Encapsulation of Parallelism in the Volcano Query Processing System

Goetz Graefe

CU-CS-458-90 March 1990

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Encapsulation of Parallelism in the Volcano Query Processing System

Goetz Graefe

University of Colorado
Boulder, CO 80309-0430
graefe@boulder.colorado.edu

Abstract

Volcano is a new dataflow query processing system we have developed for database systems research and education. The uniform interface between operators makes Volcano extensible by new operators. All operators are designed and coded as if they were meant for a single-process system only. When attempting to parallelize Volcano, we had to choose between two models of parallelization, called here the *bracket* and *operator* models. We describe the reasons for not choosing the bracket model, introduce the novel operator model, and provide details of Volcano's *exchange* operator that parallelizes all other operators. It allows intra-operator parallelism on partitioned datasets and both vertical and horizontal inter-operator parallelism. The exchange operator encapsulates all parallelism issues and therefore makes implementation of parallel database algorithms significantly easier and more robust. Included in this encapsulation is the translation between demand-driven dataflow within processes and data-driven dataflow between processes. Since the interface between Volcano operators is similar to the one used in "real," commercial systems, the techniques described here can be used to parallelize other query processing engines.

1. Introduction

In order to provide a testbed for database systems education and research, we decided to implement an extensible and modular query processing system. One important goal was to achieve flexibility and extensibility without sacrificing efficiency. The result is a small system, consisting of less than two dozen core modules with a total of about 15,000 lines of C code. These modules include a file system, buffer management, sorting, top-down 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.

The last module, called the *exchange* module, is the focus of this paper. It was designed and implemented *after* most of the other query processing modules. The design goal was to parallelize all existing query processing algorithms without modifying their implementations. Equivalently, the goal was to allow parallelizing new algorithms not yet invented without requiring that these algorithms be implemented with concern for parallelism. This goal was met almost entirely; the only change to the existing modules concerned device names and numbers to allow horizontal partitioning over multiple disks, also called disk striping [25].

Parallelizing a query evaluation engine using an operator is a novel idea: earlier research projects used template processes that encompass specific operators. We call the new method of parallelizing the *operator model*. In this paper, we describe this new method and contrast it with the method used in GAMMA and Bubba, which we call the *bracket model*. Since we developed, implemented, and tested the operator model within the framework of the Volcano system, we will describe it as realized in Volcano.

Volcano was designed to be extensible; its design and implementation follows many of the ideas outlined by Batory et al. for the GENESIS design [5]. In this paper, we do not focus on or substantiate the claim to extensibility and instead refer the reader to [17]; suffice it to point out that if new operators use and provide Volcano's standard interface between operators, they can easily be included in a Volcano query evaluation plan and parallelized by the exchange operator.

Volcano's mechanism to synchronize multiple operators in complex query trees within a single process and to exchange data items between operators are very similar to many commercial database systems, e.g., Ingres and the System R family of database systems. Therefore, it seems fairly straightforward to apply the techniques developed for Volcano's exchange operator and outlined in this paper to parallelize the query processing engines of such systems.

This paper is organized as follows. In the following section, we briefly review previous work that influenced our design, and introduce the *bracket* model of parallelization. In Section 3, we provide a more detailed description of Volcano. The *operator* model of parallelization and Volcano's *exchange* operator are 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.

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.

At the start in 1987, we felt that some decisions in WiSS [11] and GAMMA [12] were not optimal for performance or generality. 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]. However, many of the design decisions in Volcano were strongly influenced by experiences with WiSS and GAMMA. The design of the data exchange mechanism between operators, the focus of this paper, is one of the few radical departures from GAMMA's design.

During the design of the EXODUS storage manager [10], 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 [24] influenced the strong emphasis on iterators for query processing. The design of GENESIS [5] emphasized the importance of a uniform iterator interface.

Finally, a number of conventional (relational) and extensible systems have influenced our design. Without further discussion, we mention Ingres [27], System R [3], Bubba [2], Starburst [26], Postgres [28], and XPRS [29]. Furthermore, there has been a large amount of research and development in the database machine area, such that there is an international workshop on the topic. Almost all database machine proposals and implementations utilize parallelism in some form. We certainly have learned from this work and tried to include its lessons in the design and implementation of Volcano. In particular, we have strived for simplicity in the design, *mechanisms* that can support a multitude of *policies*, and efficiency in all details. We believe that the query execution engine should provide mechanisms, and that the query optimizer should incorporate and decide on policies.

Independently of our work, Tandem Computers has designed an operator called the *parallel operator* which is very similar to Volcano's exchange operator. It has proven useful in Tandem's query execution engine [14], but is not yet documented in the open literature. We learned about this operator through one of the referees. Furthermore, the distributed database system R* used a technique similar to ours to transfer data between nodes [31]. However, this operation was used only to effect data transfer and did not support data or intra-operator parallelism.

2.1. The Bracket Model of Parallelization

When attempting to parallelize existing single-process Volcano software, we considered two paradigms or models of parallelization. The first one, which we call the *bracket model*, has been used in a number of systems, for example GAMMA [12] and Bubba [2]. The second one, which we call the *operator model*, is novel and is described in detail in Section 4.

[1] This statement only pertains to the original version of WiSS as described in [11]. Both decisions were reconsidered for the version of WiSS used in GAMMA.

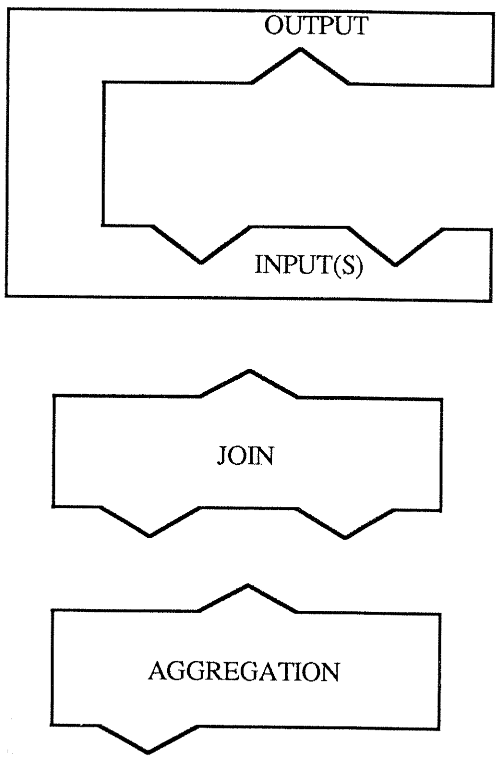

Figure 1. Bracket Model of Parallelization.

In the bracket model, there is a generic process template that can receive and send data and can execute exactly one operator at any point of time. A schematic diagram of such a template process is shown in Figure 1 with two possible operators, join and aggregation. The code that makes up the generic template invokes the operator which then controls execution; network I/O on the receiving and sending sides are performed as service to the operator on request, implemented as procedures to be called by the operator. The number of inputs that can be active at any point of time is limited to two since there are only unary and binary operators in most database systems. The operator is surrounded by generic template code which shields it from its environment, for example the operator(s) that produce its input and consume its output.

One problem with the bracket model is that each locus of control needs to be created. This is typically done by a separate scheduler process, requiring software development beyond the actual operators, both initially and for each extension to the set of query processing algorithms. Thus, the bracket model seems unsuitable for an extensible system.

In a query processing system using the bracket model, operators are coded in such a way that network I/O is their only means of obtaining input and delivering output (with the exception of scan and store operators). The reason is that each operator is its own locus of control and network flow control must be used to coordinate multiple operators, e.g., to match two operators' speed in a producer-consumer relationship. Unfortunately, this also means that passing a data item from one operator to another always involves expensive inter-process communication (IPC) system calls, even in the cases when an entire query is evaluated on a single machine (and could therefore be evaluated without IPC in a single process) or when data do not need to be repartitioned among nodes in a network. An example for the latter is the three-way join query "joinCselAselB" in the Wisconsin Benchmark [6,9] which uses the same join attribute for both two-way joins. Thus, in queries with multiple operators (meaning almost all queries), IPC and its overhead are mandatory rather than optional.

In most (single-process) query processing engines, operators schedule each other much more efficiently by means of procedure calls rather the system calls. The concepts and methods needed for operators to schedule each other using procedure calls are the subject of the next section.

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 to manage devices, buffer pools, files, records, and B^+-trees. For a detailed discussion, we refer to [17].

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 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.

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 that consists of 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 at any point of time) 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 [24] and the algebraic query evaluation system of the Starburst extensible relational database system [22].

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 their 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.

4. The Operator Model of Parallelization

When porting Volcano to a multi-processor machine, we felt it desirable to use the single-process query processing code described above *without any change.* The result is very clean, self-scheduling parallel processing. We call this novel approach the *operator model* of parallelizing a query evaluation engine. In this model, all issues of control are localized in one operator that uses and provides the standard iterator interface to the operators above and below in a query tree.

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 2 shows a complex query execution plan that includes data processing operators, e.g. file scan and join, and exchange operators.

This section describes how the *exchange* iterator implements vertical and horizontal parallelism followed by a detailed example and a discussion of alternative modes of operation of Volcano's *exchange* operator.

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

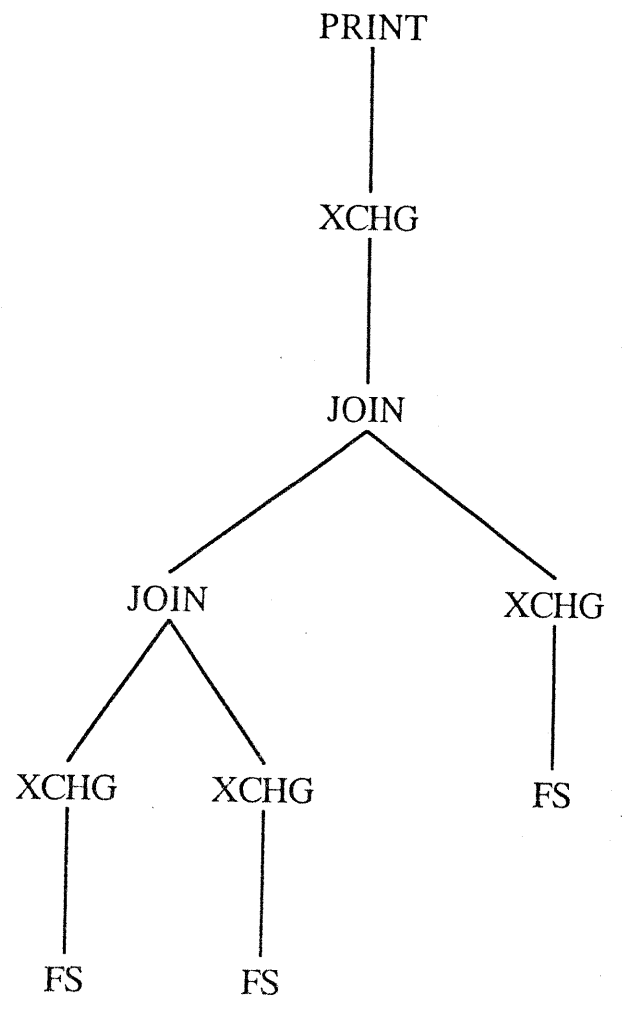

Figure 2. Operator Model of Parallelization.

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 rather than iterator (procedure) calls. 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*, 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 32,000 records. When a packet is filled, it is inserted into a linked list originating 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, passes it to the consumer, and waits until the consumer allows closing all open files. This delay is necessary in Volcano because files on virtual devices must not be closed before all their records are unpinned in the buffer. In other words, it is a peculiarity due to other design decisions in Volcano rather than inherent in the exchange iterator or the operator model of parallelization.

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 reasons for this change in paradigms. First, we intend to use the exchange operator also for *horizontal parallelism*, to be described below, which is easier to implement with data-driven dataflow. 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 very high-performance query evaluation require 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 [12, 13].

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 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[2].

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[3]. 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 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 [15]. 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 have considered using *primed processes*, i.e., processes that are always present and wait for work packets. Primed processes are used in many commercial database systems. Since portable distribution of compiled code (for support functions) is not trivial, we delayed this change and plan on using primed processes only when we move to an environment with multiple shared-memory machines[4]. Others have also observed the high cost of process creation and have provided alternatives, in particular "light-weight" processes in various forms, e.g., in Mach [1].

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 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.

4.3. 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 2a), without knowledge that a process boundary will be crossed, a consequence of anonymous inputs in Volcano. When X is *open*ed, it creates a port with one input queue for A_0 and forks BC_0 (Figure 2b), which in turn forks BC_1 and BC_2 (Figure 2c). 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 2d) which in turn forks D_1 to D_3 (Figure 2e).

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 3x4=12 flagged packets) with an *end-of-stream* tag and then waits on a semaphore for permission to *close*. The copies of the

[2] 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 [30] and pipelining in the instruction execution. Since Volcano is a software architecture and following the analysis in [8], we do not consider hardware parallelism further.

[3] In general, sorted streams can be piped directly into the join, both in the single-process and the multi-process case. Volcano's sort operator includes a parameter "final merge fan-in" that allows sharing the merge space by two sort operators performing the final merge in an interleaved fashion as requested by the merge join operator.

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

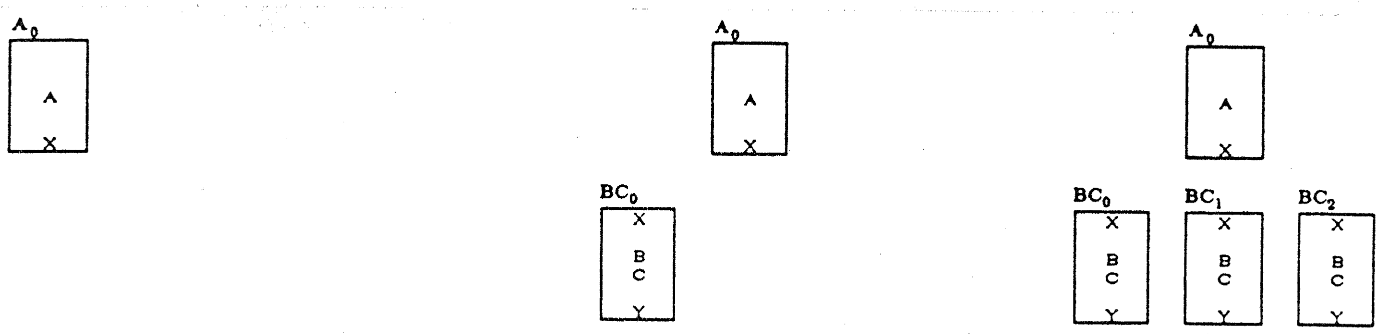

Figure 3a-c. Creating the BC processes.

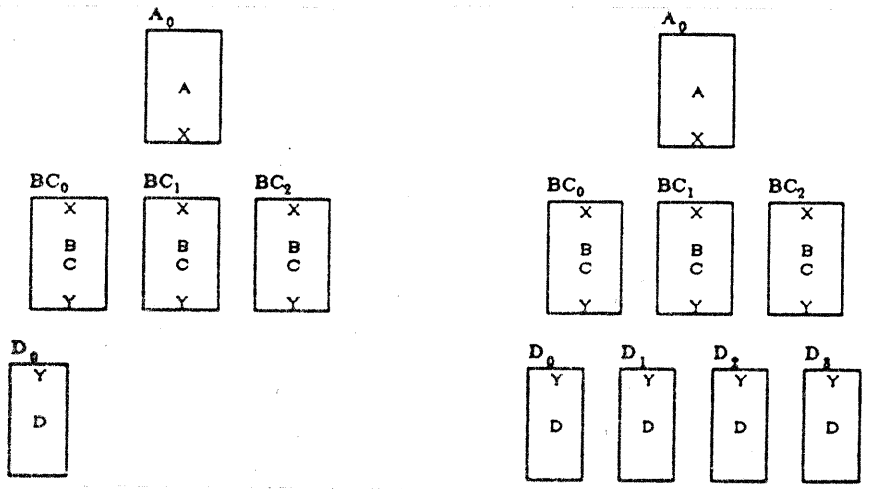

Figure 3d-e. Creating the D processes.

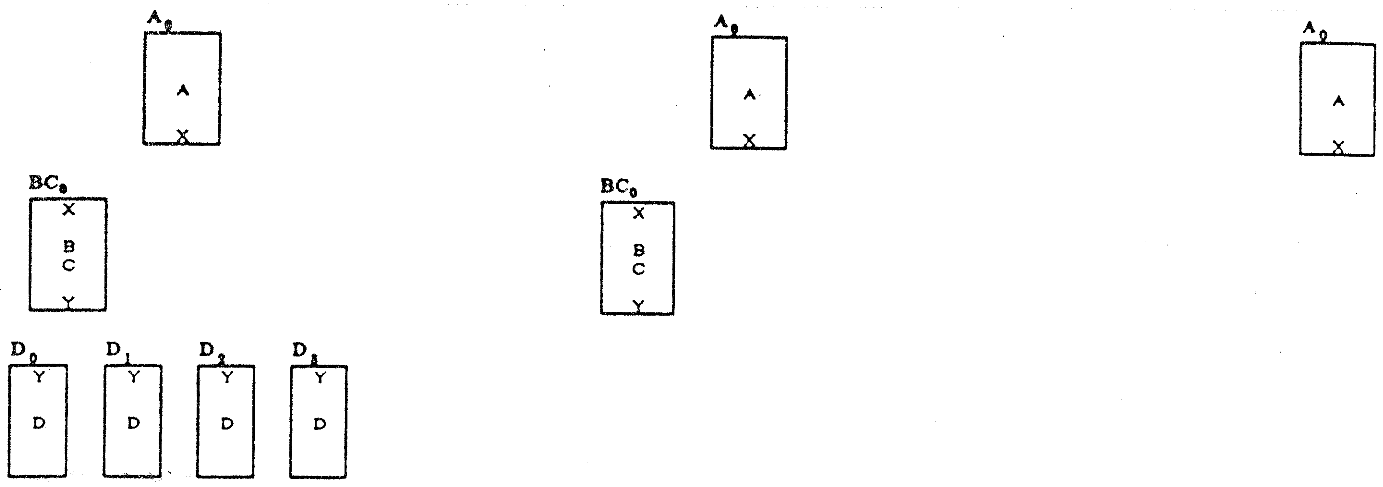

Figure 3f-h. Closing all processes down.

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 by *C* 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 *close*d. Closing the exchange operator X includes releasing the semaphore that allows the BC processes to shut down (Figure 3f). The X driver in each BC process *close*s its input, operator B. B *closes* C, and C *close*s 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 3g). When all BC processes have *close*d down, X's *close* procedure indicates the fact to A_0 and query evaluation terminates (Figure 3h).

4.4. Variants of the Exchange Operator

There are a number of situations for which the *exchange* operator described so far required some modifications or extensions. In this section, we outline additional capabilities implemented in Volcano's 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 [19] 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 [4]. 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 [19] took only about three hours and yielded not insignificant speedups.

When we implemented and benchmarked parallel sorting [21], we found it useful to add 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.* All these modifications were implemented in such a way that they support multi-level merge trees, e.g., a parallel binary merge tree as used in [7]. The merging paths are selected automatically such that the load is distributed as evenly as possible in each level.

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. When 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 and running 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 allowing processes to migrate freely and a large cache associated with each CPU, the frequent 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.

In summary, the operator model of parallel query evaluation provides for self-scheduling parallel query evaluation in an extensible database system. The most important properties of this novel approach are that the new module 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.

[5] Whether exchange forks new producer processes (the original exchange design describe in Section 4.1) or uses the existing process group to execute the producer operations is a runtime switch.

5. Overhead and Performance

From the beginning of the Volcano project, we were very concerned about performance and overhead. In this section, we report on experimental measurements of the overhead induced by the exchange operator. This is not meant to be an extensive or complete analysis of the operator's performance and overhead; the purpose of this section is to demonstrate that the overhead can be kept in acceptable limits.

We measured elapsed times of a program that creates records, fills them with four random integers, passes the records over three process boundaries, and then unfixes the records in the buffer. The measurements are elapsed times on a Sequent Symmetry with twelve Intel 16 MHz 80386 CPU's. This is a shared-memory machine with a 64 KB cache for each CPU. Each CPU delivers about 4 MIPS in this machine. The times were measured using the hardware microsecond clock available on such machines. Sequent's DYNIX operating system provides exactly the same interface as Berkeley 4.2 BSD or System V UNIX and runs (i.e., executes system calls) on all processors.

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. In a separate experiment, we reran the program multiple times varying the number of records per exchange packet. Table 1 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 three groups included three processes; thus, each of the producer processes created 33,333 records. All these experiments were conducted with flow control enabled with three "slack" packets per exchange. We used different partitioning (hash) functions for each exchange iterator to ensure that records were passing along all possible data paths, not only along three independent pipelines.

Packet Size [Records]	Elapsed Time [Seconds]
1	176.4
2	97.6
5	45.27
10	27.67
20	20.15
50	15.71
100	13.76
200	12.87
250	12.73

Table 1. Exchange Performance.

As can be seen in Table 3, 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 176 seconds to 98 seconds. As the packet size was increased further, the elapsed time shrank accordingly, to 15.71 seconds for 50 records per packet and 12.73 seconds for 250 records per packet.

It seemed reasonable to speculate that for small packets, most of the elapsed time was spent on data exchange. To verify this hypothesis, we calculated regression and correlation coefficients of the number of data packets (100,000 divided over the packet size) and the elapsed times. We found an intercept (base time) of 12.18 seconds, a slope of 0.001654 seconds per packet, and a correlation of more than 0.99. Considering that we exchanged data over three process boundaries and that on two of those boundaries there were three producers and three consumers, we estimate that the overhead was 1654μsec / 1.667 = 992μsec per packet and process boundary.

Two conclusions can be drawn from these experiments. First, vertical parallelism can pay off even for very simple query plans if the overhead of data transfer is small. Second, since the packet size can be set to any value, the overhead of Volcano's exchange iterator is negligible.

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 [19-21,23].

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.

Using an operator to encapsulate parallelism as explored in the Volcano project has a number of advantages over the bracket model. First, it hides the fact that parallelism is used from all other operators. Thus, other operators can be implemented without consideration for parallelism. Second, since the exchange operator uses the same interface to its input and output, it can be placed anywhere in a tree and combined with any other operators. Hence, it can be used to parallelize new operators, and effectively combines extensibility and parallelism. Third, it does not require a separate scheduler process since scheduling (including initialization, flow control, and final clean-up) is part of the operator and therefore performed within the standard *open-next-close* iterator paradigm. This turns into an advantage in two situations. When a new operator is integrated into the system, the scheduler and the template process would have to be modified, while the exchange operator does not require any modifications. When the system is ported to a new environment, only one module requires modifications, the exchange iterator, not two modules, the template process and the scheduler. Fourth, it does not require that operators in a parallel query evaluation system use IPC to exchange data. Thus, each process can execute an arbitrary subtree of a complex query evaluation plan. Fifth, a single process can have any number of inputs, not just one or two. Finally, the operator can be (and has been) implemented in such a way that it can multiplex a single process between a producer and a consumer. In some respects, it efficiently implements application-specific co-routines or threads.

We plan on several extensions of the exchange operator. First, we plan on extending our design and implementation to support both shared and distributed memory ("shared-nothing architecture") and to allow combining these concepts in a closely tied network of shared-memory multicomputers while maintaining the encapsulation properties. This might require using a pool of "primed" processes and interpreting support functions. We believe that in the long run, high-performance database machines, both for transaction and query processing, will employ this architecture. Second, we plan on devising a error and exception management scheme that makes exception notification and handling transparent across process and machine boundaries. Third, we plan on using the exchange operator to parallelize query processing in object-oriented database systems [16]. In our model, a complex object is represented in memory by a pointer to the root component (pinned in the buffer) with pointers to the sub-components (also pinned) and passed between operators by passing the root component [18]. While the current design already allows passing complex objects in a shared-memory environment, more functionality is needed in a distributed-memory system where objects need to be packaged for network transfer.

Volcano is the first implemented query evaluation system that combines extensibility and parallelism. Encapsulating all parallelism issues into one module was essential to making this combination possible. 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. Since the operator model of parallel query processing and Volcano's exchange operator encapsulates parallelism and both uses and provides an iterator interface similar to many existing database systems, the concepts explored and outlined in this paper may very well be useful in parallelizing other database query processing software.

Acknowledgements

A number of friends and colleagues were great sounding boards during the design and implementation of parallelism in Volcano, most notably Frank Symonds and Leonard Shapiro. Jerry Borgvedt implemented a prototype distributed-memory exchange operator. — NSF supported this work with contracts IRI-8805200 and IRI-8912618. Sequent Computer Systems provided machine time for experiments on a large machine.

References

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Summer Conference Proceedings 1986*, .
2. 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.
3. 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 (June 1976), 97-137.
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. D. S. Batory, "GENESIS: A Project to Develop an Extensible Database Management System", *Proceedings of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA., September 1986, 207-208.
6. D. Bitton, D. J. DeWitt and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", *Proceeding of the Conference on Very Large Data Bases*, Florence, Italy, October-November 1983, 8-19.
7. D. Bitton, H. Boral, D. J. DeWitt and W. K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations", *ACM Transactions on Database Systems 8*, 3 (September 1983), 324-353.
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 and D. J. DeWitt, "A Methodology for Database System Performance Evaluation", *Proceedings of the ACM SIGMOD Conference*, Boston, MA., June 1984, 176-185.
10. 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*, Kyoto, Japan, August 1986, 91-100.
11. 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 (October 1985), 943-962.
12. 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.

13. 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).

14. 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).

15. R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms", *Ph.D. Thesis*, Madison, October 1986.

16. G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: A Prospectus", in *Advances in Object-Oriented Database Systems*, vol. 334 , K. R. Dittrich (editor), Springer-Verlag, September 1988, 358-363.

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

18. G. Graefe, "Set Processing and Complex Object Assembly in Volcano and the REVELATION Project", *Oregon Graduate Center, Computer Science Technical Report*, Beaverton, OR., June 1989.

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

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

21. G. Graefe, "Parallel External Sorting in Volcano", *submitted for publication*, February 1990.

22. 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.

23. 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.

24. 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.

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

26. 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*, Pacific Grove, CA., September 1986, 85-92.

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

28. M. Stonebraker and L. A. Rowe, "The Design of POSTGRES", *Proceedings of the ACM SIGMOD Conference*, Washington, DC., May 1986, 340-355.

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

30. 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*, Los Angeles, CA., February 1988, 194-201.

31. P. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms and R. Yost, "R*: An Overview of the Architecture", in *Readings in Database Systems*, M. Stonebraker (editor), Morgan-Kaufman, San Mateo, CA., 1988.