Oregon Health & Science University
OHSU Digital Commons

CSETech

June 1989

Encapsulation of parallelism in the volcano query processing system

Goetz Graefe

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
Graefe, Goetz, "Encapsulation of parallelism in the volcano query processing system" (1989). *CSETech*. 193.
http://digitalcommons.ohsu.edu/csetech/193

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.

Encapsulation of Parallelism in the Volcano Query Processing System

Goetz Graefe

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-007
June, 1989

Encapsulation of Parallelism in the Volcano Query Processing System

Goetz Graefe

Oregon Graduate Center
Beaverton, Oregon 97006-1999
graefe@cse.ogc.edu

Abstract

Volcano is a new dataflow query processing system we have developed for database systems research and education. A uniform interface between operators, e.g., scan, select, or join, makes Volcano extensible by new operators. It includes an *exchange* operator that allows intra-operator parallelism on partitioned datasets and both vertical and horizontal inter-operator parallelism. All other operators are programmed as for single-process execution; the exchange operator encapsulates all parallelism issues, including the translation between demand-driven dataflow within processes and data-driven dataflow between processes, and therefore makes implementation of parallel database algorithms significantly easier and more robust.

1. Introduction

In order to provide a testbed for database systems education and research, we decided to implement a high-performance query processing system. Since we have only limited resources, we spent a fair amount of time thinking about how we can make our software very flexible without sacrificing efficiency. The result is a small system, consisting of less than two dozen core modules with a total of 10,000 lines of C code. These modules includes a file system, buffer management, sorting, B^+-trees, and two algorithms each for natural join, semi-join, outer join, anti-join, aggregation, duplicate elimination, division, union, intersection, difference, anti-difference, and Cartesian product. Moreover, a single module allows parallel processing of all algorithms listed above.

In the following section, we briefly review previous work that influenced our design. In Section 3, we provide a more detailed description of Volcano. Parallel processing is implemented in the *exchange* module described in Section 4. We present experimental performance measurements in Section 5 that show the *exchange* operator's low overhead. Section 6 contains a summary and our conclusions from this effort.

The art work on the cover was done by Kelly Atkinson from a photograph of Mt. Hood, Oregon.

2. Previous Work

Since so many different system have been developed to process large dataset efficiently, we only survey the systems that have strongly influenced the design of Volcano. We admit that the system grew in pieces, with first ideas developed at the University of Wisconsin without a clear design for Volcano. In particular, the ideas for dynamic query evaluation plans and for parallel execution were developed at the Oregon Graduate Center [1, 2].

At the start in the early summer of 1987, there was only a feeling that some decisions in WiSS [3] and GAMMA [4] were not optimal for performance, generality, or both. For instance, the decisions to protect WiSS's buffer space by copying a data record in or out for each request and to re-request a buffer page for every record during a scan seemed to inflict too much overhead[1]. Nevertheless, there can be no doubt about the fact that Volcano probably would not have been conceived without these two systems.

During the design of the EXODUS storage manager [5], many of these issues were revisited. Lessons learned and tradeoffs explored in these discussions certainly helped form the ideas behind Volcano. The development of E [6] influenced the strong emphasis on iterators for query processing.

Finally, a number of conventional (relational) and extensible systems have influenced our design. Without further discussion, we mention Ingres [7], System R [8], GENESIS [9], Starburst [10], Postgres [11], and XPRS [12]. Furthermore, there has been a large amount of research and development in the database machine area, such that there is an annual international workshop on the topic. Almost all database machine proposals and implementations utilize parallelism in some form. We certainly have learned from this work, in particular from GAMMA, and tried to include these lessons in the design and implementation of Volcano. In particular, we have strived for simplicity and symmetry in the design, mechanisms for well-balanced parallelism, and efficiency in all details.

---

[1] This statement only pertains to the original version of WiSS as described in [3]. Both decisions were reconsidered for the version of WiSS used in GAMMA.

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 a modules for device, buffer, file, and B^+-tree management. For a detailed discussion, we refer the reader to [13].

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. All functions on data records, e.g., comparisons and hashing, called *support functions* in Volcano, are compiled prior to execution and passed to the processing algorithms by means of pointers to the function entry points. Each of these functions uses an argument allowing interpreted or compiled query evaluation.

Arguments to support functions can be used in two ways. In compiled scans (or other operations), i.e., when the predicate evaluation function is available in machine code, it can be used to pass a constant or a pointer to several constants to the predicate function. For example, while the predicate consists of comparing a record field with a string, the comparison function is passed as predicate function while the search string is passed as predicate argument. In interpreted scans, i.e., when a general interpreter is used to evaluate all predicates in a query, it can be used to pass appropriate code for interpretation to the interpreter. The interpreter is given as predicate function. Thus, both interpreted and compiled query evaluation are supported with a single simple and efficient mechanism.

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 includes four pointers to the entry points of the three procedures implementing the operator (*open*, *next*, and *close*) and 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. We call this concept *anonymous inputs* or *streams*. Streams are a simple but powerful abstraction that allows combining any number of operators to evaluate a complex query. Together with the iterator control paradigm, streams represent the most efficient execution model in terms of time (overhead for synchronizing operators) and space (number of records that must reside in memory concurrently) for single process query evaluation.

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 [6] and the algebraic query evaluation system of the Starburst extensible relational database system [14].

The tree-structured query evaluation plan is used to execute queries by demand-driven dataflow. The return value of *next* is, besides an 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.

All operations on records, e.g., comparisons and hashing, are performed by *support functions* which are given in the state records as arguments to the iterators. Thus, the query processing modules could be implemented without knowledge or constraint on the internal structure of data objects.

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.

4. Multi-Processor Query Evaluation

The multi-processor implementation grew out of a desire to leverage as much of the effort as possible when the Oregon Graduate Center acquired an eight-processor shared-memory computer system. We decided that it would be desirable to use the query processing code described above ***without any change.*** The result is very clean, self-scheduling parallel processing.

The module responsible for parallel execution and synchronization is the *exchange* iterator. 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.

This section describes vertical and horizontal parallelism followed by an example, a discussion of variations and variants of the *exchange* operator, an overview of modifications to the file system required for parallel processing, and a comparison of Volcano's exchange operator with GAMMA's mechanisms for parallelism.

4.1. Vertical Parallelism

The first function of exchange is to provide *vertical parallelism* or pipelining between processes. The *open* procedure creates a new process after creating a data structure in shared memory called a *port* for synchronization and data exchange. The child process, created using the UNIX *fork* system call, is an exact duplicate of the parent process. The exchange operator then takes different paths in the parent and child processes.

The parent process serves as the *consumer* and the child 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. After creating the child 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.

The exchange operator in the producer process 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*, data structures of 1 KB which contain 83 *NEXT_RECORD* structures. When a packet is filled, it is inserted into the *port* and a

semaphore is used to inform the consumer about the new packet[2]. 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, passes it to the consumer, and waits until the consumer allows closing all open files. This delay is necessary because files on virtual devices must not be closed before all its records are unpinned in the buffer.

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). There are two very simple reasons for this change in paradigms. First, we intend to use the exchange operator also for *horizontal parallelism*, to be described below. Second, this scheme removes the need for request messages. Even though a scheme with request messages, e.g., using a semaphore, would probably perform acceptably on a shared-memory machine, we felt that it creates unnecessary control overhead and delays. Since we believe that very high degrees of parallelism and true high-performance query evaluation requires a closely tied network, e.g., a hypercube, of shared-memory machines, we decided to use a paradigm for data exchange that has has been proven to perform well in a shared-nothing database machine [4].

A run-time switch of exchange enables *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, e.g., 4, 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

[2] 83 records is the standard packet size. The actual packet size is an argument in the state record, and can be set between 1 and 255 records.

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.

4.2. Horizontal Parallelism

There are two forms of horizontal parallelism which we call ***bushy parallelism*** and ***intra-operator*** parallelism. In bushy parallelism, different CPU's execute different subtrees of a complex query tree. Bushy parallelism and vertical parallelism are forms of *inter-operator* parallelism. Intra-operator parallelism means that several CPU's perform the same operator on different subsets of a stored dataset or an intermediate result[3].

Bushy parallelism can easily be implemented by inserting one or two exchange operators into a query tree. For example, in order to sort two inputs into a merge-join in parallel, the first or both inputs are separated from the merge-join by an exchange operation. The parent process turns to the second sort immediately after forking the child process that will produce the first input in sorted order. Thus, the two sort operations are working in parallel.

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. 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 (or actually, into which of the packets being filled by the producer) an output record must go. 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 forks a child process in a producer-consumer relationship, the child process becomes the master

[3] A fourth form of parallelism is inter-query parallelism, i.e., the ability of a database management system to work on several queries concurrently. In the current version, Volcano does not support inter-query parallelism. A fifth and sixth form of parallelism that can be used for database operations involve hardware vector processing [15] and pipelining in the instruction execution. Since Volcano is a software architecture and following the analysis in [16], we do not consider hardware parallelism further.

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 forks the other producer processes.

Gerber pointed out that such a centralized scheme is suboptimal for high degrees of parallelism [17]. When we changed our initial implementation from forking all producer processes by the master to using a *propagation tree* scheme, we observed significant performance improvements. In such a scheme, the master forks one slave, then both fork a new slave each, then all four fork a new slave each, etc. This scheme has been used very effectively for broadcast communication and synchronization in binary hypercubes.

Even after optimizing the forking scheme, its overhead is not negligible. We are considering using *primed processes*, i.e., processes that are always present and wait for work packets. Primed processes are used in GAMMA [4] and in many commercial database systems. Since the distribution of compiled code for support functions is not trivial in our environment (Sequent Dynix), we delayed this change and plan on using primed processes only when we move to an environment with multiple shared-memory machines[4].

After all producer processes are forked, they run without further synchronization among themselves, with two exceptions. First, when accessing a shared data structure, e.g., the port to the consumers, short-term locks must be acquired for the duration of one linked-list insertion. Concurrent invocation of routines of the file system, in particular the buffer manager, is described later in this section. 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

[4] In fact, this work is currently under way.

an orderly fashion, and the entire query evaluation is self-scheduling.

4.3. An Example

Let us consider an example. Assume a query with four operators, A, B, C, and D such that A calls B's, B calls C's, and C calls D's *open*, *close*, and *next* procedures. Now assume that this query plan is to be run in three process groups, called A, BC, and D. This requires an exchange operator between operators A and B, say X, and one between C and D, say Y. B and C continue to pass records via a simple procedure call to the C's *next* procedure without crossing process boundaries. Assume further that A runs as a single process, A_0, while BC and D run in parallel in processes BC_0 to BC_2 and D_0 to D_3, for a total of eight processes.

A calls X's *open*, *close*, and *next* procedures instead of B's (Figure 1a), without knowledge that a process boundary will be crossed, a consequence of anonymous inputs in Volcano. When X is *opened*, it creates a port

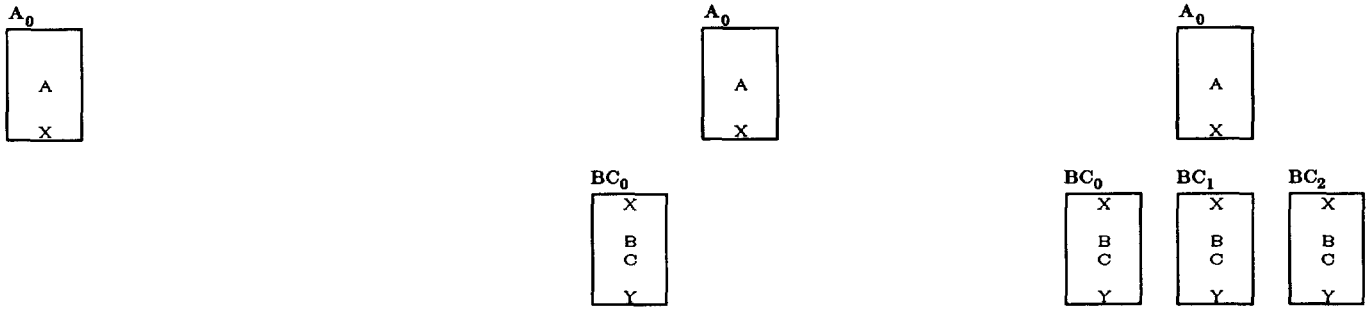

Figure 1a-c. Creating the BC processes.

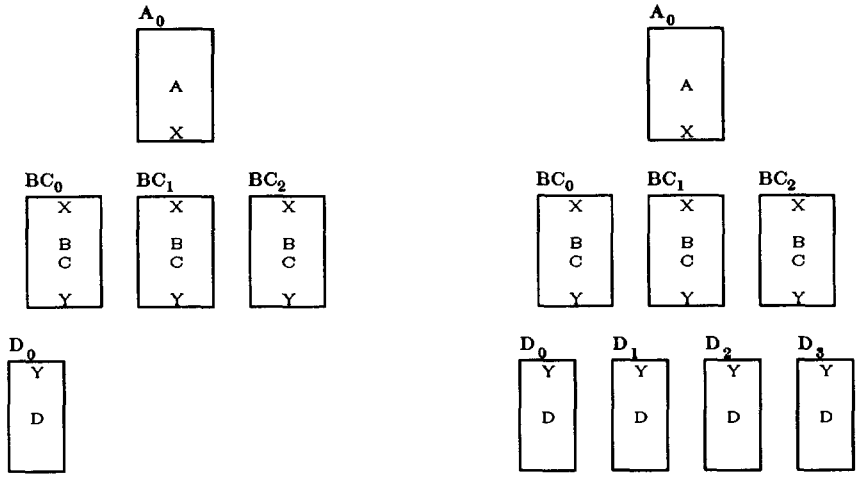

Figure 1d-e. Creating the D processes.

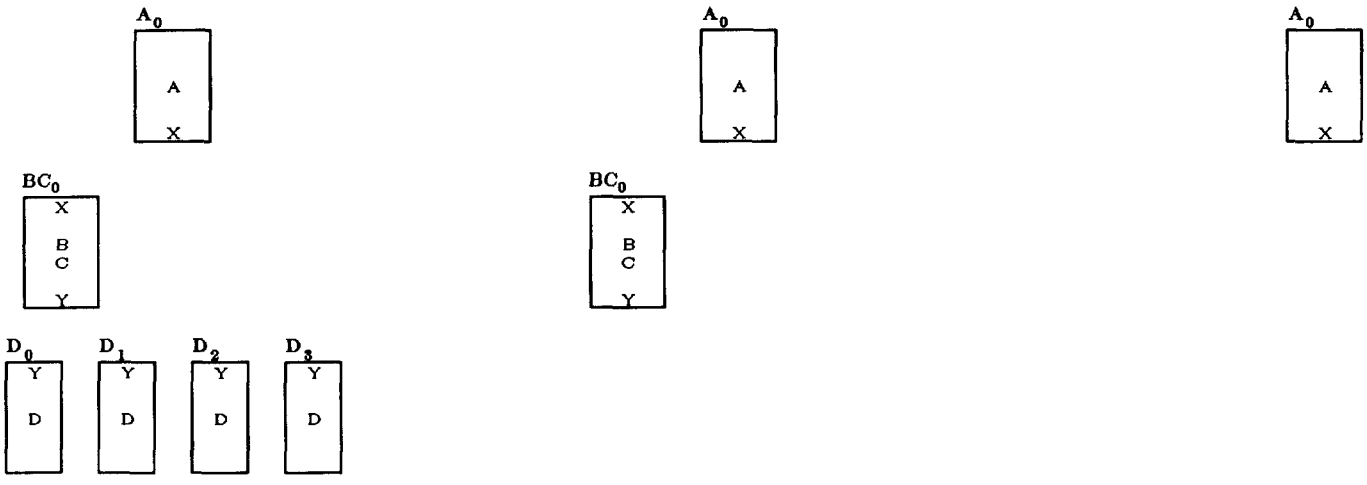

Figure 1f-h. Closing all processes down.

with one input queue for A_0 and forks BC_0 (Figure 1b), which in turn forks BC_1 and BC_2 (Figure 1c). When the BC group *opens* Y, BC_0 to BC_2 synchronize, and wait until the Y operator in process BC_0 has initialized a port with three input queues. BC_0 creates the port and stores its location at an address known only to the BC processes. Then BC_0 to BC_2 synchronize again, and BC_1 and BC_2 get the port information from its location. Next, BC_0 forks D_0 (Figure 1d) which in turn forks D_1 to D_3 (Figure 1e).

When the D operators have exhausted their inputs in D_0 to D_3, they return an *end-of-stream* indicator to the driver parts of Y. In each D process, Y flags its last packets to each of the BC processes (i.e., a total of 3×4=12 flagged packets) with an *end-of-stream* tag and then waits on a semaphore for permission to *close*. The copies of the Y operator in the BC processes count the number of tagged packets; after four tags (the number of producers or D processes), they have exhausted their inputs, and a call to Y's *next* procedure will return an *end-of-stream* indicator. In effect, the *end-or-stream* indicator has been propagated from the D operators to the C operators. In due turn, C, B, and then the driver part of X will receive an *end-of-stream* indicator. After receiving three tagged packets, X's *next* procedure in A_0 will indicate *end-of-stream* to A.

When *end-of-stream* reaches the root operator of the query, A, the query tree is *closed*. Closing the exchange operator X includes releasing the semaphore that allows the BC processes to shut down (Figure 1f). The X driver in each BC process *closes* its input, operator B. B *closes* C, and C *closes* Y. Closing Y in BC_1 and BC_2 is an empty operation. When the process BC_0 *closes* the exchange operator Y, Y permits the D

processes to shut down by releasing a semaphore. After the processes of the D group have closed all files and deallocated all temporary data structures, e.g., hash tables, they indicate the fact to Y in BC_0 using another semaphore, and Y's *close* procedure returns to its caller, C's *close* procedure, while the D processes terminate (Figure 1g). When all BC processes have *close*d down, X's *close* procedure indicates the fact to A_0 and query evaluation terminates (Figure 1h).

4.4. Variants of the Exchange Operator

For some operations, it is desirable to *replicate* or *broadcast* a stream to *all* consumers. For example, one of the two partitioning methods for hash-division [18] requires that the divisor be replicated and used with each partition of the dividend. Another example is Baru's parallel join algorithm in which one of the two input relations is not moved at all while the other relation is sent through all processors [19]. To support these algorithms, the exchange operator can be directed (by setting a switch in the state record) to send all records to all consumers, after pinning them appropriately multiple times in the buffer pool. Notice that it is not necessary to copy the records since they reside in a shared buffer pool; it is sufficient to pin them such that each consumer can unpin them as if it were the only process using them. After we implemented this feature, parallelizing our hash-division programs using both divisor partitioning and quotient partitioning [18] took only about three hours and yielded not insignificant speedups.

When we implemented and benchmarked parallel sorting [20], we added two more features to *exchange*. First, we wanted to implement a merge network in which some processors produce sorted streams merge concurrently by other processors. Volcano's *sort* iterator can be used to generate a sorted stream. A *merge* iterator was easily derived from the sort module. It uses a single level merge, instead of the cascaded merge of runs used in sort. The input of a *merge* iterator is an *exchange*. Differently from other operators, the merge iterator requires to distinguish the input records by their producer. As an example, for a join operation it does not matter where the input records were created, and all inputs can be accumulated in a single input stream. For a merge operation, it is crucial to distinguish the input records by their producer in order to merge multiple sorted streams correctly.

We modified the *exchange* module such that it can keep the input records separated according to their producers, switched by setting an argument field in the state record. A third argument to *next_exchange* is used to communicate the required producer from the *merge* to the *exchange* iterator. Further modifications included increasing the number of input buffers used by *exchange*, the number of semaphores (including for flow control) used between producer and consumer part of *exchange*, and the logic for *end-of-stream*.

Second, we implemented a sort algorithm that sorts data randomly partitioned over multiple disks into a range-partitioned file with sorted partitions, i.e., a sorted file distributed over multiple disks. Using the same number of processors and disks, we used two processes per CPU, one to perform the file scan and partition the records and another one to sort them. We realized that creating more processes than processors inflicted a significant cost, since these processes competed for the CPU's and therefore required operating system scheduling. While the scheduling overhead may not be too significant, in our environment with a central run queue processes can migrate. Considering that there is a large cache associated with each CPU, the cache migration adds a significant cost.

In order to make better use of the available processing power, we decided to reduce the number of processes by half, effectively moving to one process per disk. This required modifications to the exchange operator. Until then, the exchange operator could "live" only at the top or the bottom of the operator tree in a process. Since the modification, the exchange operator can also be in the middle of a process' operator tree. When the exchange operator is *open*ed, it does not fork any processes but establishes a communication port for data exchange. The *next* operation requests records from its input tree, possibly sending them off to other processes in the group, until a record for its own partition is found.

This mode of operation[5] also makes flow control obsolete. A process runs a producer (and produces input for the other processes) only if it does not have input for the consumer. Therefore, if the producers are in danger of overrunning the consumers, none of the producer operators gets scheduled, and the consumers consume the available records.

[5] Whether exchange forks new producer processes (the original exchange design describe in Section 4) or uses the existing process group to execute the producer operations is a run-time switch.

4.5. File System Modifications

Clearly, the file system required some modifications to serve several processes concurrently. In order to restrict the extent of such modifications, Volcano currently does not include protection of files and records other than the volume table of contents (VTOC). Furthermore, typically non-repetitive actions like mounting a device must be invoked by the query root process before or after a query is evaluated by multiple processes. The following few paragraphs list the changes that were required in the file system to allow parallel execution.

The *memory* module allocates space in a shared segment rather than a private segment, thus buffer space is also shared among all processes. In order to protect the memory allocation map, a single exclusive lock is held during the short periods of time while the allocation map is searched or updated.

The *physical I/O* module uses two exclusive locks per device. First, *device busy* lock is held while calling UNIX's *lseek*, *read*, and *write* system calls. This is necessary because otherwise two processes could get into a race-condition in which one process's seek operation determines the location of the other process's write. Second, the *map busy* lock protects the free space bit map.

Changes to the *device* module were restricted to protecting the volume table of contents. An exclusive lock is held while an entry is inserted or deleted or while the VTOC is scanned for the descriptor for an external file.

The most difficult changes were required for the *buffer* module. While we could have used one exclusive lock as in the *memory* module, decreased concurrency would have removed most or all advantages of parallel query processing. Therefore, the buffer uses a two-level scheme. There is a lock for each buffer pool and one for each descriptor (cluster in the buffer). The buffer pool lock must be held while searching or updating the hash tables and bucket chains. It is never held while doing I/O; thus, it is never held for a long period of time. A descriptors or cluster lock must be held while updating a descriptor in the buffer, e.g., to decrease its fix count, or while doing I/O.

Other buffer managers do not use a pool lock but lock each search bucket and the free chain individually, e.g., the buffer manager of Starburst [21]. The advantage is increased concurrency, while the disadvantage is increased number of locks and lock operations. We are currently working on quantifying this tradeoff for our

environment.

If a process finds a requested cluster in the buffer, it uses an atomic test-and-lock operation to lock the descriptor. If this operation fails, the pool lock is released, the operation delayed and restarted. It is necessary to restart the buffer operation including the hash table lookup because the process which holds the lock might be reading or replacing the requested cluster. Therefore, the requesting process must wait to determine the outcome of the prior operation.

Using this restart-scheme for descriptor locks has the additional benefit of avoiding deadlocks. The four conditions for deadlock are *mutual exclusion, hold-and-wait, no preemption,* and *circular wait* [22, 23]. Volcano's restart-scheme does not satisfy the second condition.

While the locking scheme avoids deadlocks, it does not avoid convoys [24]. If a process exhausts its CPU time-slice while holding a "popular" exclusive lock, e.g., on a buffer pool, probably all other processes will block in a convoy until the lock-holding process is re-scheduled and releases the lock. However, since we do not use a "fair" scheduling policy that does not allow reacquiring a lock before all waiting processes held and released the lock, we expect that convoys will quickly evaporate [24]. We intend to investigate the special problem of convoys in shared-memory multi-processors further.

It is interesting to note that spin-locks are quite effective in a multi-processor environment. For instance, the pool is locked typically for about 100 instructions. If a process finds the pool locked, it is cheaper to waste 100 instructions spinning than it is to reschedule the CPU and to perform a context switch.

After the buffer manager and the other file system modules were modified to serve multiple processes, it was straightforward to include a *read-ahead/write-behind daemon.* One or more copies of this daemon process are forked when the buffer manager is initialized, and accept work requests on a queue and semaphore similar to the one used within the exchange module. There are three kinds of work requests, the first two are accompanied by a cluster identifier. First, *FLUSH* writes a cluster if it is in the buffer and dirty. Second, *READ_AHEAD* reads a cluster and inserts it at the top of the *LRU* chain. The cluster remains in the buffer using the normal aging process. If it is not fixed and removed from the free list before it reaches the bottom of the free list, it is replaced. Third, a *QUIT* request terminates the daemon.

4.6. Review and Comparison with GAMMA

In summary, the exchange module encapsulates parallel processing in Volcano. Only very few changes had to be made to the buffer manager and the other modules of the file system in order to accommodate parallel execution. The most important properties of the exchange module are that it implements three forms of parallel processing within a single module, that it makes parallel query processing entirely self-scheduling, and that it did not require any changes in the existing query processing modules, thus leveraging significantly the time and effort spent on them and allowing easy parallel implementation of new algorithms.

It might be interesting to compare Volcano and GAMMA [4] query processing in some detail. We only want to point out differences; we do not claim that the design decisions in Volcano are superior to those in GAMMA. First, Volcano runs on a shared-memory multi-processor, whereas GAMMA runs on a shared-nothing architecture. This difference made Volcano easier to implement but will prevent very large configurations due to bus contention. We are currently investigating where the limit is for our software and hardware architecture, and how we can push it as far as possible. Second, GAMMA is a complete system, with query language, system catalogs, query optimization, concurrent transactions, etc., whereas Volcano in its current form only provides mechanisms for single-user query evaluation. Third, Volcano schedules complex queries without the help of a scheduler process. Operators are scheduled and activated top-down using a tree of iterators. In GAMMA, on the other hand, operators are activated bottom-up by a scheduler process associated with the query. Fourth, GAMMA uses only left-deep query trees, i.e., the probing relation in a hash join [25, 26] must be a stored relation or the result of a selection. In Volcano, both join inputs can be intermediate results. In fact, since Volcano uses anonymous inputs, a join operator has no way of knowing how the inputs were generated. Clearly, the decision whether to use bushy query trees or only left-deep trees has to be made very carefully since the composite resource consumption may lead to thrashing. Fifth, Volcano can execute two or more operators within the same process. In other words, vertical parallelism is optional. In the GAMMA design it is assumed that data have to be repartitioned between operators.

5. Performance and Overhead

From the beginning of the Volcano project, we were very concerned about high performance and low overhead. In this section, we report on experimental measurements of the overhead induced by the exchange operator. We measured elapsed times of the program that creates records, fills them with 4 integers, passes the records over three process boundaries, and then unfixes the records in the buffer. The measurement represent elapsed times on a Sequent Symmetry with twelve Intel 16 MHz 80386 CPU's with 64 KB cache each. Each CPU delivers about 4 MIPS in this machine. The times were measured using the hardware microsecond clock available on such machines.

First, we measured the program without any exchange operator. Creating 100,000 records and releasing them in the buffer took 20.28 seconds. Next, we measured the program with the exchange operator switched to the mode in which it does not create new processes. In other words, compared to the last experiment, we added the overhead of three procedure calls for each record. For this run, we measured 28.00 seconds. Thus, the three exchange operators in this mode added (28.00sec − 20.28sec) / 3 / 100,000 = 25.73μsec overhead per record and exchange operator.

When we switched the exchange operator to create new processes, thus creating a pipeline of four processes, we observed an elapsed time of 16.21 seconds with flow control enabled, or 16.16 seconds with flow control disabled. The fact that these times ars less than the time for single-process program execution indicates that data transfer using the exchange operator is very fast, and that pipelined multi-process execution is warranted.

We were particularly concerned about the granularity of data exchange between processes and its impact on Volcano's performance. We reran the program multiple times varying the number of records per exchange packet. Figure 2a shows the performance for transferring 100,000 records from a producer process group through two intermediate process groups to a single consumer process. Each of these groups included three processes. Each of the producer processes created 33,333 records. All these experiments were conducted with flow control enabled with three "slack" packets per exchange. The bottom axis shows the number of records per packet, increasing from 1 to the default size of 83. The vertical axis shows the elapsed time in seconds.

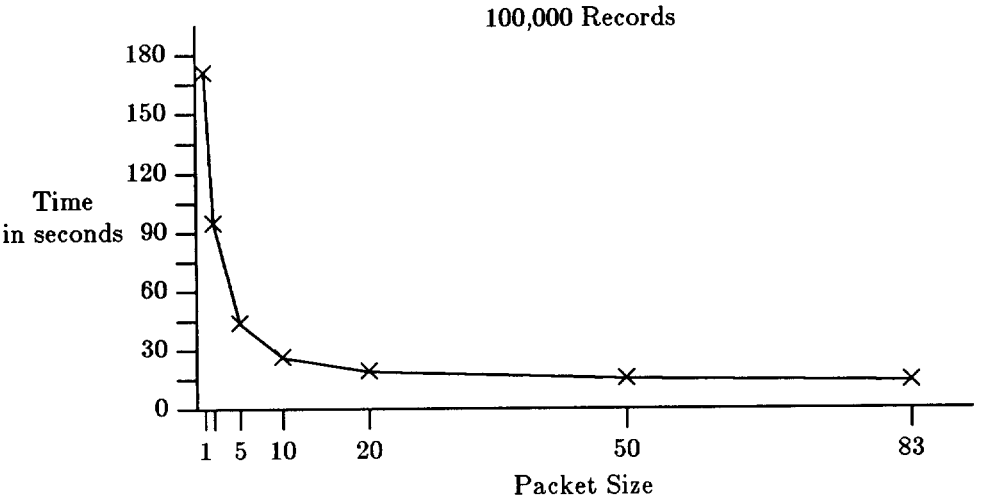

Figure 2a. Exchange Performance.

As can be seen in Figure 2a, the performance penalty for very small packets was significant. The elapsed time was almost cut in half when the packet size was increased from 1 to 2 records, from 171 seconds to 94 seconds. As the packet size was increased further, the elapsed time shrank accordingly, to 15.0 seconds for 50 records per packet and 13.7 seconds for 83 records per packet.

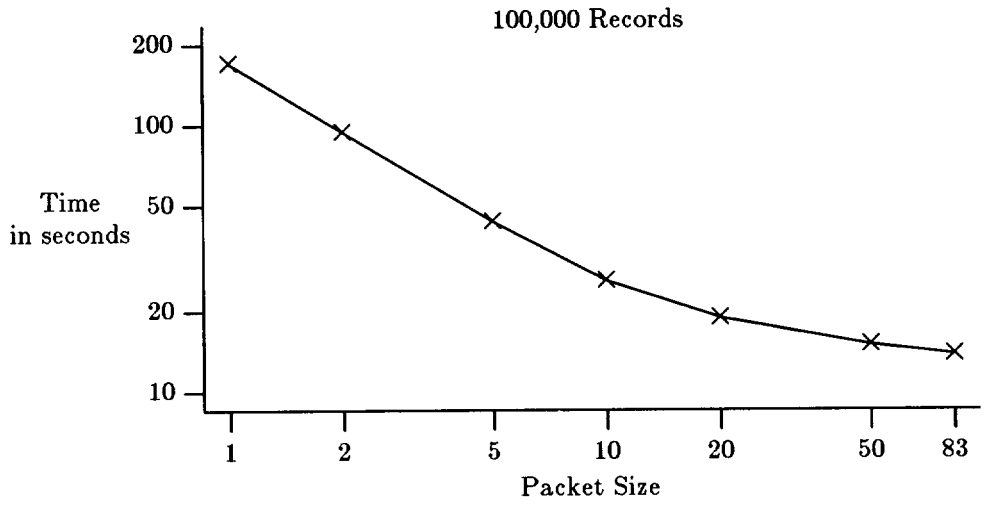

Figure 2b. Exchange Performance (doubly logarithmic scale).

It seems reasonable to speculate that for small packets, most of the elapsed time is spent on data exchange. To verify this hypothesis, we plotted the same data on a doubly logarithmic scale. If the hypothesis is correct, we should see a straight line in this scale for our data. In Figure 2b, we find the hypothesis affirmed for truly small packets, less than 10 records per packet. For larger packets, we notice that the packet size had less impact on the performance. The reason is that the cost of processing the records had become significant, even though hardly any processing was done. In fact, for packets of the standard size, the performance is limited by the consumer process which must invoke the buffer manager once for each record to unfix the record in the buffer.

6. Summary and Conclusions

We have described Volcano, a new query evaluation system, and how parallel query evaluation is encapsulated in a single module or operator. The system is operational on both single- and multi-processor systems, and has been used for a number in database query processing studies [1, 18, 20, 27, 28].

Volcano utilizes dataflow techniques within processes as well as between processes. Within a process, demand-driven dataflow is implemented by means of iterators. Between processes, data-driven dataflow is used to exchange data between producers and consumers efficiently. If necessary, Volcano's data-driven dataflow can be augmented with flow control or back pressure. Horizontal partitioning is used both on stored and intermediate datasets to allow intra-operator parallelism. The design of the exchange operator embodies the parallel execution mechanism for vertical, bushy, and intra-operator parallelism, and it performs the transitions from demand-driven to data-driven dataflow and back.

Volcano is the first implemented query evaluation system that combines extensibility and parallelism. We believe that in Volcano we have a powerful tool for database systems research and education. The encapsulation of parallelism in Volcano allows for new query processing algorithms to be coded for single-process execution but run in a highly parallel environment without modifications. We expect that this will speed parallel algorithm development and evaluation significantly.

References

1. G. Graefe and K. Ward, "Dynamic Query Evaluation Plans," *Proceedings of the ACM SIGMOD Conference*, p. 358 (May-June 1989).

2. G. Graefe, "DataCube: An Integrated Data and Compute Server Based on a Cube-Connected Dataflow Database Machine," *Oregon Graduate Center, Computer Science Technical Report,* (88-024)(July 1988).

3. H.T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug, "Design and Implementation of the Wisconsin Storage System," *Software - Practice and Experience* **15**(10) pp. 943-962 (October 1985).

4. 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,* pp. 228-237 (August 1986).

5. M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita, "Object and File Management in the EXODUS Extensible Database System," *Proceedings of the Conference on Very Large Data Bases,* pp. 91-100 (August 1986).

6. J.E. Richardson and M.J. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the ACM SIGMOD Conference,* pp. 208-219 (May 1987).

7. M. Stonebraker, E. Wong, P. Kreps, and G.D. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems* 1(3) pp. 189-222 (September 1976).

8. M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson, "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems* 1(2) pp. 97-137 (June 1976).

9. D.S. Batory, "GENESIS: A Project to Develop an Extensible Database Management System," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems,* pp. 207-208 (September 1986).

10. P. Schwarz, W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst Database System," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems,* pp. 85-92 (September 1986).

11. M. Stonebraker and L.A. Rowe, "The Design of POSTGRES," *Proceedings of the ACM SIGMOD Conference,* pp. 340-355 (May 1986).

12. M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," *Proceedings of the Conference on Very Large Databases,* pp. 318-330 (August 1988).

13. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System," *Oregon Graduate Center, Computer Science Technical Report,* (89-006)(June 1989).

14. 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,* (RJ 6182 (60892))IBM Almaden Research Center, (April 1988).

15. S. Torii, K. Kojima, Y. Kanada, A. Sakata, S. Yoshizumi, and M. Takahashi, "Accelerating Nonnumerical Processing by an Extended Vector Processor," *Proceedings of the IEEE Conference on Data Engineering,* pp. 194-201 (February 1988).

16. 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,* Springer, (1983).

17. R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," *Ph.D. Thesis,* University of Wisconsin, (October 1986).

18. G. Graefe, "Relational Division: Four Algorithms and Their Performance," *Proceedings of the IEEE Conference on Data Engineering,* pp. 94-101 (February 1989).

19. 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).

20. G. Graefe, "Parallel External Sorting in Volcano," *Oregon Graduate Center, Computer Science Technical Report,* (89-008)(June 1989).

21. B. Lindsay, *Personal Communication*. February 1989.

22. E.G. Coffman, Jr., M.J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys* **3**(2) pp. 67-78 (June 1971).

23. R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys* **4**(3) pp. 179-196 (September 1972).

24. M. Blasgen, J.N. Gray, M. Mitoma, and T. Price, "The Convoy Phenomenon," *Operating Systems Review* **13**(2) pp. 20-25 (April 1979).

25. D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD Conference*, pp. 1-8 (June 1984).

26. D.J. DeWitt and R.H. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the Conference on Very Large Data Bases*, pp. 151-164 (August 1985).

27. G. Graefe, F. Symonds, and G. Kelley, "Shared-Memory Dataflow Query Processing in Volcano," *Oregon Graduate Center, Computer Science Technical Report*, (89-010)(June 1989).

28. T. Keller and G. Graefe, "The One-to-One Match Operator of the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-009)(June 1989).