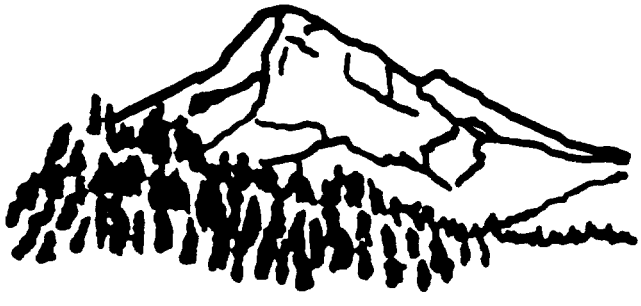
Parallel External Sorting in Volcano

Goetz Graefe

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

Technical Report No. CS/E 89-008
June, 1989



ogc

OREGON GRADUATE CENTER

19600 N.W. VON NEUMANN DRIVE
BEAVERTON, OREGON 97006-1999

Parallel External Sorting in Volcano

Goetz Graefe

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

Abstract

We add yet another paper on parallel sorting to the large body of literature on the topic. We briefly survey a new query evaluation system called Volcano developed for database systems research and education, and then focus on Volcano's sort algorithms. Volcano's single-process sort algorithm has several interesting features that make it quite efficient. Volcano's flexible multi-processing architecture provides efficient mechanisms for single- and multi-input and for single- and multi-output sort operations, in any combination. We report experimental performance results sorting medium size and large files on a shared-memory machine.

1. Introduction

A large amount of research effort has been spent on parallel sorting. Instead of reporting on yet another theoretical twist, we report here on a set of implemented parallel sort algorithms and analyze their performance. We implemented these algorithms within Volcano, a new modular, high-performance query evaluation system. Volcano is operational on both single- and multi-processor systems. It is not a complete database system since it lacks a number of features such as a query language, an optimizer, a type system for instances (record definitions), and catalogs. This is by design; Volcano is intended to provide an experimental vehicle for our earlier work in query optimization [1, 2, 3] and for multi-processor query evaluation.

In the following section, we briefly review previous work and propose our taxonomy of parallel external sort algorithms. In Section 3, we provide an overview of Volcano. Parallel processing is encapsulated in the *exchange* module introduced in Section 4. Section 5 describes the single-process sort operation which is the basis for all parallel sort algorithms. Section 6 describes alternative strategies and implementations of parallel sorting for very large files. In Section 7, we present and analyze experimental performance results for single-process and parallel sorting in Volcano. Section 8 contains a summary and our conclusions from this effort.

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

2. Related Work

Sort algorithms have been studied very extensively, both single-site and parallel algorithms. The best source for single-site algorithms probably still is [4]. For a survey of parallel algorithms, we refer the reader to [5]. We know of only very few recent investigations and implementations of parallel sort algorithms for large files, namely [6, 7, 8], the sorting method used in the Teradata database machine, and a Sequent-internal parallel sorting project.

We will attempt a taxonomy of parallel external sort algorithms. By external sort algorithm we mean a sort algorithm that utilizes secondary storage for intermediate files, and is therefore able to sort files much larger than main memory. Almost all combinations of design decisions outlined below have been proposed or tried, with disks or tapes as secondary storage, with or without hardware support, and with or without cascading individual (merge) steps. We consider these latter variations parameters rather than algorithm properties.

The first two determinants are whether the input is presented as a single file or as multiple files, and whether the output is desired as a single file or as multiple files. All four combinations have practical uses. Single-input single-output is the classical case with obvious use. Tandem's FastSort is of this kind [7]. Multiple-input single-output is useful to present a large file striped over multiple disks [9] to a user or single-thread application program. Teradata's algorithm is of this kind, using hardware support for merging multiple locally sorted streams. Single-input multiple-output has probably the fewest applications. One of those might be distributing a single-stream query output over multiple disks or sites and building separate indexes. With multiple output we typically require range-partitioning, i.e., that disjoint key ranges be assigned to the sites. A range-partitioned file can be viewed as one sorted file, and can be read or processed just as efficiently. Multiple-input multiple-output is the most general case. Using a randomly partitioned or striped input file [9], the goal is to create a range-partitioned output file. We will report on multiple-input single-output and multiple-input multiple-output algorithms in the section on experimental results.

The next determinant is how often each data item migrates between sites, i.e., the number of data exchange steps. In our algorithms, we wish to avoid transferring a data item between processing nodes more than once. The reason is that we are interested in *scalable* algorithms, i.e., algorithms that perform well for

high degrees of parallelism. In a shared-memory query processing system like the one we are using currently, a common system bus is bound to become a bottleneck as more processors are added. Therefore, an interconnection network must be introduced, in which data transfer can be a significant cost. We will not investigate parallel sort algorithms with more than one data exchange step.

Another determinant is the question of record vs. key sorts. In the former, the entire records are moved around, written to intermediate files, etc. In the latter, the sort key of each record is extracted and associated with its record identifier. The actual sort operation considers only the *key−RID* pairs and can be performed with less or even no I/O. A final phase retrieves records in the sorted order using RID's. A variant of key sorting is used in [8] to reduce communication costs.

There are three more determinants, namely which main memory sorting method is used, how intermediate files are managed on disk (or tape), and whether the local sort or merge step is performed before or after the data exchange step. We will come back to these points when we describe Volcano's sort algorithms.

3. Overview of Volcano

In this section, we provide an overview of the modules 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.

Associated with each algorithm is a *state record*. The arguments for the algorithms, e.g., predicate evaluation functions, are kept in the state record. All functions on data records, e.g., comparisons and hashing, are compiled prior to execution and passed to the processing algorithms by means of pointers to the function entry points.

In queries involving more than one operator (i.e., almost all queries), 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 multiple 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 *QEP* structure which includes four pointers to the entry points of the three procedures implementing the operator (*open*, *next*, and *close*) and a state record. 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.

Calling *open* for the top-most operator results in instantiations for the associated state record, e.g., allocation of a hash table, and in *open* calls for all inputs. In this way, all iterators in a query are initiated recursively. In order to process the query, *next* for the top-most operator is called repeatedly until it fails with an *end-of-stream* indicator. Finally, the *close* call recursively "shuts down" all iterators in the query. This model of query execution matches very closely the one being included in the E programming language design [10] and the algebraic query evaluation system of the Starburst extensible relational database system [11].

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 hold on to it for a while (e.g., in a hash table), unfix it, e.g., when a predicate fails, or pass it on to the next operator. Complex operations like join that create new records have to fix them in the buffer before passing them on, and have to unfix input records.

Another benefit of ananymous inputs is that 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 invokation of *close*.

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

4. Multi-Processor Query Evaluation

When we considered porting Volcano to a multi-processor environment, we decided that it would be desirable to use the query processing code described above *without any change*. The result is very clean, self-scheduling parallel processing.

The module responsible for parallel execution and synchronization is the *exchange* iterator. Notice that it is an iterator with *open*, *next*, and *close* procedures; therefore, it can be inserted at any one place or at multiple places in a complex query tree.

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 using *open*, *next*, and *close* on its input. The output of *next* is collected in *packets*, data structures of 1KB which contain about 80 *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.

When its input is exhausted, the exchange operator in the producer process marks the last packet with an *end-of-stream* tag, passes it to the consumer, and waits until the consumer allows closing all open files. This delay is necessary because a virtual file used for intermediate results must not be closed before all its records are

unpinned in the buffer. Virtual devices and files are used to provide space for intermediate query results.

While all other modules are based on demand-driven dataflow (iterators, lazy evaluation), the producer-consumer relationship of exchange uses data-driven dataflow (eager evaluation). 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 CPU's execute different subtrees of a complex query tree. Bushy parallelism and vertical parallelism are forms of *inter-operator* parallelism. Intra-operator parallelism means that several CPU's perform the same operator on different subsets of a stored dataset or an intermediate result.

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.

If an operator or an operator subtree is executed in parallel by a *group* of processes, one of them is designated the *master*. When a query is *open*ed, only one process is running, which is naturally the master. When a master forks a child process in a producer-consumer relationship, the child process becomes the master within its group. If the producer operation is to run in parallel, the master producer forks the other producer processes.

After all producer processes are forked, they run without further synchronization among themselves, with two exceptions. First, when accessing a shared data structure, e.g., the port to the consumers, short-term locks

must be acquired for the duration of one linked-list insertion. Concurrent invocation of routines of the file system, in particular the buffer manager, is described later in this section. Second, when a producer group is also a consumer group, i.e., there are at least two exchange operators and three process groups involved in a vertical pipeline, the processes that are both consumers and producers synchronize twice. During the (very short) interval between synchronizations, the master of this group creates a port which serves all processes in its group.

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

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 VTOC. 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; therefore it is never held for a long period of time. A descriptors or cluster lock must be held while updating a descriptor in the buffer, e.g., to decrease its fix count, or while doing I/O.

5. Single-Process External Sorting

External sorting is known to be an expensive operation, and a large number of algorithms has been devised [4]. In all of our sort algorithms, we try to exploit the duality between main memory mergesort and quicksort. Both these algorithms are recursive divide-and-conquer algorithms. The difference is that mergesort first divides physically and then merges on logical keys, whereas quicksort first divides on logical keys and then combines

physically by trivially appending sorted subarrays.

In general, one of the two phases dividing and combining is based on logical keys whereas the other arranges data items only physically. We call these the logical and the physical phases. Sorting algorithms for very large data sets stored on disk or tape are also based on dividing and combining. Usually, there are two distinct sub-algorithms, one for sorting within main memory and one for managing subsets of the data set on disk or tape. The choices for mapping logical and physical phases to dividing and combining steps are independent for these two sub-algorithms. For practical reasons, e.g., ensuring that a run fits into main memory, the disk management algorithm typically uses physical dividing and logical combining (merging). A point of practical importance is the fan-in or degree of merging, but this is a parameter rather than a defining property of the algorithm.

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 phase is left. The last merge step is performed in the *next* procedure, i.e., when demanded by the consumer of the sorted stream[1]. Similarly, the input to Volcano's sort module must be an iterator, and

[1] In the actual implementation, *next_sort* and *close_sort* are also used by *open_sort* to create runs, i.e., for writing level-0 runs after quicksort and for merging runs of one level into a run at a higher level. This is made possible by additional fields in the state record.

sort uses *open*, *next*, and *close* procedures to request its input.

Quicksort is only one of a number of alternative methods for generating initial runs. We also considered using a heap in memory because a heap allows creating initial runs twice as large as memory [4]. The basic idea is that after the first record has been written to a run, it is immediately replaced in memory by another record from the input. If the new record's key is greater than all keys written into the run so far (assuming an ascending output sequence), the new record can be included in the current run. Otherwise it is tagged to go into the next run. The tag of the last record written is considered the run's tag. The record tags are always included in record comparisons, and in case of different record tags, the record with a tag equal to the current run's tag is considered less.

At first, the probability is quite high that the next input record can be included in the current run. As more records are written into a run and as the last key written to the current run increases, the probability decreases that the next input record's key is greater. On the average, for a random input sequence, runs can be expected to contain twice as many records as the heap, and many more if the input sequence is closer to ordered.

In our environment, however, the advantage of larger initial runs is not without cost. For the sake of explanation, let us assume that we may use B buffer pages which hold R records each. Consider how records are placed in pages. Typically, and in the best case, records in the input stream are packed densely in pages in the buffer, i.e., we could quicksort BR records in this space. Ideally, we would like not to move records in the buffer; therefore, we choose to let the heap contain BR pointers to records as they were produced as sort input. If records are removed selectively from the heap, the records remaining in the heap will not be packed densely in pages. On the average, the pages pointed to in the buffer will be half full. In order not to overcommit the sort space, the heap size must be reduced to about half, i.e., $\frac{1}{2}BR$. But that exactly offsets the advantage of creating runs twice as long as the heap! In other words, nothing was gained.

In order to save the advantages of heap-based run creation, we could copy records into a designated heap space, and keep this heap space always densely packed. This, however, would introduce another copying step for all records in the input stream. We considered this prohibitively expensive, and abandoned the idea of using

heaps for creating initial runs.

The next step is merging runs into larger runs and finally into the output stream. Merging is also limited by the memory size, since an input buffer is needed for each input run[2]. The *maximal merge fan-in* can be determined by dividing the memory size by the input runs' cluster size.

Volcano's merge implementation uses a binary heap quite similar to the tournament sort in [4]. The heap size must be a power of 2, say 2^d. If the number of merge inputs is not a power of 2, the heap is "padded" with empty runs. An array of length 2^d is used to store the heap. An item from run r starts its tournament at array entry $e = (r+2^d)/2$. Thus, two runs start at each entry point. The run that "wins" the "match" at e advances, and "competes" next at $\lfloor e/2 \rfloor$. The "winner" of the entire tournament moves to entry 0.

When creating the heap, the tournament must be limited. When the first run is inserted into the heap (run 0), no tournament takes place. When the next run (run 1) is added, one comparison is performed and one of the runs advances. No tournament takes place when run 2 is added. For run 3, two comparisons take place. In the second comparison, the winner of 0-1 and the winner of 2-3 are compared, and one of them advances. In general, during creation of the heap, run r is compared exactly as often as the number of trailing 1's in r. This trick also works if run 2^d-1 is added first. In this case, trailing 0's control the tournament depth for each run.

We have encountered two basic merging strategies which we call *eager* and *lazy* merging. In eager merging, higher level runs are created as soon as the number of lower level runs reaches the maximal merge fan-in. In this strategy, the number of existing runs is quite limited, and runs can easily be kept track of.

In lazy merging, merging is delayed until the entire input is consumed and sorted into initial runs. A potentially large number of runs must be kept track of, but lazy merging has a significant advantage. Consider a system in which the maximal fan-in is F. If the input size requires $F+2$ initial runs, eager merging writes and reads each record to two run files. In lazy merging, it is sufficient to merge three level-0 runs into one level-1 run. The remaining $F-1$ level-0 runs and the single level-1 run can then be merged in a single step. Knuth describes and analyzes many such optimizations in much more detail [4].

[2] In Volcano, there is no limit on the number of open files.

From a different viewpoint, merging is a way to reduce the number of sorted runs, with the goal to reduce this number to one. In order to make best use of the final merge step, the number of runs should be reduced to F. Since each merge step reduces the number of runs by $F-1$ (removing F runs, creating 1 run), generalizing this idea suggests reducing the number of runs in the first merge step to $F+k(F-1)$ for some suitable k, and then decrement k with each merge step.

To visualize the advantage of such optimizations, consider how the cost of merging grows for increasing file sizes. For eager merging, the cost is basically linear with the number of records until the input size requires an additional merge level[3]. At this point, the cost increases by a significant step since all records make an additional trip to and from disk. For lazy merging, the cost function is much smoother. The two cost functions are equal at the lower end of each step. Between steps, however, the cost of lazy merging increases gradually following an $N \log N$ function, and is therefore much lower than the cost of eager merging.

Volcano uses a hybrid strategy which we call *semi-eager* merging. This strategy combines the advantages of eager and lazy merging, even if the input size is not known a priori. We call it semi-eager because it merges eagerly when the number of runs on a level reaches $2F$. Since there is a limit on the number of runs on each level, the number of intermediate files in a sort is also limited. At the end of the input stream, between F and $2F$ runs are left at each level except the highest. The final merging starts at level 0 and continues for all levels. Merging during *open* terminates when the number of runs at the highest level is equal to F. At each level, one of three actions is taken. If the number of runs is F or more, F runs are merged. Otherwise, if the sum of the number of runs at the current and the next higher level is less than or equal to $2F$, the runs at the current level are "promoted," i.e., moved to the next level in the merging scheme without actually moving any data. Otherwise, runs from the next level up are "demoted" to fill the current level to F runs, and these runs are merged. At each level, we make sure that we promote the largest or demote the smallest files. Since the number of runs on one level can be larger than F, a second merge step on the same level might be required.

[3] The I/O cost is linear with the number of records. The comparison cost is not exactly linear due to the increased number of runs to be merged.

Volcano's sort module includes a number of additional optimizations which we will describe in the remainder of this section. Since sorting is frequently used for aggregation or duplicate elimination [12, 13, 14], we included these operations in the sort module. Notice that from an algorithmic standpoint, these operations are almost the same. Including aggregation in the sort has the benefit that duplicates can be removed early, i.e., while writing a run to disk. Consider a grouping operation which aggregates 1,000 groups from 1,000,000 input records. If aggregations are performed as early as possible, no run at any level will include more 1,000 records. If aggregations are delayed until the sort is completed, runs with 100,000 records might be created. Bitton and DeWitt presented an external sort algorithm with duplicate elimination using two-way merging, and demonstrated its superior performance when compared to sorting with subsequent duplicate elimination [14].

During merging, half of the I/O's will be input and half will be output operations. The input operations refer to a number of files and do not exhibit any locality, but the output operations are basically sequential. To benefit from this fact, the Volcano sort module can use two alternate devices for runs of different levels. Since sequential I/O operations are much more efficient than random I/O's, this simple mechanism can result in a significant speedup by cutting the number of disk seeks in half. Unfortunately, when runs are promoted and demoted for the final merge after the end of the input stream, this benefit is lost. The exact tradeoff between promoting/demoting and alternate devices is left for future analysis.

As for all files in Volcano, the cluster size can be set for each file individually. There are a number of interesting observations about how the I/O cost depends on the cluster size. Obviously, large clusters allow more efficient data transfer and seem therefore very desirable. Recall, however, that the maximal merge fan-in is the quotient of buffer size and cluster size. If the input is sufficiently large, and if the cluster size is too large, additional merge levels might become necessary. What, then, is the best cluster size?

To calculate the I/O cost depending on the cluster size, we must distinguish between the number of *seek* operations (which for the purpose of this analysis include rotational latency) and the number of bytes transferred. The number of seek operations, interestingly, turns out to be independent of the cluster size. If we increase the cluster size, say by a factor of 2, we cut the maximal merge fan-in in half and therefore double the number of merge levels. Consequently, all records have to make twice as many trips to and from disk as with

the original cluster size. Since we doubled the cluster size, however, the seek cost per record and trip is cut in half, resulting in no difference in the total seek cost.

The transfer cost, on the other hand, has indeed doubled, since each record has to be transferred twice as often. As long as transfer costs are dominated by seek costs and therefore negligible, the cluster size does not really matter. For example, a cluster of 1 KB can typically be transferred to or from disk in 0.25 to 0.5 ms, compared to 25 to 35 ms seek and latency time. As a rule, it seems reasonable to choose the cluster size to be as large as possible such that it still allows single level merging.

We have not considered read-ahead in our I/O cost calculations. Read-ahead clearly has potential benefits, particularly if it is controlled by the smallest largest key in all pages in the merge input buffer, a technique usually called *forecasting*. Although Volcano's design and implementation include a buffer daemon for read-ahead and write-behind, we left the additional complexity of analyzing performance and experimental results due to read-ahead for the future. For the performance measurements reported in Section 7 the buffer daemon was disabled.

Finally, we would like to remark briefly on record vs. key sorts. Volcano's sort module uses record sorting but does not preclude key sorting. Recall that the input into the sort iterator is a stream. Condensing each record into a *key–RID* pair and materializing entire records from *key–RID* pairs can easily be accomplished with the *filter* and *functional join* modules, both of which are standard parts of Volcano. Volcano is designed to be a set of mechanisms; hardly anything that can easily be built from components is provided explicitly in Volcano. Therefore, Volcano's sort module only sorts entire records.

6. Parallel Sorting

We believe that parallel sorting is of interest in its own right, even though we feel that the importance of sorting for query processing will diminish as such algorithms are replaced by hash-based algorithms. Much work has been dedicated to parallel sorting, but only few algorithms have been implemented for 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 [15, 5, 6, 16]. In a highly parallel architecture, any centralized component that has to process all data is bound

to be a severe bottleneck.

For parallel sorting, we have essentially the same choices as for any divide-and-conquer sort algorithm. This fact has been observed and used before by Baer et al. [17]. Besides the two choices described above for disk-based sorts, a similar decision has to be made for the data exchange step. We assume that data redistribution among the processors or disks is required, and we wish to avoid transferring a data item between processing nodes more than once. Therefore, any algorithm has a local sort step and a data exchange step. We can perform the redistribution step either before or after the local sort step.

Consider the latter method first. After all data have been sorted locally on all nodes, all sort-nodes start shipping their data with the lowest keys to the receiving node for this key range. The receiving node merges all incoming data streams, and is the bottleneck in this step, slowing down all other nodes. After this key range is exhausted on all sources, the receiving node for the second key range becomes the bottleneck, and so on. Thus, this method allows only for limited parallelism in the data exchange phase[4]. The problem can be alleviated by reading all ranges in parallel. It is important, however, to use a smart disk allocation strategy that allows doing this without too many disk seeks. We are exploring the possible strategies and their implications on overall system performance.

The second method starts the parallel sorting algorithm by exchanging data based on logical keys. 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

[4] This is not a problem for CPU scheduling in a shared-memory system that uses one central run queue as our system does. Depending on the disk configuration, however, it might be a problem due to uneven disk load. In a shared-nothing architecture it clearly is a problem.

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[5]. For load balancing among N processors, the i/N quantiles for i=1,...,N-1 need to be determined. Finding the median for a dataset distributed to a set of processors with local memory has been studied theoretically [18]. We need to extend this research for a set of quantiles, and adapt it for a database setting, i.e., for disk-based large datasets. Sufficient load balancing can probably be achieved using good estimates for the quantiles instead of the exact values. Our work on describing data distributions using moments and density functions may provide significant assistance for this problem [19].

We implemented both parallel sorting methods in Volcano. The second 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. For the first method, local sorts followed by merges at the destination site, we needed to implemented another module, *merge*, and to extend the *exchange* module.

The merge iterator was easily derived from the sort module. It uses a single level merge, instead of the cascaded merge of runs used in sort. The input of a *merge* iterator is an *exchange*. Unlike other operators, the merge iterator must distinguish the input records by their producer. As an example, for a join operation it does not matter where the input records were created, and all inputs can be accumulated in one input stream. For a merge operation, it is crucial to distinguish the input records by their producer in order to merge multiple sorted stream correctly.

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

[5] 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.

control) used between producer and consumer part of *exchange*, and the logic for *end-of-stream*.

7. Performance Evaluation

In this section, we present experimental performance results. The measurements were obtained on a Sequent Symmetry with 8 CPU's connected via a 80 MB/s bus, 2 dual-channel disk controllers, and 8 disk drives, two per channel. The CPU's were 16 MHz Intel 80386 CPU's with 64 KB cache using a write-behind cache protocol (Rev. B boards). The disks were Fujitsu Swallow 3 drives with 264 MB storage, 20 ms average seek time, 8.3 ms average latency time, and 2.46 MB/s transfer rate. A portion of each disk was configured for UNIX file systems, while another portion of about 110 MB was opened by Volcano as raw device.

File space was allocated in extents of 4 MB with a cluster size of 32 KB. We used 8 MB sort space within a 10 MB buffer. The physical memory of the machine was more than 30 MB such that virtual memory page faults were basically eliminated. If multiple processes compete for the sort space, it is divided equally among them. The record length was uniformly 100 bytes to make the performance measurement comparable to other systems [20]. The keys are four-byte integers randomly chosen from the range 0 to $2^{21}-1$.

In the following graphs, time measurement curves are marked $\times$ and relate to the scales on the left side of the graphs. Speedups curves, where used, are marked $\square$ and relate to the scales on the right side of the graphs. Measurements are shown using solid lines, while derived data are shown using dashed lines, in particular speedup curves. The ideal speedup is shown using a dotted line. Speedups are calculated relative to single process performance.

7.1. Multiple-Input Multiple-Output Sorting

We measured the elapsed times for sorting partitioned or striped data files. We assume random partitioning for the input, and require range-partitioning for the output file. Records were exchanged between processes using quantiles determined a priori and then sorted using the sort operator. The elapsed times given in this section do not include mounting and dismounting the devices, but do include flushing the buffer at the end of the sort.

First, we measured the speed-up when increasing the number of disks and the number of processes while keeping the total number of records constant.

Originally, we used two processes per disk, one to perform the file scan and partition the records and another one to sort them. We realized that creating more processes than processors inflicted a significant cost, since these processes competed for the CPU's and therefore required operating system scheduling. While the scheduling overhead may not be too significant, in our environment with a central run queue processes migrate. Considering that a large cache associated with each CPU, cache migration adds significant costs.

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

This mode of operation[6] also makes flow control obsolete. A process runs the producer operators (and produces input for the other processes) only if it does not have input for the consumer operators. Therefore, if the producers are in danger of overrunning the consumers, none of the producer operators gets scheduled, and the consumers consume the available records.

Figure 1 shows the elapsed times and speedup for sorting 100,000 100-byte records, i.e., 10 MB of data. The elapsed time (marked $\times$) increases when a single process is split into a pipeline, from 119 to 136 seconds, for reasons that we have not determined yet. As the number of processes and disks increased, the elapsed time decreased from 136 to 25 seconds.

The speedup curve (marked $\square$) shows these improvements, but also shows that the speedup falls short of the ideal, linear speedup, shown by the dotted line. The reason is that we only increased the number of processes and disks, but not the sort and buffer space. It cannot really be expected that the performance of a

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

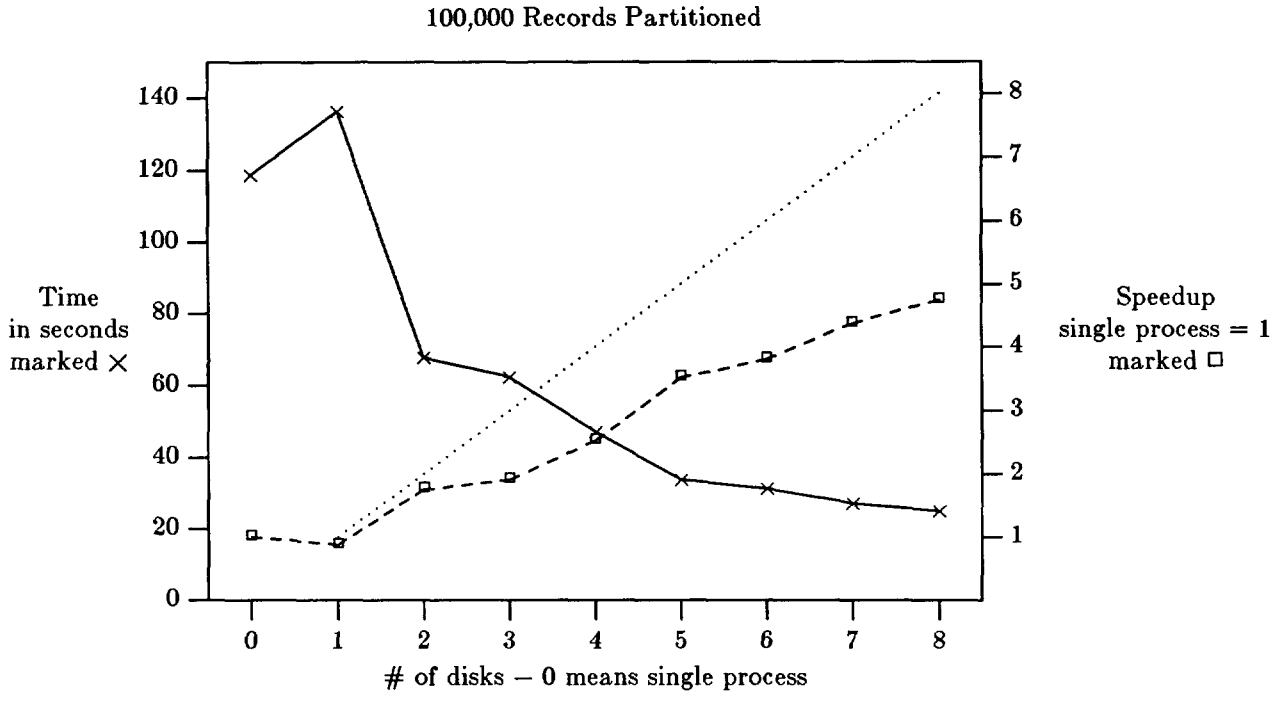


Figure 1. Multiple-input multiple-output Sort.

memory-intensive operation such as sorting improves linearly with the number of processes or disks if only those and not all resources are increased simultaneously.

For 1,000,000 records, or 100 MB of data, the disks partitions available for our use were too small to allow single-process single-disk sorting. Recall that disk space for about three times the actual data volume is needed for the input file, intermediate files, and the output file. Since there is always some amount of fragmentation on the disk, we started our experiments with 4 disks and sort processes.

Figure 2 shows that the elapsed times, between 362 and 195 seconds, are a little less than 10 times those for 100,000 records, even tough one would have expected them to be a little more. Again, we haven't determined the exact reason, but we suspect that it results from the burst pattern of the sort input requests. Recall that the sort space is loaded without any processing of the records, and that no input requests are issued while the records in the sort space are sorted using quicksort and written into an initial level-0 run. While a process performs a quicksort or writes a run, it does not produce any records for the other processes. We suspect that in the 100,000 record case this might lead to waiting times at the end when almost all processes have exhausted

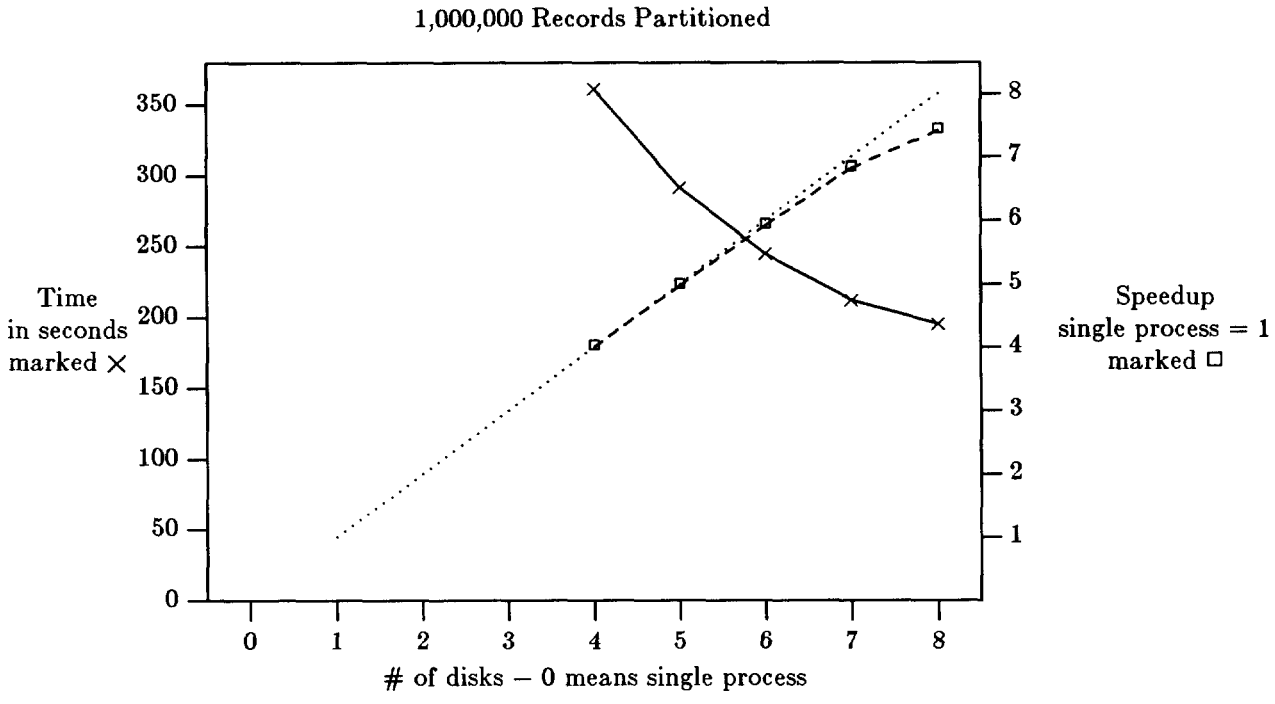


Figure 2. Multiple-input multiple-output Sort.

their local inputs.

It is interesting to note that the speedup in this experiment is quite close to linear. This fact gives encouragement for our plan for using the sort algorithm with even higher degrees of parallelism.

The next experiment was a capacity scaling experiment, i.e., we increased the number of records, CPU's and disks proportionately. First, we assigned 10,000 records to each disk. Thus, Figure 3 represents sort runs between 10,000 and 80,000 records, or 1 to 8 MB. It would have been desirable if the response time had been constant, but it is immediately apparent that this is not the case. In fact, the response time more than doubles between single-process performance (marked 0 at the bottom) and eight sort processes, from 7.66 to 17.48 seconds.

We suspect a number of reasons for this behavior. First, we had observed in earlier experiments [21] that forking processes is quite expensive. Second, with an increased number of sort processes, waiting increased at the end of the input, as discussed above. Third, data exchange between operators is not entirely free since it

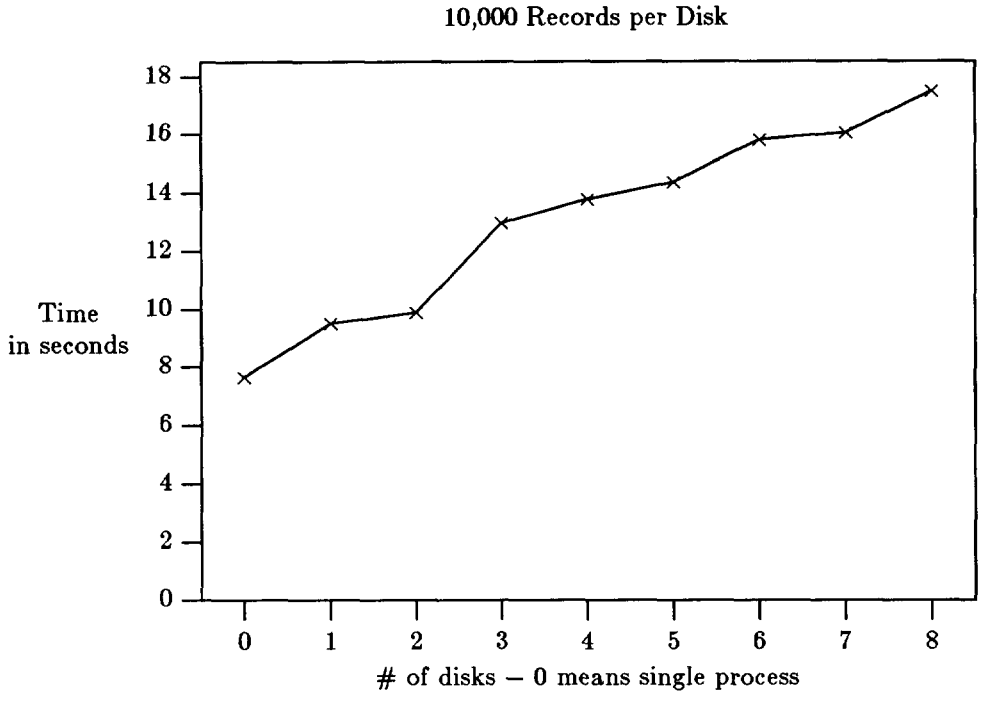


Figure 3. Single Output, Medium File.

uses operating system semaphores.

When we run the same experiment with 10 times as many records, as shown in Figure 4, we observed closer to constant elapsed times. While single-process performance is superior to single-pipeline performance by 120 to 137 seconds for 100,000 records in either case, elapsed times do not increase very much when more disks and records were added, to 159 seconds for 800,000 records on eight CPU's and disks. As in the speedup experiments, we observe better performance for large data sets.

7.2. Multiple-Input Single-Output Sorting

In the following experiments, there was one sort process for each disk, using the sort operator to deliver a sorted stream to an exchange module. Furthermore, there was one process that merged the sorted input streams and simulated the application program. The application program did nothing, it only consumed the records and unfixed them in the buffer. In the case of one input process, the sort process and the merge process were simply a pipeline. We did not run this program as single process.

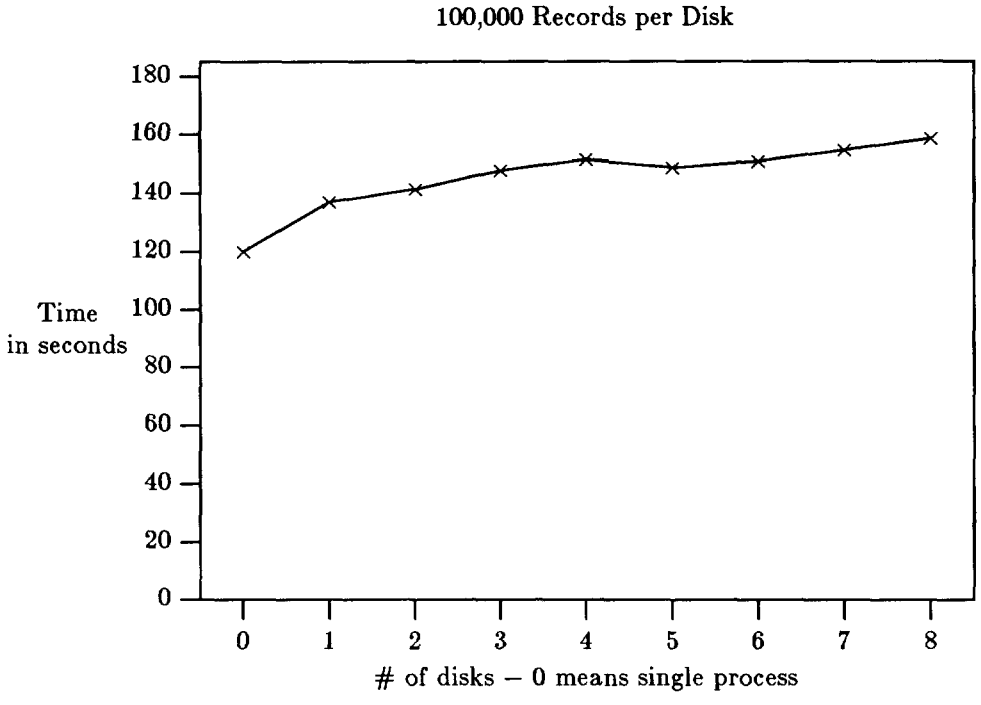


Figure 4. Single Output, Large File.

In these experiments, the processor utilization for the merge processor is obviously a potential bottleneck. We decided to include this process' time in user mode in the following figures. This time indicates what percentage of a single CPU's power was spent on the merge process, independent of which CPU's actually ran this process. This time was gathered by the operating system, DYNIX. The system time was only about 5% of the user time, which speaks for the efficiency of Volcano's data exchange mechanism. The scale, given on the left inside of the graphs, runs from 0% to 100%, and corresponds to the dashed line marked with ○'s.

Figure 5 shows elapsed times for sorting and merging data from one to eight disks into an application program. Initially, when there were only very few sort processes, the merge process could easily keep up with them, and we observe almost linear speedup, 101 to 51 seconds. As more disks and processes were added, however, the merge process became a bottleneck. Toward the end of the curve, the elapsed time hardly decreased at all, e.g., from 35 seconds for 6 disks to 32 seconds for 8 disks, while the merge process became more and more loaded, 51% for 6 disks to 58% for 8 disks.

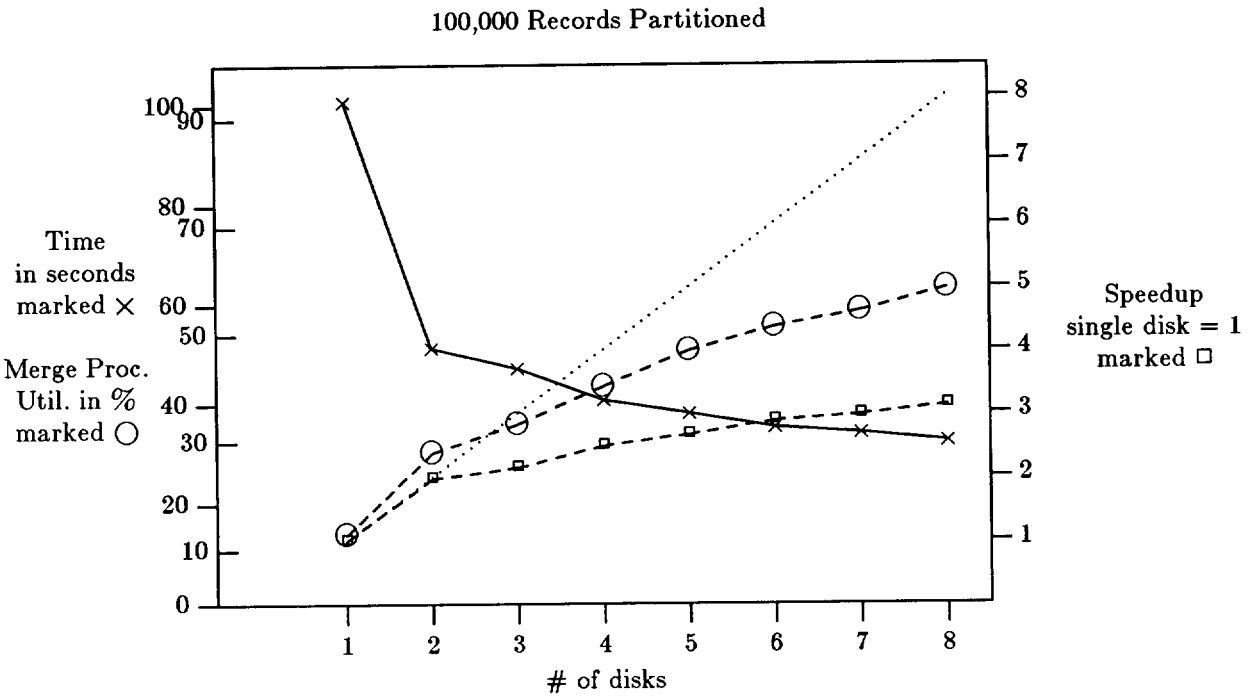


Figure 5. Multiple-input single-output Sort.

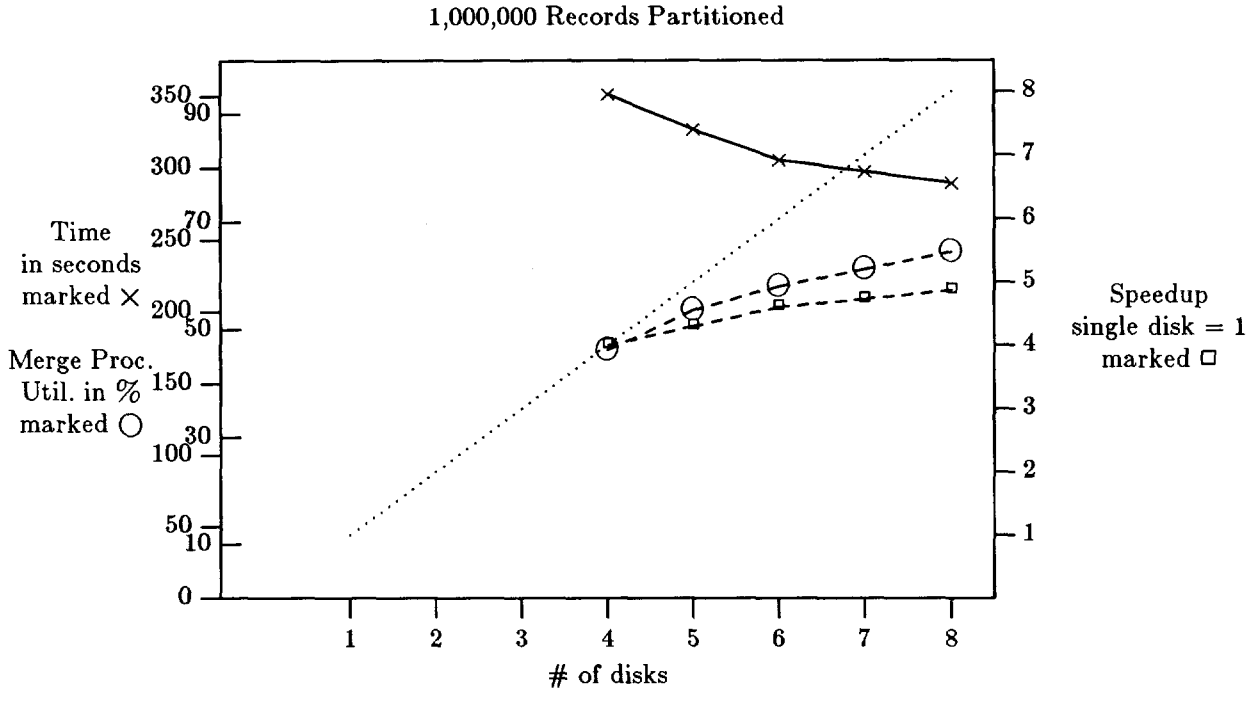


Figure 6. Multiple-input single-output Sort.

Figure 6 shows the elapsed times for sorting and merging 1,000,000 records in an application program. As in the earlier experiments, we could not sort 100 MB of data with less than four disks. As could be expected from the previous graph, additional resources for the sort phase do not improve performance as much as desirable, only from 351 seconds for 4 disks to 289 seconds for 8 disks. The bottleneck, obviously, is the merge process, which requires from 46% to 64% of one processor's power.

We conclude from these experiment that making effective use of parallel sorting is not possible when "feeding" sorted records into a single-thread application program. Two alternative conclusions can be drawn from this. Either we restrict ourselves to low degrees of parallelism, as was presumed in the design of FastSort [7], or we focus on parallelizing not only the database query processing engine but also the applications, as suggested in [22].

In these experiments, we did not see a detrimental effect of parallelism. Since we assume a fixed total memory in the machine, the available memory had to be divided among the sort processes for sorting and merging. As the number of processes increases, there is less space for each single process. Consequently, the initial

runs are shorter, and the fan-in in each merge is reduced. Thus, important tuning parameters have changed for the worse, and sorting cannot be expected to perform equally well. We are currently investigating how to adjust to variety of parameters to always guarantee optimal sort performance in Volcano.

8. Summary and Conclusions

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

Volcano's sort operator not only includes aggregation and duplicate elimination, it also uses a number of performance enhancements. Most important among those is the semi-eager merging scheme. Using stream input and output makes the sort iterator a versatile operator that can easily and efficiently be combined with other query processing modules.

When the sort operator and the exchange operator are combined, they allow for very efficient parallel sorting. If the parallel sort is to produce a single output stream, local sort with subsequent merge can be used. If key distribution quantiles are known a priori or can be gathered by sampling, range-partitioning with subsequent local sort gives good performance and almost linear speed-up.

Acknowledgements

David DeWitt sparked my interest in parallel sorting, and many of the thoughts in this paper were developed in connection with the GAMMA project. The semi-eager merging scheme was developed while visiting the University of Wisconsin in the summer of 1988. I appreciate David letting me graduate without implementing parallel sort algorithms on GAMMA; I hope I could convince him with this work that his efforts did bear

fruit after all. Gary Kelley, Frank Symonds, and Satish Doshi of Sequent provided a sounding board for ideas, in particular the exchange operator, and generously let me to use their machine for experiments.

References

1. G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems," *Ph.D. Thesis,* University of Wisconsin, (August 1987).
2. G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference,* pp. 160-171 (May 1987).
3. G. Graefe and K. Ward, "Dynamic Query Evaluation Plans," *Proceedings of the ACM SIGMOD Conference,* p. 358 (May-June 1989).
4. D. Knuth, *The Art of Computer Programming,* Addison-Wesley, Reading, MA. (1973).
5. D. Bitton, D.J. DeWitt, D.K. Hsiao, and J. Menon, "A Taxonomy of Parallel Sorting," *ACM Computing Surveys* **16**(3) pp. 287-318 (September 1984).
6. M. Beck, D. Bitton, and W.K. Wilkinson, "Sorting Large Files on a Backend Multiprocessor," *Department of Computer Science Technical Report,* (March 1986).
7. A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan, "FastSort: An External Sort Using Parallel Processing," *Tandem Technical Report* **86.3**(1986).
8. R.A. Lorie and H.C. Young, "A Low Communication Sort Algorithm for a Parallel Database Machine," *IBM Research Report* **6669**(February 1989).
9. K. Salem and H. Garcia-Molina, "Disk Striping," *EECS Techical Report 332,* Princeton University, (December 1984).
10. J.E. Richardson and M.J. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the ACM SIGMOD Conference,* pp. 208-219 (May 1987).
11. L.M. Haas, W.F. Cody, J.C. Freytag, G. Lapis, B.G. Lindsay, G.M. Lohman, K. Ono, and H. Pirahesh, "An Extensible Processor for an Extended Relational Query Language," *Computer Science Research Report,* (RJ 6182 (60892))IBM Almaden Research Center, (April 1988).
12. R. Epstein, "Techniques for Processing of Aggregates in Relational Database Systems," *UCB/ERL Memorandum,* (M79/8)University of California, (February 1979).
13. A. Klug, Statistical Query Facility"" "Investigating Access Paths for Aggregates using the "Abe" Statistical Query Facility," *IEEE Database Engineering* 5(3)(September 1982).
14. D. Bitton and D.J. DeWitt, "Duplicate Record Elimination in Large Data Files," *ACM Transactions on Database Systems* **8**(2) pp. 255-265 (June 1983).
15. D. Bitton Friedland, "Design, Analysis, and Implementation of Parallel External Sorting Algorithms," *Computer Sciences Technical Report* **464**University of Wisconsin, (January 1982).
16. J. Menon, "A Study of Sort Algorithms for Multiprocessor Database Machines," *Proceeding of the Conference on Very Large Data Bases,* pp. 197-206 (August 1986).
17. J.-L. Baer, S.C. Kwan, G. Zick, and T. Snyder, "Parallel Tag-Distribution Sort," *Computer Sciences Technical Report,* (85-01-03)University of Washington, (January 1985).
18. M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan, "Time Bounds for Selection," *Journal of Computer and System Sciences* **7**(4) pp. 448-461 (1972).
19. G. Graefe, "Selectivity Estimation Using Moments and Density Functions," *Oregon Graduate Center, Computer Science Technical Report,* (87-012)(November 1987).
20. Anon. et al., "A Measure of Transaction Processing Power," *Datamation,* pp. 112-118 (April 1, 1985).
21. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System," *Oregon Graduate Center, Computer Science Technical Report,* (89-006)(June 1989).

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