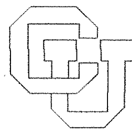
Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor

Goetz Graefe
Shreekant S. Thakkar

CU-CS-470-90



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor

Goetz Graefe and Shreekant S. Thakkar

CU-CS-470-90 April 1990

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

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor

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

Shreekant S. Thakkar
Sequent Computer Systems
Beaverton, OR 97006
ticky%sequent.uucp@cse.ogi.edu

Abstract

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 be 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 we have built for research and education. Its primary design goals are extensibility and performance. 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 report, we present Volcano's parallel external sorting algorithm and a sequence of enhancements to improve its performance. We observed the best external sort performance reported to-date 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.

Keywords: Database Systems, External Sorting, Parallel Processing, Performance, Shared-Memory, Linear Speed-up, TP-1 Sort, Tuning.

1. Introduction

Several research and development projects over the last decade have shown that query processing in relational database systems can benefit significantly from parallel algorithms [1,4,5,8,12-15,17,19,27,33,34,38,42,44,46,52-55,57,58]. The main reasons why parallelism is relatively easy to exploit in relational 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 attributes, 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 [47], and usually benefit from large memories [48, 51]. For best performance, it is imperative to balance not only CPU and IPC performance but also I/O bandwidth and memory allocation.

We believe that sorting is an excellent example for a parallel database 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 but 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 increases with the *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.

In order to provide an experimental testbed for database systems research and education, we implemented a query processing tool-kit called Volcano which allows several forms of parallel execution in any combination [27, 28]. Volcano's design goals are extensibility, modularity, performance, effective use of parallelism, and versatility as experimental platform. To support the last goal, we focused on *mechanisms*

to support *policies* chosen by a human experimenter or a query optimizer. Volcano includes all conventionally used query processing algorithms in relational database systems as well as several new ones [22,25,26,30,36], 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, sequential performance and parallelism issues.

In this report, we present Volcano's parallel sorting algorithm and a sequence of performance enhancements that we used to tune it on our machine. This study was initiated because an earlier investigation showed less than linear speedups for as little as eight processors [24], and we wanted to determine whether this is the indeed the maximal degree of useful parallelism in a shared-memory machine. In the earlier study, the best performance we observed was 195 seconds for 8 processors and disks for a 100 MB file, with a marked "knee" in the speedup curve. However, 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. In other words, the tuning efforts resulted in both improved throughput per processor and disk[1] and improved speed-up parallelism. We believe that the techniques explored in this report 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 [22,23,25-28,30,36], and is provided here to show Volcano's similarity to the query execution mechanisms of existing database systems. It is this similarity that makes us believe that the experiences

---

[1] To verify this statement, calculate $16 \times 84 = 1344 < 8 \times 195 = 1560$.

and conclusions reported here are applicable to other systems as well.

At this point, Volcano is a query processing 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. Later on, we will also provide a query optimizer that embeds and determines *policies*. Building on earlier experience [20], we are currently designing a new optimizer generator that will become part of Volcano.

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. They are pointers to a 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. 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, and have been used recently in the E database implementation language [,45] and the algebraic query evaluation system of the Starburst extensible relational database system [31,32].

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), 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.

There are two significant benefits in using anonymous inputs. First, we can use a generic driver module for all queries. The driver module is part of Volcano; it consists of a call to its input's *open* procedure, a loop calling *next* until it fails, unfixing the produced records in the buffer, and an invocation of *close*. Second, we could build a "parallelization" module that interfaces with all other operators and encapsulates all parallelism issues [28]. 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. 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.

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 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 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 [50]. First, a user request may state explicitly that the result data be returned in a particular sort order. Second, several query processing algorithms, in particular for join and aggregation, require sorted input data [11, 18].

External sorting is known to be an expensive operation, and many sorting algorithms have been devised [37]. Recently, parallel external sorting has gained substantial interest [6, 10, 35, 39, 41, 49, 56]. We have described our algorithm in detail in [30], so we only provide an overview here.

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. All such algorithms are variants of the well-known merge-sort technique and require a final centralized merge step [6, 9, 10, 41]. In a highly parallel architecture, any centralized component that has to process all data is bound to be a severe bottleneck.

Volcano's preferred sort method starts by exchanging data based on logical keys [30]. Notice that, 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 those sites which 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% or 0.5 quantile[2]. For load balancing among N processors, the i/N quantiles for i=1,...,N-1 need to be determined, e.g., by sampling [35, 39, 43].

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. Actually, the data exchange and the sort phase overlap naturally due to the iterator behavior of the algorithms.

3. Environment, Workload, and Initial Performance

3.1. The Sequent Symmetry Multiprocessor System

Sequent's Symmetry series is a bus-based shared-memory multiprocessor [40]. A machine can contain from two to thirty CPUs with a performance of around 4 MIPS per CPU. Each processor subsystem contains a 32-bit microprocessor, a floating point unit, optional floating point accelerator, and a private cache. The system features a 53 MB/sec 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 1.8 MB/sec. 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 and Weitek 1167 processor sub-system. was used for this study. Each CPU had a private 64 KB local cache supporting the Symmetry copyback cache coherence protocol. The cache coherence protocol is based on the concept of ownership. That is, to perform a write operation, a cache has to first perform 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 held the block in *modified* state, it has to respond to the read exclusive request and invalidate it's copy.

The synchronization mechanism on the Symmetry model uses cache-based locks. The locks are also ownership based. That is, a locked read from a processor is treated like a write operation by the cache con-

[2] Notice that 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.

troller. 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 system where locks are lightly contended and the critical sections are short. They do not work very well in some parallel applications where locks are heavily contended, and several other software synchronization schemes can be used to reduce contention for the locks in the hardware [2].

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 in [3] because it allows measuring a system's internal performance independent of the application and 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. Range partitioned means that the total key domain is split into disjoint subsets and that 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, 15% are used by the operating system for code, internal tables, and file system I/O buffer space.

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 shared memory, so it was allocated only once independent 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, we believe that 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.

Table 1 shows the initial measurements taken with our system. The performance does indeed improve as additional resources are committed to the problem. The performance improvement from 2 to 8 disks is good, reasonably close to linear, although not as close as we had hoped. The parallel efficiency can be calculated as $(2\times1346)/(8\times379) = 88.78\%$. However, there seems to be a barrier of 300 seconds (5 minutes) which this software or hardware cannot break. Since this is rather undesirable, we considered and measured several modifications.

4. Data Access Improvements

In this section, we report on modifications to improve data access performance. For disk access, the first thought is always clustering pages of a file to eliminate seeking 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 disk space. Another method to improve disk bandwidth is to increase the unit of I/O. Effectively, more data is transferred with each operation, and total I/O initialization, seek, and latency costs are reduced.

For memory data access, we observed that packing records densely on pages might not align records in an optimal way. While the hardware can deal with unaligned accesses, such accesses are known to

Disks	Time [sec]
2	1346
4	662
8	379
12	313
16	302

Table 1. Initial Measurements.

cause performance degradation.

In order to avoid I/O operations, it may be beneficial to increase the buffer size, in the hope that 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 first of changing the I/O cluster size, next of aligning records (and comparison keys), and finally 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. In fact, all requests to the buffer manager are in terms of clusters. When a new cluster is requested, the buffer manager removes clusters from the bottom of its LRU[3] list until the 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.

Table 2 shows the effects of increasing the cluster size. When compared to the initial measurements, there is 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, there seems to be a similar limit at about 240 seconds (4 minutes). Increased cluster sizes, while helpful, do not provide a breakthrough in performance nor do 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.

Cluster Size [KB]	8 disks Time [sec]	16 disks Time [sec]
4	379	302
8	302	262
16	266.93	251.60
32	253.19	251.97
64	249.75	248.26

Table 2. Increased Cluster Sizes.

---

[3] *L*east-*R*ecently-*U*sed is a simple and widely used (though not always optimal) buffer replacement algorithm; see [16] 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 similar to IBM's Starburst research project [32].

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. We tried three alignment boundaries. Single-byte means no alignment, 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 are aligned according to cache lines.

Unfortunately, the results in Table 3 do not show appreciable performance improvements. Since we intended to study the cache performance later on, records are 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 allocated for sorting. Therefore, we experimented with smaller and large buffer sizes. In Table 4, 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 %. The explanation can be found in two considerations. First, this increase in buffer size does not reduce the number of merge levels. With 8 MB or more of sort space, only a single merge level is required. To avoid merging altogether,

Record Alignment [bytes]	8 disks Time [sec]	16 disks Time [sec]
1	253.19	251.97
4	251.90	244.28
16	258.31	242.62

Table 3. Record Alignment.

Sort Buffer [Mbytes]	8 disks Time [sec]	16 disks Time [sec]
12, 3, 15	258.31	242.62
24, 6, 30	261.64	247.25

Table 4. Increased Buffer Size.

100 MB (the file size) of sort buffer space would be required. 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 buffer 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 might be the source of the problem. The buffer manager is invoked each time a record is unfixed, i.e., when an operator is done with an input record but does not pass it to the next operator. In our implementation, the buffer manager is called 3,000,000 times for this purpose: Once for each record by the *filter* operator that collects records of one partition, once by the sort operator whenever it has 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 whenever the buffer pool is locked or a process waits to acquire the buffer pool lock.

Table 5 shows some statistics about buffer lock utilization and contention. Note that the times shown reflect all processes. In the case of 2 processes, the total wait time was very moderate compared to 1346×2=2692 seconds. However, as the degree of parallelism increased, the wait times increased dramatically, and a relief had to be found.

Disks	Time [sec]	Total Lock [sec]	Total Wait [sec]
2	1346	148	21.5
4	662	153	116
8	379	167	428
12	313	169	1047
16	302	179	1964

Table 5. Initial Measurements, including Buffer Statistics.

The table also explains the performance barrier observed earlier. Since all buffer operations combined require the pool lock between 150 and 180 seconds, no amount of parallelism will improve the performance beyond this elapsed time.

In this section, we consider two possible relief measures. First, we used multiple buffer pools which spread the load and lock contention among themselves. Second, we introduced a slightly more sophisticated scheme that reduced the number of buffer manager invocations to a fraction.

5.1. Multiple Buffer Pools

In this section, we consider multiple buffer pools. Since Volcano was designed to support multiple pools and dynamic creation of buffer pools [23], a simple change of a compile time constant was all that was required. The policy function that assigned devices to pools calculates the device number modulo the number of buffer pools.

Table 6 shows the times observed with 1 to 16 buffer pools. The effects of multiple buffer pools are quite impressive; it seemed that the "log jam" had been broken. However, running a database system with multiple buffer pools, in particular as many as 16 or even more in larger systems, has an undesirable side effect. If the processing load is not spread as evenly as in this example, the buffer contention might vary widely between buffer pools. In effect, some of the buffers might be thrashing while others retain stale data. For a real system, it seems that such a large number of buffer pools is not realistic. Therefore, we attempted to find an alternative solution.

Buffer Pools	8 disks			16 disks		
	Time [sec]	Total Lock [sec]	Total Wait [sec]	Time [sec]	Total Lock [sec]	Total Wait [sec]
1	258.31	120	452	242.62	130	2062
2	216.76	121	117	145.35	129	574
4	209.56	120	53	121.16	130	190
8	200.16	125	13	112.24	137	76
16				113.13	145	59

Table 6. Multiple Buffer Pools.

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, 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. 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 once with an additional parameter N. In our modified scheme, unfix operations referring to the same cluster are gathered, and the buffer manager is invoked much less frequently, 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. Thus, no concurrency control is needed. In addition, the comparison function is much faster than a hash table lookup which requires a hash calculation and possibly many comparisons.

Volcano uses this idea already when fixing records. In a file scan, for example, the cluster is fixed immediately when read as often as it contains 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 the producing iterator where clusters and cluster boundaries are visible. The unmodified version of *sort*, for example, fixes its output records a cluster at-a-time but unfixes its input a record at-a-time.

For unfixing records passed between query processing iterators, we implemented two alternative gathering methods. First, we used a procedure that performs the gathering function for all iterators (in space private to the iterator, of course). Alternatively, this procedure is also implemented as a macro. This avoids the procedure call overhead at the expense of additional code space.

In the case that input records originate from two alternative sources, e.g., the output of a *merge* iterator described in [30], the gathering idea will not work. To provide efficient support for this situation, we

implemented the procedure and macro in such a way that it can work with a little hash table. In the case of collisions, the buffer manager is invoked to unfix the old hash table entry, and the entry is overwritten. While this alternative is more costly than the single-entry version, it is not as expensive as calling the buffer manager for each record.

Table 7 shows the times measured for five unfixing schemes. For the group unfixing schemes, the size of the hash table is also given. All times were measured with a single buffer pool. The first row shows results for the "simple" scheme in which the buffer manager is invoked for each record. When the simple scheme is replaced by the "group" scheme, the performance improves dramatically. It does not make much difference whether the group scheme is implemented as a procedure or as a macro. Similarly, whether the records are gathered with a single entry or a small hash table has only very little effect. 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 our situation, they do not improve performance. The performance is slightly less since multiple entries have a higher computation cost for the hash function. More importantly, they may increase buffer contention because cluster that should be completely unfixed remain is some group-unfix hash table.

We would like to point out that the group unfixing schemes work best with large clusters because large clusters contain more records. Furthermore, 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. 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 [28], some of the effect will be retained.

Unfix Scheme	8 disks			16 disks		
	Time [sec]	Total Lock [sec]	Total Wait [sec]	Time [sec]	Total Lock [sec]	Total Wait [sec]
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

Table 7. Alternative Record Unfixing Schemes.

5.3. Grouping Buffer Calls and Multiple Buffer Pools

Seeing the success of both of these techniques to reduce buffer contention, we were tempted to combine them. Fortunately, this was easily done. Table 8 shows the result for multiple buffer pools using the macro[1] group-unfixing scheme. There are some improvement over a single pool, in fact these were the first runs below 1½ minutes, but from the last three tables it is obvious that the bulk of improvements comes from the improved unfixing technique. In all runs reported below, the macro [1] group unfixing mechanism is used combined with two buffer pools.

6. Hardware-Oriented Enhancements

The Symmetry system has a built-in performance monitoring capability. This allows the system designers and users to dynamically 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 done 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 observed 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 modifying Volcano'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.

Buffer	8 disks			16 disks		
Pools	Time [sec]	Total Lock [sec]	Total Wait [sec]	Time [sec]	Total Lock [sec]	Total Wait [sec]
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
4	165.42			89.56		
8	164.97			90.56		
16	165.77			90.09		

Table 8. Grouping Buffer Calls and Multiple Buffer Pools.

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 occurs when all processes request the buffer pool locks at the same time to gain access to the hash tables. Recall that processes terminate their subtrees very much at 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, other, seemingly more complex schemes can work more efficiently with the Sequent copy-back cache protocol [2].

In order to alleviate such lock contention, we artificially delayed processes when they entered the flush phase. Table 9 shows the effect of such delays. Each process was delayed by the time shown in the table multiplied with its process number within its group. Thus, the largest delay in the table is $15 \times 4ms = 60ms$.

The table shows that even minimal delays make a difference. Larger delays do not increase the improvement, largely because the bus contention is almost entirely removed once lock requests are staggered sufficiently to allow one process to acquire the lock before the next request on the bus. In the following experiments, we used $2ms$ delays.

6.2. Process-to-Processor Affinity

Significant performance improvements can be realized if cache faults can be reduced. There are two benefits from a hardware perspective. First, it reduces demand for the bus, and second, it reduces cache-

Flush Delay [msec]	8 disks Time [sec]	16 disks Time [sec]
0	169.03	89.96
0.01	165.04	88.86
0.1	165.23	88.87
0.5	164.83	89.65
1	164.52	90.10
2	164.46	88.86
4	164.65	88.62

Table 9. Staggering Buffer Flushes.

to-cache operations. The second reduction is important because the latency of cache-to-cache operations is longer than memory-to-cache operations in our system since the caches, unlike memory, have to serve both their local CPU and all requests coming over the bus while memory only reponds to bus traffic.

One method to reduce the number of cache faults is to ensure that processes are scheduled in their last CPU after a preemption or an I/O. Thus, cache loading time after rescheduling a process should be reduced or eliminated. In DYNIX, processes can be bound to CPUs using the *affinity* system call. Notice that affinity puts the burden on load balancing on the application. In our case, a very symmetric problem tackled with symmetric processes on a dedicated machine, this did not seem to be a major problem.

In Table 10, we compare the performance of our sort program with and without affinity. As can be seen, affinity does not have a major impact on the elapsed sort time. We suspect, however, that the impact would have been larger if the number of processes had exceeded the number of processors. 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% additional data bytes. Expressed differently, we hoped to save 12% 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.

Affinity	2 Disks [sec]	4 Disks [sec]	8 Disks [sec]	12 Disks [sec]	16 Disks [sec]
Off			164.46		88.86
On	663.32	327.56	163.34	111.96	88.09

Table 10. Processor Affinity.

Alignment [bytes]	2 Disks [sec]	4 Disks [sec]	8 Disks [sec]	12 Disks [sec]	16 Disks [sec]
16	663.32	327.56	163.34	111.96	88.09
4	623.42	310.80	158.12	106.26	83.66

Table 11. Final Performance Measurements.

Table 11 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%, the average improvement was about 5%, indicating that fragmentation and cache alignment had competing effects on performance.

7. Summary and Conclusions

In this report, we have detailed several performance improvements to Volcano's query processing software, explored and demonstrated with parallel sorting. Considering that the original time for 16 processors and 16 disks was above 5 minutes, a final result of less than 1½ minutes represents a substantial improvement. Furthermore, our second goal of linear speedup was almost attained. To verify this, we calculated the parallel efficiency as $(2\times623.42)/(16\times83.66) = 93.15\%$.

Figure 1 shows the initial and final measurements in graphical form. 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

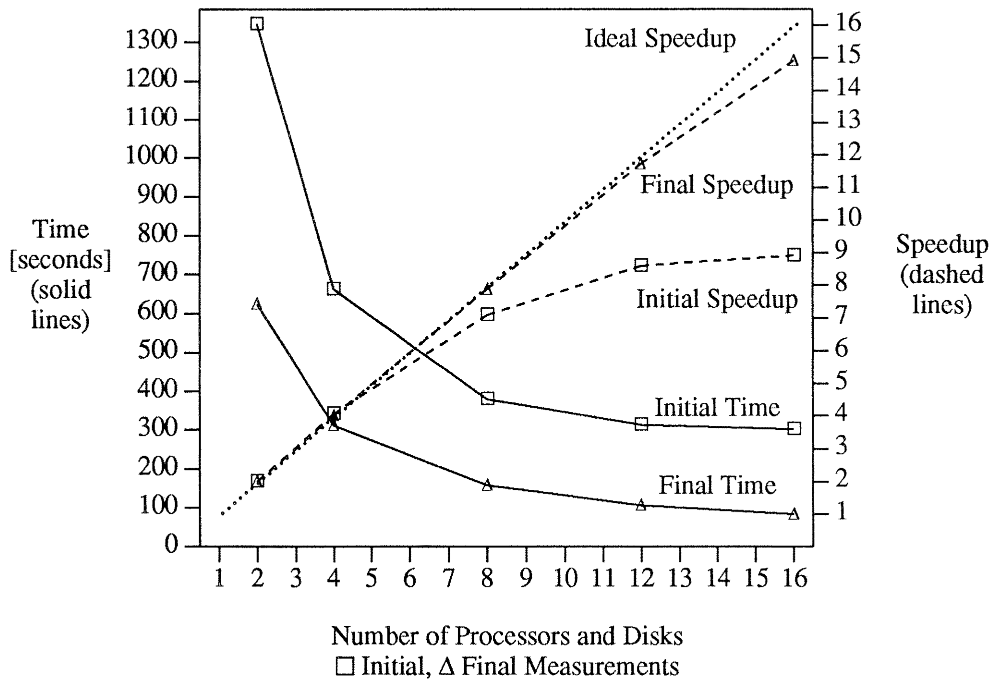


Figure 1. Initial and Final Sort Performance.

tuning measures. For two to eight processors and disks, we observed performance improvements by a factor slightly more than two 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 completely 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, DYNIX 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 [48] 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 back this up.

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, shared-memory architectures with limited degrees of parallelism are very well suited for database query processing and allow linear speedup. We realize that there is a limit beyond which a distributed-memory architecture must be employed. However, our results demonstrate that shared-memory systems have their place in query processing just as in online transaction processing [7], 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. Furthermore, we postulate that nodes in distributed-memory machines should be

shared-memory multiprocessors in spite of the complexities of such hierarchical designs. Furthermore, we suspect such designs will make shared-disk systems obsolete since shared memory implies shared disks.

Second, it is important that code be designed and implemented to allow for modifications. In other words, it was very useful that Volcano includes so many choices, many of them as run-time parameters. 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 query processing vehicle in further research, namely in query optimization [22], query processing in object-oriented database systems [21, 29], and clustering in object-oriented data models.

Third, it seems to be a general concept that contention can be addressed 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.

Multiple copies of a resource always introduce the danger of uneven load and therefore performance degradation. Hence, 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. Finally, to put all these directions together, we are working on more general 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 support we received from our colleagues at Sequent. The work on Volcano is supported by NSF through grants IRI-8996270 and IRI-8912618, and by the Oregon Advanced Computing Institute (OACIS).

References

1. W. Alexander and G. Copeland, "Process and Dataflow Control in Distributed Data-Intensive Systems", *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 90-98.
2. T. E. Anderson, "The Performance of Spin-Waiting Alternatives For Shared Memory Processors", *Computer Science Technical Report 89-04-03*, April 1989.
3. Anon. et al., "A Measure of Transaction Processing Power", *Datamation*, April 1, 1985, 112-118.
4. C. K. Baru, O. Frieder, D. Kandlur and M. Segal, "Join on a Cube: Analysis, Simulation, and Implementation", *Proceedings of the 5th International Workshop on Database Machines*, 1987.
5. R. Bayer, H. Heller and A. Reiser, "Parallelism and Recovery in Database Systems", *ACM Transactions on Database Systems 5*, 2 (June 1980).
6. M. Beck, D. Bitton and W. K. Wilkinson, "Sorting Large Files on a Backend Multiprocessor", *IEEE Transactions on Computers 37* (1988), 769-778.
7. A. Bhide and M. Stonebraker, "A Performance Comparison of Two Architectures for Fast Transaction Processing", *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1988, 536-545.
8. 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.
9. D. Bitton, D. J. DeWitt, D. K. Hsiao and J. Menon, "A Taxonomy of Parallel Sorting", *ACM Computing Surveys 16*, 3 (September 1984), 287-318.
10. D. Bitton Friedland, "Design, Analysis, and Implementation of Parallel External Sorting Algorithms", *Computer Sciences Technical Report 464* (January 1982), University of Wisconsin — Madison.
11. M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases", *IBM Systems Journal 16*, 4 (1977).
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. Ghandeharizadeh and D. Schneider, ''A Performance Analysis of the GAMMA Database Machine'', *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 350-360.

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

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

16. W. Effelsberg and T. Haerder, ''Principles of Database Buffer Management'', *ACM Transactions on Database Systems* 9, 4 (December 1984), 560-595.

17. 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).

18. R. Epstein, ''Techniques for Processing of Aggregates in Relational Database Systems'', *UCB/Electronics Research Lab. Memorandum*, Berkeley, February 1979.

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

20. G. Graefe and D. J. DeWitt, ''The EXODUS Optimizer Generator'', *Proceedings of the ACM SIGMOD Conference*, San Francisco, CA., May 1987, 160-171.

21. 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.

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

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

24. G. Graefe, ''Parallel External Sorting in Volcano'', *Oregon Graduate Center, Computer Science Technical Report*, Beaverton, OR., June 1989.

25. 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.

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

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

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

29. G. Graefe, D. Maier, S. Daniels and T. Keller, ''A Software Architecture for Efficient Query Processing in Object-Oriented Database Systems with Encapsulated Behavior'', *submitted for publication*, April 1990.

30. G. Graefe, ''Parallel External Sorting in Volcano'', *submitted for publication*, February 1990.

31. 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.

32. L. Haas, W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey and E. Shekita, ''Starburst Mid-Flight: As the Dust Clears'', *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (March 1990), 143-160.

33. T. Haerder, H. Schoning and A. Sikeler, "Parallel Query Evaluation: A New Approach To Complex Object Processing", *IEEE Database Engineering 12*, 1 (March 1989), 58-64.

34. H. Itoh, M. Abe, C. Sakama and Y. Mitomo, "Parallel Control Techniques for Dedicated Relational Database Engines", *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1987, 208.

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

36. 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.

37. D. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.

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

39. R. A. Lorie and H. C. Young, "A Low Communication Sort Algorithm for a Parallel Database Machine", *IBM Research Report 6669* (February 1989).

40. T. Lovett and S. S. Thakkar, "The Symmetry Multiprocessor System", *Proceedings ICCP 1988*, 1988.

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

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

43. G. Piatetsky-Shapiro and C. Connell, "Accurate Estimation of the Number of Tuples Satisfying a Condition", *Proceedings of the ACM SIGMOD Conference*, Boston, MA., June 1984, 256-276.

44. H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu and P. Selinger, "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches", *IBM Research Report*, San Jose, CA, March 1990.

45. 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.

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

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

48. B. Salzberg, "Merging Sorted Runs Using Large Main Memory", *Acta Informatica 27* (1990), 195-215, Springer International.

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

50. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database Management System", *Proceedings of the ACM SIGMOD Conference*, Boston, MA., May-June 1979, 23-34.

51. L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems 11*, 3 (September 1986), 239-264.

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

53. 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.

54. S. Y. W. Su and K. P. Mikkilineni, ''Parallel Algorithms and Their Implementation in MICRONET'', *Proceeding of the Conference on Very Large Data Bases*, Mexico City, Mexico, September 1982, 310-324.

55. S. Y. W. Su, K. P. Mikkilineni, R. A. Liuzzi and Y. C. Chow, ''A Distributed Query Processing Strategy Using Decomposition, Pipelining, and Intermediate Result Sharing Techniques'', *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1986, 94.

56. A. Tsukerman, J. Gray, M. Stewart, S. Uren and B. Vaughan, ''FastSort: An External Sort Using Parallel Processing'', *Tandem Technical Report 86.3* (1986).

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

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