30], shows performance measurements for sorting 1,000,000 records of 100 bytes each, a standard database benchmark [3], with varying degrees of

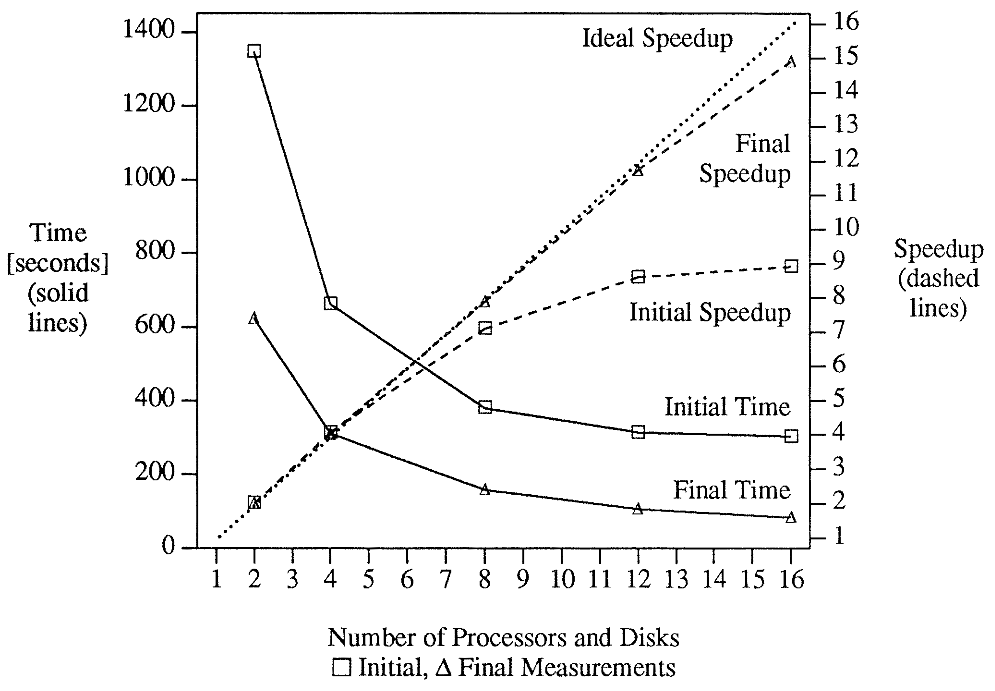

Figure 8. Shared-Memory Sorting 1,000,000 Records of 100 Bytes.

parallelism on a (shared-memory) Sequent Symmetry [47]. The time measurements are shown using solid lines and refer to the labels on the left. The speedups are shown with dashed lines and refer to the labels on the right. The initial times and speedups, i.e., the starting point of the tuning study, are marked with □'s while the final ones are marked with Δ's. The ideal speedup is also shown by the dotted line.

There are three important conclusions to be drawn from Figure 8. First, a comparison of the dashed and dotted lines shows very close to linear speedup with the fully tuned software. Thus, shared-memory machines are perfectly reasonable platforms for parallel database query evaluation. Second, it is immediately obvious that the final times are significantly lower than the initial ones, and that the speedup for the initial software was far from linear. Thus, the tuning improved the parallel behavior as well as the absolute performance. On the other hand, it means that in order to successfully exploit a shared-memory machine, careful tuning might be necessary. Third, the speedup is close to linear only within a certain range; obvious for the initial software but also visible for the tuned software. We suspect that experiments with much higher degrees of parallelism, e.g., 32 or 64 CPUs and disks, would have revealed a limitation in speedup even for the fully tuned software. Thus, shared-memory machines do not scale to very high degrees of parallelism.

Distributed-memory machines have been shown to deliver both good basic performance and linear speedup to high degrees of parallelism, e.g., Bubba [9], Gamma [14], and Tandem [15]. A careful simulation study that was thoroughly verified with a working prototype (for small degrees of parallelism) demonstrated further that such machines can be scaled to very large degrees of parallelism [20, 21]. Thus, very high performance database machines and servers can probably be built based on distributed-memory machines.

For Volcano's exchange operator, we obtained some preliminary performance measurements to show that speedups can be achieved with the operator model on distributed-memory machines. We measured exchanging records between Volcano processes on a network of Sun SparcStations (about 12 MIPS RISC) running SunOS connected via an ethernet. The test query only created and exchanged records, i.e., there was no data manipulation in this experiment. Since we were primarily interested in exchange performance, we eliminated disk I/O by using only virtual devices.

Figure 9 shows the performance of distributed exchange, comparing local data exchange between processes with exchange across a network with two or four machines. Measurements are shown for 0, 1,000, 10,000, and 100,000 records. Measurements for 0 records are included to assess the overhead of allocating and deallocating all processes and data paths. For each data point in the figure, we took five to

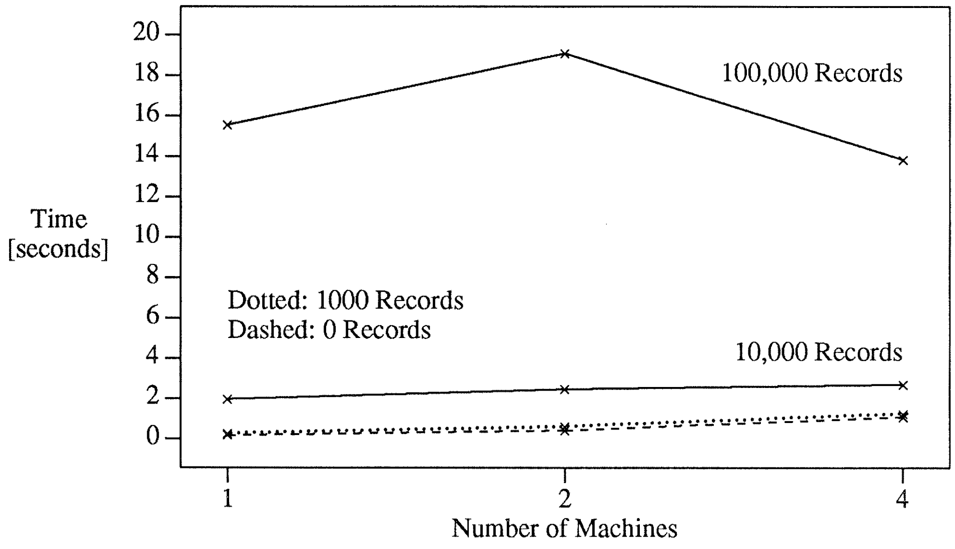

Figure 9. Volcano Distributed-Memory Performance.

ten measurements and reported the mean of all observed times. It is interesting to note that just as for shared memory, obtaining acceptable and reasonably consistent measurements in the distributed-memory environment required a fair amount of tuning [30]. Tuning to-date for distributed memory involved slack in flow control, packet size, and timeouts for various network system calls. We hope that we can obtain better performance on the same hardware and OS platform by February 1991.

There are two interesting observations possible from the curves. First, allocating remote processes is not much slower than local processes. Thus, the attempt to minimize initialization and cleanup overheads was effective, and we can hope that reasonable speedups can be obtained even for relatively small data sets. Nonetheless, the overhead of our operating system still weighs heavily in these measurements. Second, for large amounts of data, performance improves for more nodes, in spite of the fact that for more machines, more records travel across the network. For *N* machines, *1/N* of each process' output remains in the same machine, while the remainder is exchanged across the network (½ of all records in a network of two nodes, ¾ for for nodes). Thus, we can hope that for large set cardinalities, the exchange operator may permit close to linear speedup.

Figure 9 represents only the cost of the exchange operator, without regard to work operators like scan and join. Recall that no data manipulation or physical I/O was performed during these measurments. Thus, the figure shows the performance of distributed exchange in the worst and most critical light. If data sets are partitioned and smaller amounts of data are processed at each network node, the speedup for all work operators is linear or even super-linear if merge levels during sorting or recursion levels during hash overflow resolution can be avoided. Thus, even if the exchange performance is not linearly improving but about constant for multiple nodes, query processing performance still improves.

We realize that the absolute execution times might not be as fast as one could hope for considering the hardware used, but we suspect that the UNIX and TCP/IP communication software (the only effective development platform available to us at the time) introduced significant overhead. Furthermore, we suspect that some network packets might have collided on the ethernet and therefore introduced delays. Wang and Luk observed similar network limitations for parallel join algorithms on multiple workstations, even though they used an operating system with faster communication (V-Kernel) [66]. Our overhead could be reduced on a distributed-memory machine with simplified and streamlined communication, e.g., an iPSC/2 as used in Gamma [14]. Nonetheless, the important point is that Volcano's exchange operator for distributed-memory machines does allow speedups in data transfer in addition to linear speedups in data manipulation. Considering the designs and implementations of distributed-memory database machines (Bubba, Gamma, Tandem) with less communication overhead, there is no reason why Volcano should not deliver similar speedup and scaleup results on a more suitable hardware and operating system platform.

In summary, we have observed near-linear speedup in shared-memory machines, but realize that the useful degree of parallelism based on shared memory is limited. For a distributed-memory ensemble of workstations, we have observed some, although not linear speedup. However, considering the small overhead for establishing and terminating process groups and data paths, we believe that Volcano's exchange operator will permit linear speedup for distributed memory machines if the communication overhead due to operating system and network hardware can be reduced. Nonetheless, the communication cost between nodes will remain higher than communication cost via shared memory. By supporting both shared and distributed memory and their combination, Volcano's exchange operator will be able to exploit both of them and hopefully allow very high query processing performance on forthcoming hierarchical computer architectures.

6. Summary and Conclusions

In this report, we have reported on 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 article, we reported on the shared-memory version [29]; 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 [29], has been maintained during the extension from the shared-memory version to the distributed-memory version of the exchange operator. Therefore, the other 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, or 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 three-way join algorithm.

Volcano's design and implementation also supports a hierarchical combination of the two conventional architectures, i.e., a networked group of shared-memory machines. At least one such closely-tied machine has been built, the Evans & Sutherland ES-1, and at least two currently active computer manufacturers are considering this general hardware architecture. We plan on porting Volcano to these new architectures as soon as such machines become available. Current 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 conventional distributed-memory machine uses nodes with single CPUs. Hence, 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 non-numerical ones alike. However, different applications, even different algorithms, have different communication-to-computation ratios. Therefore, the maximal degree of parallelism to which shared-memory machines can be scaled varies for different applications. Furthermore, as different hardware components (CPUs, busses, memories) become faster at different paces, the computation-to-communication time ratio might change. Thus, it seems essential for successful future architectures to be modular, meaning that CPUs, memories, etc. can be replaced individually and reconnected using busses and network interconnections to optimally exploit the hardware for an application.

For database applications, the connectivity of CPUs and memories to I/O devices is another important consideration. Boral and DeWitt argued that I/O is the most likely bottleneck in database machines [8], and designed two independent prototype database machines in which storage devices are connected closely to the CPUs [9, 14]. Since I/O typically places significant load on a system's bus, the I/O load has to be considered, too, not only the communication-to-computation ratio. For future database machines or database servers, the different I/O requirements of different applications further support the argument that interconnection topologies should be modular, in particular the assignment of processing units and disks to local busses and local busses to network interconnections.

If indeed future parallel architectures use hierarchical designs based on shared-memory parallel nodes connected by high-speed networks, the new version of Volcano and its exchange operator are a powerful experimental vehicle for parallel database query evaluation. The entire Volcano system is designed and implemented to provide mechanisms for experimentation with and exploration of policies [32]; the exchange operator that encapsulates all parallelism issues for shared-memory, distributed-memory, and hierarchical hardware architectures is an excellent example for this design principle.

References

1. W. Alexander and G. Copeland, ''Process and Dataflow Control in Distributed Data-Intensive Systems'', *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 90-98.
2. D. P. Anderson, S. Tzou and G. S. Graham, ''The DASH Virtual Memory System'', Technical Report 88/461, UC Berkeley CS Division, November 1988.
3. Anon. et al., ''A Measure of Transaction Processing Power'', *Datamation*, April 1, 1985, 112-118.
4. C. K. Baru, O. Frieder, D. Kandlur and M. Segal, ''Join on a Cube: Analysis, Simulation, and Implementation'', *Proceedings of the 5th International Workshop on Database Machines*, 1987.
5. M. Beck, D. Bitton and W. K. Wilkinson, ''Sorting Large Files on a Backend Multiprocessor'', *IEEE Transactions on Computers 37* (1988), 769-778.
6. A. Bhide and M. Stonebraker, ''A Performance Comparison of Two Architectures for Fast Transaction Processing'', *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1988, 536-545.
7. A. Bhide, ''An Analysis of Three Transaction Processing Architectures'', *Proceedings of the Conference on Very Large Databases*, Long Beach, CA., August 1988, 339-350.
8. H. Boral and D. J. DeWitt, ''Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines'', *Proceeding of the International Workshop on Database Machines*, Munich, 1983.
9. 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 Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 4-24.
10. D. Burkes and R. Treiber, ''Design Approaches for Real-Time Transaction Processing and Remote Site Recovery'', *Digest of Papers, 35th CompCon Conference*, San Francisco, CA., Feb-Mar 1990.
11. G. Copeland, W. Alexander, E. Boughter and T. Keller, ''Data Placement in Bubba'', *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 99-108.
12. D. J. DeWitt and P. B. Hawthorn, ''A Performance Evaluation of Database Machine Architectures'', *Proceeding of the Conference on Very Large Data Bases*, Cannes, France, September 1981, 199-213.
13. D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar and M. Muralikrishna, ''GAMMA - A High Performance Dataflow Database Machine'', *Proceedings of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 228-237.

14. D. J. DeWitt, S. Ghandeharadizeh, D. Schneider, A. Bricker, H. I. Hsiao and R. Rasmussen, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 44-62.

15. 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 Systems Technical Report 89.4* (May 1989).

16. R. Epstein and M. Stonebraker, "Analysis of Distributed Data Base Processing Strategies", *Proceedings of the Conference on Very Large Data Bases*, Montreal, Canada, October 1980, 92-101.

17. M. T. Fang, R. C. T. Lee and C. C. Chang, "The Idea of Declustering and Its Applications", *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 181-188.

18. O. Frieder, "Multiprocessor Algorithms for Relational-Database Operators on Hypercube Systems", *IEEE Computer 23*, 11 (November 1990), 13.

19. S. Fushimi, M. Kitsuregawa and H. Tanaka, "An Overview of The System Software of A Parallel Relational Database Machine GRACE", *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 209-219.

20. R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms", *PhD. Thesis*, October 1986.

21. R. H. Gerber and D. J. DeWitt, "The Impact of Hardware and Software Alternatives on the Performance of the Gamma Database Machine", *Computer Sciences Technical Report 708* (July 1987), University of Wisconsin — Madison.

22. S. Ghandeharizadeh and D. J. DeWitt, "Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines", *Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, 1990, 481.

23. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. L. Henessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proc. 17th Annual Int'l Symposium on Computer Architecture, ACM SIGARCH Computer Architecture News 18*, 2 (June 1990), 15.

24. J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor", *Computer Sciences Technical Report 766* (April 1988), University of Wisconsin — Madison.

25. J. R. Goodman, M. D. Hill and P. J. Woest, *Scalability and Its Application to Multicube*, University of Wisconsin — Madison.

26. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System", *Oregon Graduate Center, Computer Science Technical Report*, Beaverton, OR., June 1989.

27. G. Graefe, "Relational Division: Four Algorithms and Their Performance", *Proceedings of the IEEE Conference on Data Engineering*, Los Angelos, CA, February 1989, 94-101.

28. G. Graefe and K. Ward, "Dynamic Query Evaluation Plans", *Proceedings of the ACM SIGMOD Conference*, Portland, OR, May-June 1989, 358.

29. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", *Proceedings of the ACM SIGMOD Conference*, Atlantic City, NJ., May 1990, 102.

30. G. Graefe and S. S. Thakkar, "Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor", *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 470*, April 1990.

31. G. Graefe, "Parallel External Sorting in Volcano", *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 459*, February 1990.

32. G. Graefe, "Volcano, An Extensible and Parallel Dataflow Query Processing System", *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 481*, July 1990.

33. G. Graefe, "Parallelizing the Volcano Database Query Processor", *Digest of Papers, 35th CompCon Conference*, San Francisco, CA., Feb-Mar 1990, 490-493.

34. J. N. Gray, "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course*, vol. 60 , Springer, New York, 1978, 393-481.

35. J. Gray, "A Census of Tandem System Availability Between 1985 and 1990", Tandem Computers Technical Report 90.1, Tandem Computers, January 1990.

36. L. M. Haas, W. F. Cody, J. C. Freytag, G. Lapis, B. G. Lindsay, G. M. Lohman, K. Ono and H. Pirahesh, "An Extensible Processor for an Extended Relational Query Language", *Computer Science Research Report*, San Jose, CA., April 1988.

37. H. I. Hsiao and D. J. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines", *Proceedings of the IEEE Conference on Data Engineering*, Los Angelos, CA, February 1990, 456.

38. K. A. Hua and C. Lee, "An Adaptive Data Placement Scheme for Parallel Database Computer Systems", *Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, 1990, 493.

39. B. R. Iyer and D. M. Dias, "System Issues in Parallel Sorting for Database Systems", *Proceedings of the IEEE Conference on Data Engineering*, Los Angelos, CA, February 1990, 246.

40. T. Keller and G. Graefe, "The One-to-One Match Operator of the Volcano Query Processing System", *Oregon Graduate Center, Computer Science Technical Report*, Beaverton, OR., June 1989.

41. T. Keller, G. Graefe and D. Maier, "Efficient Complex Object Assembly in the REVELATION Project", *in preparation*, November 1990.

42. M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Skew in the Super Database Computer (SDC)", *Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, 1990, 210.

43. S. C. Kwan, "External Sorting: I/O Analysis and Parallel Processing Techniques", *Ph.D. Thesis*, January 1986.

44. D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz and M. Lam, "Design of Scalable Shared-Memory Multiprocessors: The DASH Approach", *CompCon Spring 1990*, Stanford, CA, .

45. R. Lorie, J. Daudenarde, G. Hallmark, J. Stamos and H. Young, "Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience", *IEEE Database Engineering 12*, 1 (March 1989), 58-64.

46. R. A. Lorie and H. C. Young, "A low communication sort algorithm for a parallel database machine", *Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989, 125.

47. T. Lovett and S. S. Thakkar, "The Symmetry Multiprocessor System", *Proceedings International Conference on Parallel Processing*, August 1988.

48. J. Lyon, "Tandem's Remote Data Facility", *Digest of Papers, 35th CompCon Conference*, San Francisco, CA., Feb-Mar 1990.

49. J. Menon, "A Study of Sort Algorithms for Multiprocessor Database Machines", *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 197-206.

50. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networkks", *Communications of the ACM 19*, 7 (July 1976), 395-404.

51. K. P. Mikkilineni and S. Y. W. Su, "An Evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment", *IEEE Transactions on Software Engineering 14*, 6 (June 1988), 838.

52. M. Mori, K. Sazuki, H. Abe and K. Itoh, "A Very Large Data Base System to Serve National Welfare", *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 496-501.

53. M. C. Murphy and D. Rotem, "Effective resource utilization for multiprocessor join execution", *Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989,

67.

54. D. A. Patterson, G. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 109-116.

55. H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu and P. Selinger, "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches", *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 1990, 4.

56. J. E. Richardson and M. J. Carey, "Programming Constructs for Database System Implementation in EXODUS", *Proceedings of the ACM SIGMOD Conference*, San Francisco, CA., May 1987, 208-219.

57. J. P. Richardson, H. Lu and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", *Proceedings of the ACM SIGMOD Conference*, San Francisco, CA., May 1987, 399-409.

58. K. Salem and H. Garcia-Molina, "Disk Striping", *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1986, 336.

59. B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren and B. Vaughan, "FastSort: An Distributed Single-Input Single-Output External Sort", *Proceedings of the ACM SIGMOD Conference*, Atlantic City, NJ., May 1990, 94.

60. D. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", *Proceedings of the ACM SIGMOD Conference*, Portland, OR, May-June 1989, 110.

61. J. W. Stamos and H. C. Young, "A Symmetric Fragment and Replicate Algorithm for Distributed Joins", *Technical Report RJ7188* (December 5, 1989), IBM Almaden Research Lab.

62. H. S. Stone, "Parallel Querying of Large Databases: A Case Study", *IEEE Computer 21*, 10 (October 1987), 11-21.

63. M. Stonebraker, "The Case for Shared-Nothing", *IEEE Database Engineering 9*, 1 (1986).

64. M. Stonebraker, R. Katz, D. Patterson and J. Ousterhout, "The Design of XPRS", *Proceedings of the Conference on Very Large Databases*, Long Beach, CA., August 1988, 318-330.

65. M. Stonebraker, P. Aoki and M. Seltzer, "Parallelism in XPRS", *UCB/Electronics Research Lab. Memorandum M89/16*, Berkeley, February 1989.

66. X. Wang and W. S. Luk, "Parallel Join Algorithms on a Network of Workstations", *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Austin, TX., December 1988, 87.

67. E. Wong and R. H. Katz, "Distributing a Database for Parallelism", *Proceedings of the ACM SIGMOD Conference*, San Jose, CA., May 1983, 23-29.

68. H. Zeller, "Parallel Query Execution in NonStop SQL", *Digest of Papers, 35th CompCon Conference*, San Francisco, CA., Feb-Mar 1990, 484-487.