SOFTWARE—PRACTICE AND EXPERIENCE. VOL. 22(7), 495–517 (JULY 1992)

Tuning a Parallel Database Algorithm on a Shared-memory Multiprocessor

GOETZ GRAEFE
University of Colorado at Boulder, Boulder, CO 80309-0430, U.S.A.

AND

SHREEKANT S. THAKKAR
Sequent Computer Systems, 15450 SW Koll Parkway, Beaverton, OR 97006-6063, U.S.A.

SUMMARY

Database query processing can benefit significantly from parallelism. Parallel database algorithms combine substantial CPU and I/O activity, memory requirements, and massive data exchange between processes, all of which must he considered to obtain optimal performance. Since parallel external sorting is a very typical example, we have focused on sorting to tune Volcano, a new query processing system. The purpose of the Volcano project is to provide efficient, extensible tools for query and request processing in novel application domains, particularly in object-oriented and scientific database systems, and for experimental database performance research. It includes all query processing algorithms conventionally used in relational database systems as well as several new ones, and can execute all of them in parallel. In this article, we present Volcano's parallel external sorting algorithm and a sequence of enhancements to improve its performance. We obtained very good absolute performance, 84 seconds for 100 MB of data, as well as near-linear speedup with sixteen CPUs and disks. Furthermore, these results were achieved on a shared-memory machine despite the common belief that parallel query processing is best implemented on distributed-memory systems. We detail our tuning measures and report on their effectiveness.

KEY WORDS Database systems Query processing Sorting Performance Parallelism Shared memory

1. INTRODUCTION

Several research and development projects over the last decade have shown that database query processing can benefit significantly from parallel algorithms, as shown by several research projects and products.[1,2] The main reasons why parallelism is relatively easy to exploit in database query processing systems are that (1) query processing is performed using a tree of operators which can be executed in separate processes and processors connected with pipelines (*inter* -operator parallelism), and (2) each operator consumes and produces sets which can be partitioned or fragmented into disjoint subsets to be processed in parallel (*intra* -operator parallelism). Both forms of parallelism require data exchange between processes. For example, in order to perform a relational join by two processes, one process could join all tuples with odd join attribute values (from both join inputs) while the other joins tuples with even join attribute values. If a tree includes multiple operations on different attri-

0038–0644/92/070495–23$16.50

Received 11 September 1991
Revised 30 December 1991

butes, data must be repartitioned between operations, even if the same processes are used for multiple operations.

Parallel query processing algorithms are characterized by their complex combination of resource needs. While most research on parallel algorithms focuses on processing and communication, parallel database algorithms also require I/O, preferably parallel, [3] and usually benefit from large memories. [4,5] For best performance, it is imperative to balance not only CPU and IPC performance but also I/O bandwidth and memory allocation.

Sorting is an excellent example for a parallel query processing algorithm. Most parallel query processing algorithms require significant CPU and I/O activity as well as massive data exchange between processes. All these elements are present in sorting; thus, gaining experience with and understanding of the performance of parallel sorting will help tuning other parallel database algorithms as well. *I/O activity* is required not only to read the input and write the final sorted file, but also to write intermediate sorted runs and to merge them. Input and final output are sequential as is writing the sorted runs, whereas input for merging is random and requires many seeks on disk. The *CPU activity* stems from several sources. The main sources in most database systems are probably interpretation of predicates, comparisons, hash functions, etc.; copying data between different work areas and for assembling pages of intermediate files; concurrency control (latching and locking); and internal table management, in particular in the buffer manager. The *data exchange* needs depend on the original and desired final distribution of data and on the sorting algorithm itself. In our experiments, we assumed that input data are randomly and evenly distributed over several disks and that the desired final data must be *range-partitioned* (non-overlapping key ranges are assigned one to each disk) and sorted within each range. Since we assigned one process to each disk, almost all data had to be passed from one process to another. Finally, it is well-known that external sort performance improves with *memory* size: the larger the available memory, the longer the initial runs and the larger the merge fan-in, i.e. the number of runs that can be combined into one sorted run in a single step.

Novel database applications demand not only high functionality but also high performance. To combine these two requirements, the Volcano project provides efficient, extensible tools for query and request processing in novel application domains, particularly in object-oriented and scientific database systems. The three mainstays of Volcano's performance are a new optimizer generator, [6,16] efficient query execution algorithms, and parallel execution based on a novel 'parallelism' operator that allows several forms of parallel execution in any combination. [7,8] Volcano's design goals are extensibility, modularity, performance, effective use of parallelism, and versatility as experimental platform. For the last goal, we focused on *mechanisms* to support *policies* chosen by a human experimenter or a query optimizer. Volcano includes all query processing algorithms conventionally used in relational database systems (file and index scans and maintenance, sort- and hash-based join, semi-join, outer join, intersection, union, difference, aggregation, and duplicate elimination) as well as several new ones (e.g., hash-based relational division or universal quantification, [9]) and can execute all of them in parallel. As with any system designed for performance research, we had to spend a fair amount of effort on tuning. Since only the best sequential algorithms should be parallelized, we focused on both complementary aspects, i.e., sequential performance and parallel execution.

In this article, we present Volcano's parallel sorting algorithm and a sequence of performance enhancements that we used to tune parallel sorting on our machine. This study was initiated because other researchers had preferred 'shared-nothing' [10] or distributed memory over shared memory for parallel database machines, e.g., Gamma, [1] Bubba, [11] Teradata, [12] and Tandem. [2] Our goal was to explore how or how far shared memory can be used for parallel database query processing. Using all the improvements explored in this study, we observed almost linear speedup from 2 to 16 processors and disks, with 84 seconds for 16 processors and disks for sorting a 100 MB file. The tuning efforts resulted in both improved throughput per processor and disk and improved parallelism speed-up. We believe that the techniques explored in this article are applicable to a variety of other algorithms within Volcano as well as to other query processing systems.

Several improvements were not obvious to us at the beginning; therefore, we guide the reader through our tuning study step by step in the same way we explored it. In the next section, we present an overview of the Volcano query processing software and its sort algorithms. Section 3 describes the hardware platform, the benchmark workload, and the initial performance measurements. In Sections 4 to 6, we present the tuning modifications and report on their effectiveness. In Section 7, we summarize the study and present our conclusions.

2. OVERVIEW OF VOLCANO

In this section, we provide a brief overview of Volcano. Most of the design has been described elsewhere, [7–9,13,14] and is provided here to show Volcano's similarity to the query execution mechanisms of existing database systems. This similarity ensures that the experiences and conclusions reported here are applicable to other systems as well.

At the current time, Volcano is a query execution engine only—it does not include a high-level user interface, a data model, a schema, or a query optimizer. It was designed and implemented as a research tool for query processing algorithms and strategies. As such, it provides *mechanisms* from which an experimenter can choose. Very soon we will also provide a query optimizer that embeds and determines *policies*. Building on earlier experience, [15] we are currently designing a new optimizer generator for Volcano. [6,16]

Most of Volcano's file system is rather conventional. It provides data files, scans with predicates, and B^+-tree indices. The unit of I/O and buffering, called a *cluster* in Volcano, is set for each file individually when it is created. Files with different cluster sizes can reside on the same device. Volcano uses its own buffer manager and bypasses operating system buffering by using raw devices.

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. Calling open on a Volcano iterator prepares the iterator to produce data. The next operation returns exactly one data item or an end-of-stream error. It is meant to be called repeatedly until this error is returned. The close operation performs final house-keeping tasks such as deallocating a hash table.

Associated with each algorithm is a *state record.* The arguments for the algorithms, e.g. a hash table size or a predicate evaluation function, 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, allowing the query processing modules to be implemented without knowledge of or constraint on the internal structure of data objects. Support functions are compiled prior to execution and passed to the processing algorithms by means of pointers to the function entry points, although mechanisms to use an interpreter are also provided.

In complex queries or algebra expressions, state records are linked together by means of *input* pointers. All state information for an iterator is kept in its state record; thus, an algorithm may be used many times in a query by including more than one state record in the query. The input pointers are also kept in the state records. An input is represented by a structure that consists of four pointers to the entry points of the three procedures implementing the operator (the open, next, and close procedures) and a state record. Because an operator invokes its input by means of function pointers, 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 and space for single process query evaluation. They have been used in numerous relational database systems such as System R, SQL/DS, DB2, Ingres, Informix, and Oracle, as well as in the E database implementation language [17, 18] and the algebraic query evaluation system of the Starburst extensible relational database system. [19, 20]

The tree-structured query evaluation plan is used to execute queries by demand-driven dataflow. The return value of next is a structure called NEXT_RECORD which consists of a record identifier and a record address in the buffer pool. This record is pinned in the buffer. The protocol for 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 retain it for a while (e.g. in a hash table or while quicksorting a pointer array), unfix it (e.g., when a predicate fails), or pass it on to the next operator. Complex operations such as join that create new records have to fix them in the buffer before passing them on, and have to unfix their input records.

This protocol of fixing records combines two important advantages over mechanisms used in several commercial systems that copy data out of the I/O buffer and use a 'data transfer area' between any two operators. First, Volcano's procedure permits transfer of records between operators entirely without copying. This is a significant consideration, in particular on a shared-memory machine with a single bus. Second, the protocol is flexible enough for individual records to pass or fail predicates in subsequent operators.

The concept of anonymous inputs, i.e., each operator using a function pointer to invoke its input operator(s), is commonly used in database query processing systems. In Volcano, it was exploited further by the design and implementation of a 'parallelization' operator that interfaces with all other operators and encapsulates all parallelism issues. [7] This module is described in the next section.

2.1. Multi-processor query evaluation

When we considered exploiting multi-processors with Volcano, we decided that it would be desirable to use the query processing code *without any change*. The result is very clean, self-scheduling parallel processing. The module responsible for parallel execution and synchronization is the exchange iterator. Since it is an iterator with open, next, and close procedures, it can be inserted at any one place or at multiple places in a complex query tree.

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 now 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 to 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 may 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 invoking open, next, and close on its input. The output of next is collected in *packets* of 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. The last packet is marked with an end-of-stream *tag* to inform the consumer that no more input is available.

While all other modules are based on demand-driven dataflow (iterators, lazy evaluation), the producer-consumer relationship within the exchange operator uses data-driven dataflow (eager evaluation). If the producers are significantly faster than the consumers, they may pin a significant portion of the buffer, thus impeding overall system performance. A run-time switch of exchange enables *flow control* or *back pressure* using an additional semaphore, 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.

There are two forms of horizontal parallelism which we call *bushy parallelism* and *intra-operator* parallelism. In bushy parallelism, different CPUs 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 CPUs perform the same operator on different subsets of a stored dataset or an intermediate result.

Bushy parallelism can easily be implemented by inserting one or two exchange operators into a query tree. 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 queue. The producers use a support function to decide which of the queues (or actually, which of the packets being filled by the producer) an output record has to go to. Using a support function allows implementing round-robin, key range, or hash partitioning.

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. Furthermore, typically non-repetitive actions like mount must be invoked by the query root process before or after a query is evaluated by multiple processes.

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 algorithms. 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 descriptor 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.

2.2. Volcano's sort algorithm

Sorting contributes probably more to the cost of database query processing than any other algorithm. Sorting can be required for two reasons. First, a user request may state explicitly that the result data be returned in a particular sort order, e.g., using the SQL ORDER BY clause. Second, several query processing algorithms, in particular for join and aggregation, require sorted input data. [21,22] Although equivalent hash-based algorithms are commonly regarded as more efficient, there are cases in which sort-based algorithms outperform hash-based algorithms, in particular in complex queries that permit exploiting *interesting orderings.* [23,24]

External sorting is known to be an expensive operation, and many sorting algorithms have been devised; most relevant techniques have been described by Knuth . [25] For Volcano, we needed a simple, robust, and efficient algorithm. Therefore, we opted for quicksort in main memory with subsequent merging. The initial runs are as large as the sort space in memory. Initial runs are also called level-0 runs. When several level-0 runs are merged, the output is called a level-1 run. The sort module does not impose a limit on the size of the sort space, the fan-in of the merge phase or the number of merge levels in Volcano.

In order to ensure that the sort module interfaces well with the other operators in Volcano, e.g., file scan or merge join, we had to implement it as an iterator, i.e., with open, next, and close procedures. Most of the sort work is done during open. This procedure consumes the entire input and leaves appropriate data structures for next to produce the final, sorted output. If the entire input fits into the sort space in main memory, open leaves a sorted array of pointers to records in the buffer which is used by next to produce the records in sorted order. If the input is larger than main memory, the open procedure creates sorted runs and merges them until only one final merge step is left, i.e., the number of runs is less than or equal to the maximal fan-in. The last merge step is performed in the next procedure, i.e., when demanded by the consumer of the sorted stream. Similarly, the input to

Volcano's sort module must be an iterator, and sort uses open, next, and close procedures to request its input.

2.3. Volcano's parallel sort algorithms

Much work has been dedicated to parallel sorting, but only a few algorithms have been implemented in database settings, i.e., where the total amount of data is a large multiple of the total amount of main memory in the system. Most algorithms are variants of the well-known merge-sort technique and require a final centralized merge step. [26–29] In a highly parallel architecture, any centralized component that has to process all data is bound to be a severe bottleneck.

Most recent investigations of parallel sorting focussed on load-balancing issues [30–33] In this paper, we assume that sufficiently accurate quantiles for partitioning and therefore load balancing are available, e.g., based on sampling techniques. [34] We have described our algorithm in detail in, [14] so we only provide an overview here.

Volcano's preferred sort method starts by exchanging data based on logical keys. [14] Provided a sufficiently fast network in the first step, all data exchange can be done in parallel with no node depending on a single node for input values. First, all sites with data redistribute the data to all sites where the sorted data will reside. Second, all sites that have received data sort them locally. This algorithm does not contain a centralized bottleneck, but it creates a new problem. The local sort effort is determined by the amount of data to be sorted locally. To achieve high parallelism in the local sort phase, it is imperative that the amount of data be balanced among the receiving processors. The amount of data at each receiving site is determined by the range of key values that the site is to receive and sort locally, and the number of data items with keys in this range. In order to balance the local sorting load, it is necessary to estimate the quantiles of the keys at all sites prior to the redistribution step. Quantiles are key values that are larger than a certain fraction of key values in the distribution, e.g., the median is the 50 per cent or 0.5 quantile. * For load balancing among N processors, the i/N quantiles for $i = 1, \ldots . N - 1$ need to be determined, e.g., by sampling. [34]

This sorting method, data exchange followed by local sorts, can readily be implemented using the methods and modules described so far, namely the exchange module and the sort iterator. In fact, since the exchange iterator has been modified for distributed-memory and hierarchical-memory machines, Volcano's sort iterator can also be used on these architectures and presumably scaled to very high degrees of parallelism.

3. ENVIRONMENT, WORKLOAD, AND INITIAL PERFORMANCE

3.1. Hardware and operating system environment

In order to explore shared-memory query processing, we used a Sequent Symmetry machine, which is a bus-based shared-memory multiprocessor that can contain from

* If the distribution is skewed, the mean and the median can differ significantly. Consider the sequence 1,1,1,2, 10,10,10. The mean is 35/7 = 5, whereas the median is 2.

two to thirty CPUs. [35] Each processor subsystem contains a 32-bit microprocessor, a floating point unit, and a private cache. The system features a 53 MB/s pipelined system bus, up to 240 MB of main memory, and a diagnostic and console processor. A Symmetry can support five dual-channel disk controllers (DCCs), with up to 8 disks per channel. Each channel can transfer at 1·8MB/s. Earlier experiments exhibited performance impairment if the I/O transfer load is very high and more than 2 disks are attached to a single channel, in particular if sequential I/O dominates.

A Symmetry Model B system with 16 MHz Intel 80386/80387 processors delivering about 4 MIPS per CPU* was used for this study. In this system, each CPU has a private 64 KB local cache supporting a copyback cache coherence protocol. The cache coherence protocol is based on the concept of ownership. To perform a write operation, a cache has to perform first an exclusive read operation on the bus (assuming a cache miss) to gain ownership of the block. Only then can the block be updated in the cache. Thus, if another cache holds the block in *modified* state, it has to respond to the exclusive read request and invalidate its copy.

Hardware synchronization on the Symmetry model uses cache-based locks. The locks are also ownership based. A locked read from a processor is treated like a write operation by the cache controller. The cache controller performs an exclusive read operation on the bus (assuming a cache miss) to gain ownership of the block. The atomic operation is then completed in the cache. These locks are optimized for multi-user systems where locks are lightly contended and the critical sections are short. They do not work very well in some parallel applications in which locks are heavily contended, and other software synchronization schemes can be used to reduce contention for the hardware locks . [36]

The DYNIX operating system is a parallel version of UNIX implemented by Sequent for its Balance and Symmetry machines. It provides all services of AT&T System V UNIX as well as Berkeley 4.2 BSD UNIX.

3.2. Work load and system configuration

As our workload, we used a file with 1,000,000 records of 100 bytes. Sorting this file is one of the three work loads suggested by Gray *et al.* [37] because it allows measuring a system's internal performance independently of the application and the application interface. When the measurements start, the input records are partitioned randomly but evenly over all disks, and the buffer is empty. The measurement stops after all records have been written to disk, range-partitioned and sorted within each range. In range partitioning, the key domain is split into disjoint ranges and each disk drive has one subset assigned to it.

The experimental system was equipped with 10 CPU boards (20 CPUs), 5 DCCs (10 channels), and 20 disks, 2 disks on each channel. One of the DCCs and its disks were used for DYNIX file systems. In order to measure only comparable numbers, we used only the other four DCCs in the experiments reported here. These 16 disks were opened in raw mode, i.e., DYNIX did not provide buffering, read-ahead, or write-behind for these disks. Of the 96 MB physical memory available in the system, about 15 per cent were used by the operating system for code, internal tables, and file system I/O buffer space.

* Since we performed this study, faster CPUs have become available. However, the absolute hardware speeds and sizes do not affect the concepts and ideas explored in this tuning study.

In our experiments, we allocated 12 MB of sort space within 15 MB of I/O buffer space. For other tables (e.g. state records and the arrays used in quicksort) and to prevent failure due to memory fragmentation, we allocated another 15 MB, just to be safe. This space was allocated in a shared memory segment; in other words, it was allocated only once independently of how many processes or processors were used. Volcano's page size was set to 4 KB, and the cluster size was set to one page.

3.3. Initial performance measurements

Throughout our experiments, we used the same number of processes as disks. While we observed some performance improvements in earlier experiments using more processes than disks, eliminating one variable makes the sequence of performance measurements reported here more readable. Furthermore, we were particularly interested in exploiting parallel I/O capabilities, and in achieving speedup linear with the I/O capability of the system.

Figure 1 shows the initial measurements taken with our system. The time measurements are shown using a solid line and refer to the labels on the left. The speedups are shown with a dashed line and refer to the labels on the right. The ideal speedup is also shown by the dotted line.

The performance did indeed improve as additional resources were committed to the problem. The performance improvement from 2 to 8 disks was good, reasonably close to linear, although not as close as we had hoped. The parallel efficiency can be calculated as (2 × 1346 s)/(8 × 379 s) = 88·78 per cent. However, there seemed to be a barrier of 300 seconds (5 minutes), which this software or hardware could not break. Since this is rather undesirable, we considered and measured several modifications.

4. DATA ACCESS IMPROVEMENTS

In this section, we report on three directions to improve data access performance. For disk access, a first thought is to cluster pages of a file to eliminate seeks during file scans. Unfortunately (or fortunately), Volcano already clusters data in contiguous disk space. In our experiments, the unit of space allocation was 4 MB of contiguous

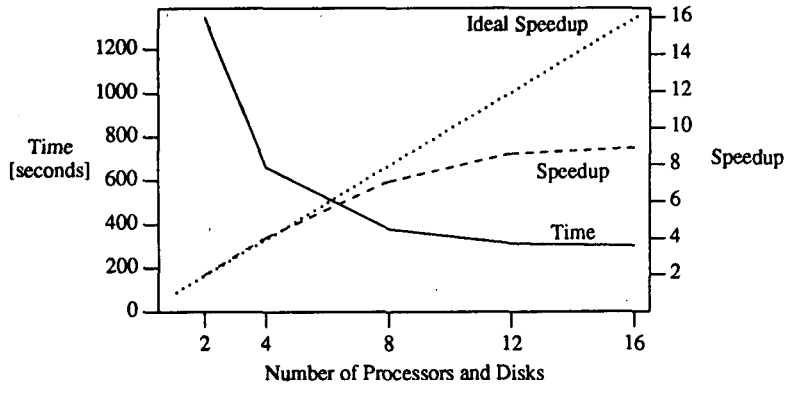

Figure 1. Initial measurements

disk space. Another method to improve disk bandwidth is to increase the unit of I/O. If more data is transferred with each operation, total I/O initialization, seek, and latency costs are reduced.

Because memory accesses on word or cache line tends to be faster than unaligned access, packing records densely on pages might not align records in an optimal way. Thus, 'padding' might be introduced to improve memory access speed. The sensible alignment choices for our system are unaligned, dense record layout to minimize storage requirements; 4-byte alignment according to CPU word boundaries; or 16-byte alignment to ensure that record boundaries coincide with cache line boundaries.

In order to avoid I/O operations, it may be beneficial to increase the buffer size. In general, fewer I/O and improved performance will result. While this makes sense in general, it does not pay off in our situation, as will be explained below. In this section, we report the effects of changing the I/O cluster size, of aligning records (and comparison keys), and of increasing the buffer size.

4.1. Increasing the unit of I/O and buffering

Volcano is designed to support units of I/O larger than single pages. Such units, called *clusters,* can be any multiple of a page. All requests to the buffer and I/O managers are in terms of clusters. When a new cluster is requested, the buffer manager removes clusters from the bottom of its LRU* list and deallocates their memory until the required new cluster will fit into the preset buffer size, and then allocates an appropriate segment of memory for the new cluster. The buffer manager does not shuffle data; rather, it calls on the memory manager to find a suitable contiguous segment in memory. For these experiments, the page size was compiled to be 4 KB.

Figure 2 shows the effects of increasing the cluster size. When compared to the initial measurements, there was a definite improvement, both for 8 disks and for 16 disks. However, while there seemed to be a limit of 300 seconds for 4 KB clusters,

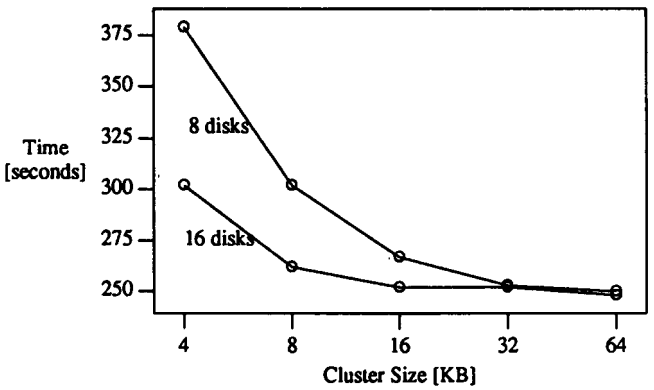

Figure 2. Increased cluster sizes

* *L* east-*R* ecently-*U* sed is a simple and widely used (though not always optimal) buffer replacement algorithm; see Reference 38 for a survey of database buffer management techniques. Volcano's buffer manager augments LRU with hints to keep a page in its LRU list or to toss it immediately, similarly to IBM's Starburst. [20]

there seemed to be a similar limit at about 240 seconds (4 minutes) for larger clusters. Increased cluster sizes, while helpful, did not provide a performance breakthrough nor did they solve the problem why the parallelism speedup is far from linear. In all runs reported below, the cluster size was set to 32 KB.

4.2. Record alignment

We realized that another area for improvement was the alignment of records and sort keys in memory and on disk pages. Record alignment has two effects in database systems. On the positive side, aligning records (and in particular comparison keys within records) on word boundaries will ensure faster access by our CPUs, which use 32-bit words. While the CPUs can perform non-aligned access, aligned access is faster. Similarly, aligning records on cache line boundaries ensures that the same cache line is never held by two CPUs and their caches simultaneously, and may thus contribute to better caching of records. On the negative side, aligning records requires padding. Even if the data volume does not change, the number of records per cluster changes, thus requiring more clusters to be moved from and to disk.

We tried three alignment boundaries. Single-byte alignment stands for densely packed records, 4-byte alignment was chosen to conform with the machine's word size, and 16 bytes were chosen to observe differences in the cache performance if data were aligned according to cache lines. The results in Figure 3 do not show appreciable performance effects, meaning that the positive and negative effects of alignment just about cancelled out in these measurements. Since we intended to study cache performance later on, records were aligned to 16-bytes boundaries in all runs reported below.

4.3. Increasing the buffer size

It is well-known that sort performance is a function of the memory size allocated for sorting. Therefore, we experimented with smaller and larger buffer sizes. In

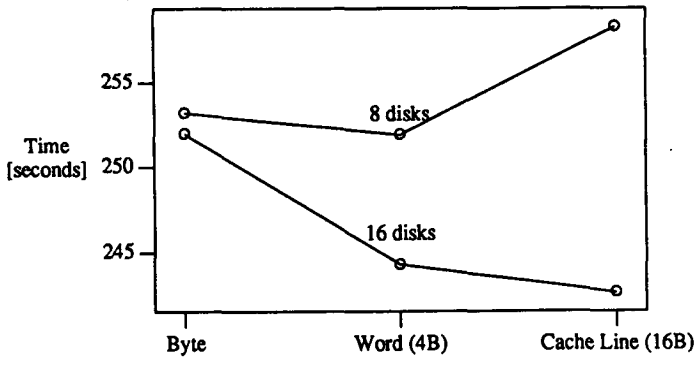

Figure 3. Record alignment

Table I we show the performance for a larger buffer size. Instead of 12 MB sort space, 3 MB additional I/O buffer space, and 15 MB work space administered by Volcano's memory module, we measured the performance for twice the amount of memory.

Interestingly, we observed a small performance degradation, by about 1 or 2 per cent. The explanation can be found in two considerations. First, the increase in buffer size did not reduce the number of merge levels. With 8 MB or more of sort space, only a single merge level is required for 100 MB of data. To avoid merging altogether, 100 MB (the file size) of sort buffer space would be required. Thus, the increase in buffer size did not reduce the number of I/O operations. This confirms Sacco and Schkolnik's observations of discontinuities in the cost functions of database query processing algorithms which led to the design of the hot-set buffer management model . [39] Second, additional sort space requires more management of tables, both internally in Volcano, e.g., when sweeping the hash table during a buffer flush, and externally in the operating system, in particular while forking new processes.

In light of the negligible impact, we left the space allocation at 12 MB sort space, 3 MB additional I/O buffer space, and 15 MB work space.

5. ALLEVIATING CONTENTION IN THE BUFFER MANAGER

At this point, we suspected that Volcano's buffer manager was the source of the problem. The buffer manager was invoked each time a record was unfixed, i.e., when an operator was done with an input record but did not pass it to the next operator. In our implementation, the buffer manager was called 3,000,000 times for this purpose: once for each record by the filter operator that collected records of one partition, once by the sort operator when it had appended a record to a run file, and once by the query processing driver for each record in the sorted output.

During a buffer manager operation, several data structures must be accessed and therefore protected against concurrent updates. In particular, the 'pool lock' must be held while searching or updating the hash tables used to find clusters in the buffer. We instrumented the buffer manager to collect timing information when the buffer pool was locked or a process waited to acquire the buffer pool lock.

Figure 4 shows some statistics about utilization and contention of the buffer pool lock. The dashed lines and the scale on the right indicate the time each process holds or waits for the buffer pool lock, shown as fraction of elapsed time and averaged over all processes. With a constant workload, the total lock hold time is constant and the amount of time each process needs to hold the buffer pool lock is inversely proportional to the number of processes. Since the speedup was not linear, the elapsed time was not inversely proportional to the number of processes, and the fraction of time during which each process held the lock decreases. The waiting

Table I. Increased buffer size

Sort buffer (MBytes)	8 disks time (s)	16 disks time (s)
12, 3, 15	258·31	242·62
24, 6, 30	261·364	247·25

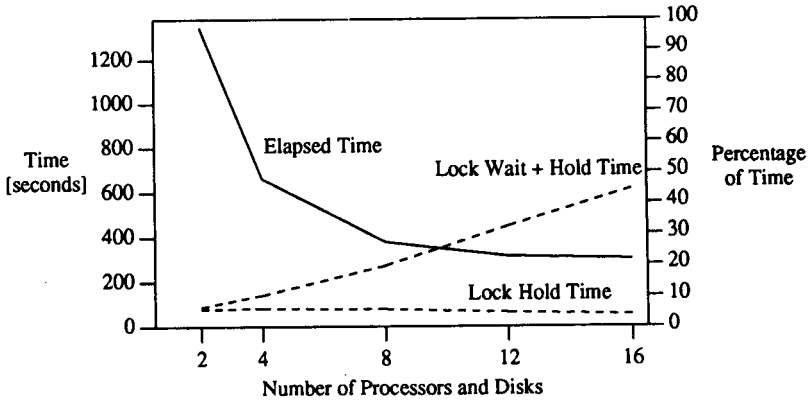

Figure 4. Initial measurements, including buffer statistics

time, on the other hand, increased dramatically as the degree of parallelism increases, and a relief had to be found to obtain better performance.

Figure 4 also explains the performance barrier observed in the initial measurements (Figure 1). Since all buffer operations combined required the pool lock between 150 and 180 seconds, no amount of parallelism could improve the performance beyond this elapsed time.

In this section, we consider two possible relief measures. First, we used multiple buffer pools, thus creating multiple buffer pool locks. Second, we introduced a more sophisticated scheme that reduced the number of buffer manager invocations.

5.1. Multiple buffer pools

Since Volcano was designed to support multiple pools, a simple change of a compile time constant was all that was required to explore the performance effects of multiple buffer pools. The policy function that assigned devices to buffer pools calculated the device number modulo the total number of buffer pools.

Figure 5 shows the times observed with 1 to 16 buffer pools. The effects of multiple

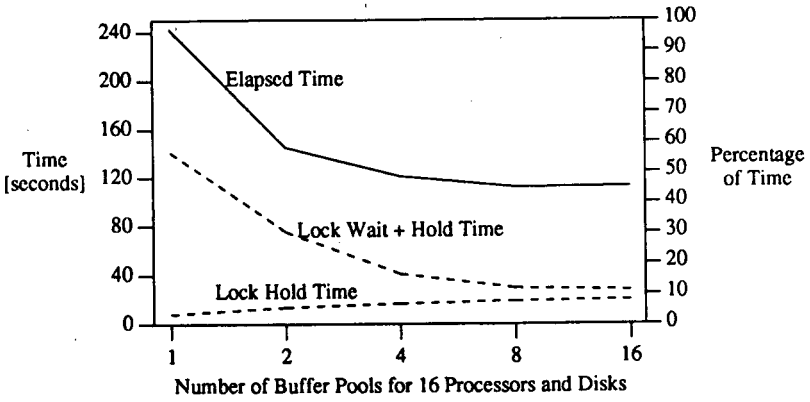

Figure 5. Multiple buffer pools

buffer pools are quite impressive, both in the decrease of elapsed time and of lock waiting time; it seemed that the 'log jam' had been broken. Both lock wait time and elapsed time were reduced substantially, while the lock holding time remained basically constant (the fraction of time during which the lock is held increases as the elapsed time decreases).

However, running a database system with multiple buffer pools, in particular as many as 16 or even more in larger systems, has a very undesirable side-effect. In general, processing loads are not spread as evenly as in our experiments, and the buffer contention may vary widely between buffer pools. In the extreme, some of the buffer pools might be thrashing while other ones retain stale data. Furthermore, as seen in the slight increase of elapsed time from 8 to 16 buffer pools in Figure 5, there is some of this effect for even loads. For a real system, it seems that a large number of buffer pools is not realistic. Therefore, we looked for an alternative solution.

5.2. Grouping buffer calls

Instead of *spreading* the load over multiple pools, we considered ways to *reduce* the load. Consider the sequence of buffer unfix operations that are performed, for example by the query driver in each process. If records of one cluster are processed sequentially, consecutive unfix operations pertain to the same cluster. For example, if N records from one cluster are passed from a query tree to the driver and the driver calls the buffer manager for each record, the buffer manager's fix count for the cluster is originally N and is decremented by one in each call. The last of the N calls reduces the fix count to zero, which allows the buffer manager to place the cluster on the free list and eventually replace it in the buffer pool. The first N -1 buffer manager calls have no effect on buffer contention or replacement options, and there is no harm in delaying them. Thus, we considered grouping all these N calls and calling the buffer manager only once with an additional parameter N. In the modified scheme, unfix operations referring to the same cluster are gathered and the buffer manager is invoked much less frequently, i.e., only when consecutive records do not belong to the same cluster.

It is important that these references be gathered in space private to the process, not in a shared memory segment, to ensure that no concurrency control is needed. In addition, the comparison function that determines whether or not two consecutive records belong to the same cluster must be faster than a hash table lookup in the buffer manager, which requires a hash calculation and possibly many comparisons.

Notice that grouping buffer manager calls has an inherent limitation. In the best case, all records from one file are passed to one operator, which unfixes all of them in the buffer. If some records are removed from a stream using some predicate, however, two operators perform unfixing operations, and the buffer manager is called twice for each cluster. Similarly, if a stream (and also the set of records from one cluster) is partitioned and forwarded to multiple consumer processes, the buffer manager is invoked once for each cluster by each consumer. However, this situation is still preferable over unfixing individual records as well as over copying data.

Volcano always used the idea of invoking the buffer manager once for a cluster's load of records when fixing records. In a file scan, for example, a cluster is fixed immediately after reading it from disk with the fix count equal to the number of

record slots. As records are passed from the file scan to the requesting iterator, a counter called ‘over-fixed’ is decremented in the scan descriptor. Obviously, it is easier to implement and exploit this idea in a stream-originating iterator such as file scan (where clusters and cluster boundaries can easily be made visible) than in a consumer iterator with its algorithmic control flow focussed on records. The unmodified version of the sort iterator, for example, fixed its output records one cluster at a time but unfixed its input one record at a time.

Gathering and unfixing records received by an iterator were implemented both by a procedure that performs the gathering function for all iterators (in space private to the iterator, of course) or by an equivalent macro because macro implementation of very frequently used functions such as lock requests is very common in commercial database systems.

In the case that input records originate from multiple sources, e.g., the output of a merge iterator, [14] the gathering idea will not work. To provide efficient support for this situation, we implemented procedure and macro in such a way that they can work with a small hash table. The buffer manager is invoked to unfix a hash table entry when a hash collisions occurs, and the old hash table entry is overwritten. While this alternative is more expensive than the single-entry version, it is not as expensive as invoking the buffer manager for each record.

Table II shows the times measured for five unfixing schemes. The ‘Total lock’ columns indicate the amount of time the buffer pool lock was held by some process; the ‘Total wait’ columns indicate the sum of all waiting times for the buffer pool lock by all processes. Note that these two times reflect the sum over all processes. For the group unfixing schemes, the size of the hash table is given. All times were measured with a single buffer pool. The first row shows results for the original, ‘simple’ scheme in which the buffer manager is invoked to unfix each record. When the simple scheme was replaced by the ‘group’ scheme, the performance improved dramatically. This improvement can be seen in the lock holding times (reflecting fewer buffer invocations) and, even more so, in the waiting times for the buffer pool lock. Whether the group scheme is implemented as a procedure or as a macro does not make much difference. Similarly, whether the records are gathered with a single entry or a small hash table had only very little effect in these experiments. The differences between the individual group schemes are minimal and easily explained. Macros are a little faster than procedures, and because the multiple entries do not provide any benefit in this particular experiment, they do not improve performance here. The performance is slightly less since multiple entries have a higher computation cost for the hash function. More importantly, perhaps, is that they may

Table II. Alternative record unfixing schemes

Unfix scheme	8 disks Time, s	8 disks Total lock, s	8 disks Total wait, s	16 disks Time, s	16 disks Total lock, s	16 disks Total wait, s
simple	258·31	120	452	242.62	130	2062
procedure [1]	168·00	7·260	0·738	93.69	9·725	3·168
macro [1]	166·10	7·418	0·687	90.34	9·690	3·168
macro [4]	170·78	7·241	0·732	91.07	9·337	2·955
macro [16]	171·76	7·485	0·642	91.57	9·985	3·219

increase buffer contention because clusters that should be completely unfixed remain in some group-unfix hash table, thus increasing both search time in the buffer tables and I/O of buffer pages.

The group unfixing schemes work, best with large clusters because large clusters contain more records. On the other hand, the effect of group unfixing is reduced if data are repartitioned between operations because each receiving process receives only some of the records of each input cluster. The higher number of buffer manager calls for 16 compared to 8 processors can be seen in the higher total lock time in Table II. In principle, if input records can come from multiple sources, it is unlikely that consecutive records are in the same cluster. However, since Volcano's exchange operator groups records into packets before transferring them to another process, some of the grouping effect is retained.

5.3. Grouping buffer calls and multiple buffer pools

Seeing the success of both of these techniques to reduce buffer contention, we considered combining them. Table III shows the result for multiple buffer pools using the macro[l] group-unfixing scheme. Two buffer pools showed some improvement over a single pool—in fact these were the first runs below 1½ minutes—but it is obvious that the bulk of the improvements came from the improved unfixing technique. In all runs reported below, the macro [1] group unfixing mechanism was combined with two buffer pools.

6. HARDWARE-ORIENTED ENHANCEMENTS

The Symmetry system has a built-in performance monitoring capability, which allows system designers and users to observe the performance of the system hardware for different applications. This tool has become very useful for monitoring contention for hardware resources, and for designing and evaluating software modifications. In this section, we describe some of the improvements considered as a result of observing the behavior of this application using the hardware monitor.

When we observed bus activity during our sort runs using a bus monitor, we found that the different phases in the sort and exchange code are reflected in the patterns of bus activity. Therefore, we tried to identify hot spots of bus activity and alleviate the problems by suitably modifyingVolcano's software.The contention for the buffer manager was first observed in the hardware, and we identified the buffer pool lock as a 'hot lock' in Volcano. As a result, the two enhancements reported above were explored.

Table III. Grouping buffer calls and multiple buffer pools

Buffer pools	8 disks			16 disks		
	Time, s	Total lock, s	Total wait, s	Time, s	Total lock, s	Total wait, s
1	166·10	7·418	0·687	90·34	9·690	3·000
2	169·03	7·410	0·233	89·96	9·651	1·091

6.1. Staggering buffer flushes

Using the bus monitor, we found a significant peak in bus contention at the beginning of buffer flushing. We suspected that this occurred when all processes had finished sorting and requested the buffer pool locks to gain access to the hash tables for buffer flushing. Recall that processes terminate at very much the same time, namely when the last process sends a packet with the end-of-stream tag.

The spin locks we used during these experiments are quite simple, using a single byte of shared memory. In the case of very high lock request traffic, the hardware negotiation for the exclusive ownership of this lock byte is relatively slow. In order to alleviate such lock contention, we artificially delayed processes when they entered the flush phase. Figure 6 shows the effect of such delays. Each process was delayed by the time shown in the table multiplied with its process number (0-15). Thus, the largest delay in the table is 15 × 4 ms = 60 ms.

Figure 6 shows that even minimal delays make a difference. In fact, we observed even smaller delays than shown in Figure 6 to make a difference, although not a systematic one. Longer delays did not increase the improvement, largely because bus contention was almost entirely removed once lock requests were staggered sufficiently to allow one process to acquire the lock before the next request on the bus. On the other hand, even the longest delays in Figure 6 are so small that they have hardly any effect on the elapsed times. In the following experiments, we used 2 ms delays.

6.2. Process-to-processor affinity

CPU caches were invented and are used in modern computer systems because they can result in substantial performance improvements if the fault rate can be k low. If processes are scheduled in their previous CPU after a preemption or an I/O, cache loading time (i.e., many cache faults) after rescheduling a process can be reduced or eliminated. In DYNIX, processes can be firmly bound to CPUs using the affinity system call, which had been used very effectively to improve transaction processing (OLTP) performance on our hardware.

Since affinity in DYNIX is a firm assignment, i.e., the process cannot run in any other CPU that the one it has an affinity to, it requires the application to perform load balancing among the processors in the system. In our situation where both problem and processing are very symmetrical, load balancing did not seem to be a

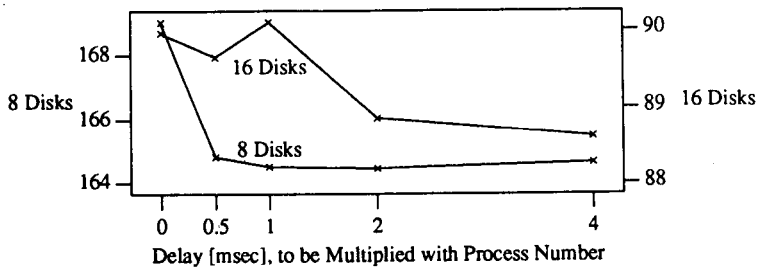

Figure 6. Staggering buffer flushes

Table IV. Processor affinity

Alignment	8 disks	16 disks
	Time, s	
Off	164·46	88·86
On	163·34	88·09

major problem. In general, however, we suggest making process 'sticky', i.e., augmenting the CPU scheduler with a heuristic to increase the probability that a process is assigned repeatedly to the same CPU and cache but at the same time permitting the operating system to assign a process to a different CPU if the processing loads become very unbalanced.

In Table IV, we show the performance of our sort program with and without affinity. As can be seen, affinity did not have a major impact on the elapsed sort times. We suspect, however, that the impact would have been larger if the number of processes had exceeded the number of processors, and most context switches had occurred due to time slicing rather than due to I/O as in our experiments. In the remaining experiments, affinity was enabled.

6.3. Revisiting record alignment

When looking for further improvements, we revisited the decision to use 16-byte alignment. After all, aligning 100-byte records to 16-byte boundaries requires 12 bytes of padding per record. In other words, all I/O operations had to transfer 12 per cent additional data bytes. Expressed differently, we hoped to save 12 per cent I/O by not padding 12 bytes to each 100-byte record, and expected that the savings would offset the loss in memory and cache access performance.

Table V shows that this is indeed the case. For all degrees of parallelism, 4-byte alignment significantly outperforms 16-byte alignment. While we did not realize 12 per cent, the average improvement was close to 5 per cent, indicating that fragmentation and cache alignment had competing effects on performance.

7. SUMMARY AND CONCLUSIONS

In this article, we have considered a number of performance improvements to the Volcano query execution software, and explored their effects on parallel sorting. In light of the original time for 16 processors and 16 disks above 5 minutes, a final

Table V. Final performance measurements

Alignment, bytes	2 disks, s	4 disks, s	8 disks, s	12 disks, s	16 disks, s
16	663·32	327·56	163·34	111·96	88·09
4	623·42	31O·8O	158·12	106·26	83·66

result of less than 1½ minutes represents a substantial improvement. The sorting throughput improved from 323 KB/s to over 1·1 MB/s. As each data item was moved from or to disk four times and sixteen disks were used, the disk bandwidth utilization was 4 × 100 MB/83·66 s/16/l·8 MB/s = 16·6 per cent, which is a satisfactory result considering that our cluster size did not coincide with the disks' track size and that ½ of all I/O (input and output during merging) was random I/O with many disk arm movements. Since each data item had to travel over the system bus at least seven times (two times from disk to memory to cache to disk plus process-to-process data exchange of almost all data), the bus bandwidth of 53 MB/s was used to 7 × 100 MB/83·66 s/53 MB/s = 15·8 per cent by actual and necessary data transfer, with additional bus usage for instruction transfer and process synchronization. Furthermore, our second goal of linear speedup was almost attained. To verify this, we calculated the parallel efficiency as (2 × 623·42 s)/(16 × 83·66 s) = 93·15 per cent.

The best performance for parallel sorting has recently been obtained by DeWitt *et al.* using the Gamma database machine.[30] Using 30 processors and disks and a hypercube interconnection network, they observed around 57 seconds clasped time for sorting 100 MB of data. The processors and disks used in Gamma and Volcano were very similar (Intel 80386 CPUs and SCSI disk drives). The main differences in the experiments were the interconnection technology (shared memory vs. hypercube), the number of processors and disks (16 vs. 30), and the assumption about the data distribution. While we assumed quantiles for data partitioning known *a priori,* DeWitt *et al.* investigated sampling techniques to determine appropriate quantiles from the data. The effect of these three differences makes the sorting throughput of Gamma and Volcano very similar.

Figure 7 shows the initial and final measurements. The time measurements are shown using solid lines and refer to the labels on the left. The speedups are shown with dashed lines and refer to the labels on the right. The initial times and speedups are marked with □ s while the final ones are marked with Δ s. The ideal speedup is also shown by the dotted line.

It is immediately obvious from the solid lines that the final times are significantly lower than the initial ones, demonstrating the effect of our tuning measures. For two to eight processors and disks, performance improved by a factor slightly more

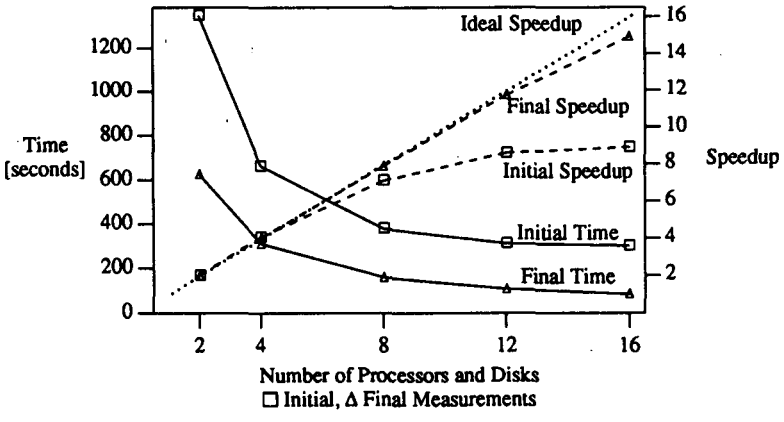

Figure 7. Initial and final sort performance

than 2, which is largely due to increased cluster size and reduced I/O cost. The effect did not show in our earlier comparison because this was the first improvement we attempted while still using the simple buffer unfixing scheme. Beyond eight processors and disks, the dashed lines indicate that the modifications and adjustments also improved the speedup which had been quite unsatisfactory with the initial software. For sixteen processors and disks, the fully tuned software performed about 3·6 times better than the original version. A comparison of the dashed and dotted lines shows very close to linear speedup with the fully tuned software. Thus, the tuning improved the parallel behavior as well as the absolute performance.

One of the options we did not explore in this study was *read-ahead* and *write-behind*. Since we used raw devices, the DYNIX operating system did not perform any buffering or read-ahead/write-behind in these experiments. Volcano does include an optional buffer daemon process for this purpose, but we did not use it for three reasons. First, its implementation is not fully tuned, in particular messages and synchronization between work processes and daemon processes, and we did not have time to tune it while we had access to the large machine. Second, additional space would have been committed [4] and we only wanted to use buffer space for the sort operation proper. Third, we wanted to use an equal number of processes and processors since earlier studies had indicated performance problems if processes migrate too much, and we felt that the space and process management issues were too complex to explore in the limited time available. Using a rough 'back-of-the-envelope' calculation, we estimate that carefully tuned read-ahead and write-behind would save about 20 of the 84 seconds, but we do not have experimental data to substantiate this estimation.

Beyond the lesson that performance tuning is a never-ending task (which we should have known !), we have drawn a number of conclusions from this effort. First, and most importantly, shared-memory architectures with limited degrees of parallelism are very well-suited for database query processing and do indeed allow linear speedup. We realize that there is a limit beyond which a distributed-memory architecture must be employed. However, our results show that shared-memory systems have a place in query processing just as in online transaction processing, [40] and that the limit to which shared-memory systems can be pushed depends not only hardware characteristics (e.g., CPU speed, bus speed, and cache size), but also on how carefully the software has been tuned. Because of faster and cheaper synchronization and communication in shared-memory machines, we believe that nodes in distributed-memory machines should be shared-memory multiprocessors in spite of the complexities of such hierarchical designs. As shown in the most recent extension of Volcano's exchange operator, these complexities can be hidden from the data manipulation operators even in distributed and hierarchical memory architectures. [16, 41]

Second, implementing parallel query processing engines by combining operators designed and implemented for sequential architectures with a generic parallelism operator is a sound, modular approach that does not prevent good performance and linear speedup. We hope that the similarly encouraging results can be obtained on distributed and hierarchical memory machines.

Third, it is very important that query processing modules be designed and implemented to allow future modifications, tuning, and performance enhancements. In other words, extensible database management systems that are intended to support

a variety of new database applications like EXODUS , [42] Postgres [43] and Starburst [20] should be designed to be extensible and modifiable not only in the data modelling level of the software architecture but also in their run-time system in order to permit not only representing a new application domain but also tuning the database system to the application's needs and to the hardware and operating system environment. For our tuning study, it was very useful that the Volcano software was designed and implemented to support a large number of choices, many of them as run-time parameters, e.g., the unit of I/O and buffer hint mechanisms. Volcano's design goal to provide *mechanisms* such that a query optimizer or a human experimenter could choose *policies* was met and proved extremely valuable. This encourages us to use Volcano as an experimental vehicle for further research, namely in query optimization [6,13,16] and query processing in scientific and object-oriented database systems [44,45]

Fourth, it seems to be a general concept that contention can be relieved by reducing the number of critical sections and lock requests, by maintaining multiple copies of resources protected by critical sections, and by making the critical sections perform faster. The last method can be achieved by tuning the code, i.e., shortening the instruction path length. Volcano's buffer code had been very carefully tuned over an extended period of time such that there was only limited leverage left in this direction. The other two methods, replicating critical resources to spread the load over more resources and reducing the number of resource requests, proved to be powerful tuning measures. Because multiple copies of a resource always introduce the danger of uneven load and therefore performance degradation, we focused on reducing the number of critical sections and lock requests. It turned out that this could be done to such an extent that spreading the load over many resources (buffer pools) had only limited additional effect, and we could safely limit the number of buffer pools to two. The particular technique employed, unfixing records in groups, was very effective for the problem at hand. It is a curious observation that the buffer pool contention could be relieved by 'buffering' buffer manager calls, i.e., by gathering multiple requests in process-private space and calling the shared buffer manager only when we experienced a 'buffer fault' as the unfix requests progressed from one cluster to the next.

We are currently extending this research in several directions. First, we undertook this study of the parallel sort algorithm as one example database query processing algorithm that requires CPU processing, memory, I/O, and data exchange. While we expect that most of the improvements identified here also apply to other algorithms, e.g., sort- and hash-based join and aggregation algorithms, we will have to verify it experimentally. Second, we will explore the effects of our tuning measures in different environments and situations, in particular when data need to be repartitioned multiple times between the operations in a complex query evaluation plan. Third, we are exploring detrimental effects of parallelism. For example, as the number of processes sharing memory increases, each process' space becomes smaller. For parallel sorting, this means that both the size of initial runs and the merge fan-in decrease, which may lead to overall performance degradation. Another well-known example is the need for finer load-balancing for higher degrees of parallelism. Finally, to put all these directions together, we are working on more comprehensive methods for resource distribution among competing operators in a complex query evaluation plan, both for single-process and parallel query evaluation plans.

ACKNOWLEDGEMENTS

We appreciate the anonymous reviewers' perceptive comments. The work on Volcano has been partially funded by NSF with grants IRI-8996270 and IRI-8912618, and by the Oregon Advanced Computing Institute (OACIS).

REFERENCES

1. D. J. DeWitt, S. Ghandeharadizeh, D. Schneider, A. Bricker, H. I. Hsiao and R. Rasmussen, 'The Gamma database machine project', *IEEE Trans. Knowledge and Data Eng.* **2,** (l), 44 (1990).
2. 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.
3. K. Salem and H. Garcia-Molina, 'Disk striping', *Proc. IEEE Conf. on Data Eng.,* Los Angeles, CA, February 1986, p. 336.
4. B. Salzberg, 'Merging sorted runs using large main memory', *Acta Informatica,* **27,** 195 (1990).
5. L. D. Shapiro, 'Join processing in database systems with large main memories', *ACM. Trans. Database Systems,* **11,** (3), 239 (1986).
6. G. Graefe and W. J. McKenna, 'The Volcano optimizer generator', submitted for publication, 1991. Also *CU Boulder Computer Science Technical Report 563.*
7. G. Graefe, 'Encapsulation of parallelism in the volcano query processing system', *Proc. ACM SIGMOD Conf.,* Atlantic City, NJ, May 1990, p. 102.
8. G. Graefe, 'Volcano, an extensible and parallel dataflow query processing system', to appear in *IEEE Trans. Knowledge and Data Eng., 1992.* A more detailed version is available as *CU Boulder Computer Science Technical Report 481,* July 1990.
9. G. Graefe, 'Relational division: four algorithms and their performance', *Proc. IEEE Conf. on Data Eng.,* Los Angeles, CA, February 1989, p. 94.
10. M. Stonebraker, 'The case for shared-nothing', *IEEE Database Eng.,* **9,** (1) (1986).
11. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith and P. Valduriez, 'Prototyping Bubba, a highly parallel database system', *IEEE Trans. Knowledge and Data Eng.,* **2,** (l), 4 (1990).
12. P. M. Neches, 'Hardware support for advanced data management systems', *IEEE Computer,* 17, (11), 29 (1984).
13. G. Graefe and K. Ward, 'Dynamic query evaluation plans', *Proc. ACM SIGMOD Conf.,* Portland, OR, May-June 1989, p. 358.
14. G. Graefe, 'Parallel external sorting in Volcano', submitted for publication, 1990. Also *CU Boulder Computer Science Technical Report 459.*
15. G. Graefe and D. J. DeWitt, 'The EXODUS optimizer generator', *Proc. ACM SIGMOD Conf.,* San Francisco, CA, May 1987, p. 160.
16. G. Graefe, 'Extensible query optimization and parallel execution in Volcano', submitted for publication, 1991. Also *CU Boulder Computer Science Technical Report 548.*
17. J. E. Richardson and M. J. Carey, 'Programming constructs for database system implementation in EXODUS', *Proc. ACM SIGMOD Conf.,* San Francisco, CA, May 1987, p. 208.
18. J. E. Richardson, 'E: a persistent systems implementation language', *Computer Sciences Technical Report 868,* University of Wisconsin—Madison, August 1989.
19. L. M. Haas, W. F. Cody, J. C. Freytag, G. Lapis, B. G. Lindsay, G. M. Lehman, K. Ono and H. Pirahesh, 'An Extensible processor for an extended relational query language', *Computer Science Research Report,* San Jose, CA, April 1988.
20. L. Haas, W. Chang, G. Lehman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey and E. Shekita', Starburst mid-flight: as the dust clears', *IEEE Trans. Knowledge and Data Eng.,* **2,** (l), 143 (1990).
21. M. Blasgen and K. Eswaran, 'Storage and access in relational databases', *IBM Systems Journal,* **16,** (4), (1977).
22. R. Epstein, 'Techniques for processing of aggregates in relational database systems', *UCB/Electronics Research Lab. Memorandum M79/8,* University of California, February 1979.
23. G. Graefe, A. Linville and L. D. Shapiro, 'Sort versus hash revisited', *CU Boulder Computer Science Technical Report 534,* July 1991, 1992.

24. P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie and T. G. Price, 'Access path selection in a relational database management system', *Proc. ACM SIGMOD Conf.,* Boston, MA, May-June 1979, p. 23.
25. D. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching,* Addison-Wesley, Reading, MA, 1973.
26. M. Beck, D. Bitton and W. K. Wilkinson, 'Sorting large files on a backend multiprocessor', *IEEE Trans. Computers,* **37,** 769 (1988).
27. D. Bitton, D. J. DeWitt, D. K. Hsiao and J. Menon, 'A taxonomy of parallel sorting', *ACM Computing Surveys,* **16,** (3) 287 (1984).
28. D. Bitton Friedland, 'Design, analysis, and implementation of parallel external sorting algorithms', *Computer Sciences Technical Report 464,* University of Wisconsin—Madison, January 1982.
29. J. Menon, 'A study of sort algorithms for multiprocessor database machines', *Proc. Int. Conf. on Very Large Data Bases,* Kyoto, Japan, August 1986, p. 197.
30. D. DeWitt, J. Naughton and D. Schneider, 'Parallel sorting on a shared-nothing architecture using probabilistic splitting', *Proc. Int'l. Conf. on Parallel and Distributed Information Systems,* Miami Beach, FL, December 1991.
31. B. R. Iyer and D. M. Dias, 'System issues in parallel sorting for database systems', *Proc. IEEE Conf. on Data Eng.,* Los Angeles, CA, February 1990, p. 246.
32. R. A. Lorie and H. C. Young, 'A low communication sort algorithm for a parallel database machine', *Proc. Int. Conf. on Very Large Data Bases,* Amsterdam, The Netherlands, 1989, p. 125.
33. B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren and B. Vaughan, 'FastSort: an distributed single-input single-output external sort', *Proc. ACM SIGMOD Conf.,* Atlantic City, NJ, May 1990, p. 94.
34. S. Seshadri and J. F. Naughton, 'Sampling issues in parallel database systems', *Proc. Int Conf. Extending Database Technology,* Vienna, Austria, March 1992.
35. T. Lovett and S. S. Thakkar, 'The symmetry multiprocessor system', *Proc. Int. Conf. on Parallel Processing,* August 1988.
36. T. E. Anderson, 'The performance of spin-waiting alternatives for shared memory processors', *Computer Science Technical Report 89-04-03,* April 1989.
37. 'A measure of transaction processing power', *Datamation,* 1 April, 1985, p. 112.
38. W. Effelsberg and T. Haerder, 'Principles of database buffer management', *ACM Trans. on Database Systems,* **9,** (4), 560 (1984).
39. G. M. Sacco and M. Schkolnik, 'A mechanism for managing the buffer pool in a relational database system using the hot set model', *Proc. Int. Conf. on Very Large Data Bases,* Mexico City, Mexico, September 1982, p. 257.
40. A. Bhide and M. Stonebraker, 'A performance comparison of two architectures for fast transaction processing', *Proc. IEEE Conf. on Data Eng.,* Los Angeles, CA, February 1988, p. 536.
41. G. Graefe and D. L. Davison, 'Encapsulation of parallelism and architecture-independence in extensible database query processing', submitted for publication, 1991. Also *CU Boulder Computer Science Technical Report 559.*
42. M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Shuh, E. J. Shekita and S. Vandenberg, 'The EXODUS extensible DBMS project: an overview', in D. Maier and S. Zdonik (ed.), *Readings on Object-Oriented Database Systems,* Morgan Kaufman, San Mateo, CA, 1990.
43. M. Stonebraker, L. A. Rowe and M. Hirohama, 'The implementation of postgres', *IEEE Trans. Knowledge and Data Eng.* **2,** (l), 125 (1990).
44. S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt and B. Vance, 'Query optimization in revelation, an overview', *IEEE Database* Eng. **14,** (2), (1991).
45. R. Wolniewicz and G. Graefe, 'Automatic optimization and parallelization in scientific databases', in preparation, 1991.