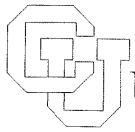
Parallel External Sorting in Volcano

Goetz Graefe

CU-CS-459-90



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Parallel External Sorting in Volcano

Goetz Graefe

CU-CS-459-90 March 1990

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

Parallel External Sorting in Volcano

Goetz Graefe

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

Abstract

Volcano is a new query evaluation system developed for database systems research and education. Its parallel sort algorithms are implemented by combining a single-process sort operator with a novel "parallelism operator." For parallel sorting, this operator provides efficient mechanisms for single- and multi-input and for single- and multi-output operations, in any combination. We use experimental performance results for sorting medium-size and large files on a shared-memory machine to evaluate parallelizing a database operation by combining a single-process operator and the novel parallelism operator, and to assess the feasibility of linear speed-up and scale-up for parallel sorting.

1. Introduction

In this study, we develop a new taxonomy of parallel sorting and a set of parallel external sort algorithms, discuss the choices made in Volcano, and report on their performance. By "external sort algorithm" we mean one that utilizes secondary storage for intermediate files, and is therefore able to sort files much larger than main memory. All our parallel sort algorithms are implemented by combining a single-process sort operator[2] with a novel "parallelism operator." This operator provides efficient mechanisms for parallel sorting with single-stream input or multiple, parallel inputs and single-stream output or multiple, parallel outputs, in any combination. This study addresses three questions and answers them using actual performance measurements. First, can parallel sorting be implemented efficiently using a single-process sort operator as basis and combining it with a generic "parallelization" operator? Second, can linear or at least near-linear speed-ups be obtained? Third, can parallel sorting provide substantial benefits even if the sort performance is limited by slow data input (e.g., a preceding database operation) or output (e.g., a single-process application program)?

To summarize our findings, combining a sort operator and a parallelism operator allows a wide variety of efficient parallel sort algorithms; however, linear or near-linear speed-up can only be obtained if all three components — input, sorting, and output — are parallelized. We achieved near-linear speed-up with 16 processors and disks, with a total elapsed time for sorting 1,000,000 100-byte records in less than 1½ minutes, which is, to the best of our knowledge, the best measured performance to-date for this benchmark.

We implemented our algorithms within Volcano, a new parallel and extensible query evaluation system operational on both single- and multi-processor systems. Volcano is not a complete database system since it lacks features such as a high-level query language, an optimizer, a type system for instances (record definitions), and catalogs. This is by design; Volcano is intended as an experimental vehicle for our earlier work in query optimization [13, 14, 18], as the query

---

[1] This technical report is a substantial extension of Oregon Graduate Center Computer Science and Engineering Technical Report 89-008 which it replaces.

[2] We use the term "single-process sort operator" to differentiate our sort operator from sort implementations designed specifically for parallel execution and query processing.

processing engine for research into query processing in object-oriented database systems [15, 17], and for multi-processor query evaluation [19]. The general design goal of Volcano is to provide *mechanisms* to allow research into *policies*, and to implement these mechanisms as efficiently as possible.

All operators in Volcano are designed and implemented for a single-process query evaluation system using demand-driven or lazy evaluation. Single-process operators are much easier to design, implement, debug, and tune than parallel programs because of their lower complexity and the existence of powerful tools like debuggers and profilers (e.g., *gprof* [20]). Only one operator in Volcano is designed specifically for parallel execution. This is the *exchange* operator which parallelizes all other operators. Since all operators, including the exchange operator, use and provide a uniform interface, the existence and location of exchange operators in a query tree are invisible to and do not affect the other operators.

The paper is organized as follows. In the following section, we briefly review previous work and propose a new taxonomy of parallel external sort algorithms. In Section 3, we provide an overview of Volcano and its mechanisms for parallel query processing. Section 4 details the single-process sort operation which is the basis for all parallel sort algorithms. Section 5 describes strategies and implementations of parallel sorting for very large files. In Section 6, we present and analyze experimental performance results for single-process and parallel sorting in Volcano. Section 7 contains a summary and our conclusions from this effort.

2. Previous Work

Sort algorithms have been studied extensively, both sequential and parallel algorithms. The best source for sequential algorithms probably still is [27]. For a survey of parallel sort algorithms, we refer the reader to [5]. Following Boral and DeWitt's arguments [8], we focus on software implementations of sort algorithms and ignore hardware solutions, e.g., [24, 25, 28, 31, 37, 38].

We know of only a few recent investigations and implementations of parallel sort algorithms for large files, namely [3, 6, 23, 29, 30, 36], the sorting method used in the Teradata database machine, and a Sequent-internal parallel sorting project. Bitton *et al.* focus of merging locally produced sorted streams using a multi-stage pipelined network, with each machine merging two or more streams into one output stream [3, 6], and found that limited speed-ups are obtainable. Teradata's implementation and Iyer and Dias' model of parallel sorting use multiple machines to sort local data sets and then merge them in a single step [23]. Menon used a modified block bitonic sort [30] that requires each data item to cross process and processor boundaries multiple times which might be an expensive proposition because of actual transfer time as well as synchronization delays associated with transfers. Lorie and Young found that communication costs during the final merge can be reduced by exchanging only keys, not entire records, over the network and determine records' locations in the final output on each machine for its data [29]. Tandem's FastSort and Sequent's internal sorting project assume that a single stream is randomly partitioned to multiple slave processes which produce sorted streams to be merged by the master process [36].

We now introduce a new taxonomy of parallel external sort algorithms. 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 pipelining multiple merge operations. We consider these latter variations parameters rather than algorithm properties.

The first two determinants in our taxonomy are whether the input is presented as single or as multiple files, and whether the output is desired as single or as multiple files. All four combinations have practical uses. Single-input single-output sorting is the classical case with obvious use. Tandem's FastSort is of this kind [36]. Multiple-input single-output is useful to present a large file striped over multiple disks [34] to a user or single-thread application program. Iyer and Dias' model [23], and Teradata's algorithm are of this kind, with hardware support for merging multiple locally sorted streams in the Teradata database machine. 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 local indexes. With multiple output we typically require range-partitioning, i.e., that disjoint key ranges be assigned to the sites. A range-partitioned file

with sorted partitions 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 [34], 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 shared system bus becomes a bottleneck as more processors are added. Therefore, an interconnection network must be introduced, e.g., a ring [10] or a hypercube [11], in which data transfer can be a significant cost. We will not investigate parallel sort algorithms requiring 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 (RID). 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 by Lorie and Young to reduce communication costs [29].

Table 1 gives an overview of the determinants we have identified including three points we have not discussed yet: which main memory sorting method is used, how intermediate files are managed on secondary storage, 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 in Sections 4 and 5.

3. Overview of Volcano

In this section, we provide an overview of Volcano's architecture. We outline single-process query processing and then discuss Volcano's generic "parallelization" module used to implement parallel sort algorithms and all other parallel query processing strategies.

Volcano is currently operational on a number of UNIX systems including DEC VAX, Sun, and Sequent machines. Beyond the *sort* and *exchange* operators which are the focus of this report, Volcano's operator set includes join, semi-join, outer join, intersection, union, difference, aggregate functions, duplicate elimination, and division, all of them both sort- and hash-based, plus scans, updates, and the *choose-plan* operator described in [18].

Determinant	Possible Options
Input	Single-stream vs. parallel
Output	Single-stream vs. parallel
Number of data exchanges	One vs. multiple
Data exchange	Before or after local sort
Input division	Logical keys (partitioning) or physical division
Result combination	Logical keys (merging) or physical concatenation
Main-memory sort	Quicksort or replacement selection
Sort objects	Original records or *key-RID* pairs (substitute sort)

Table 1. Taxonomy of Parallel Sorting.

3.1. Single-Process Query Evaluation

Volcano includes its own file system which is similar to WiSS [9]. Much of Volcano's file system is rather conventional. It provides data files, bidirectional scans with optional 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 and can be buffered in the same buffer pool. 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 operator is a *state record*. The arguments for the algorithms, e.g., hash table size or a hash function, are part of 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. There is also an argument passed to each function so that the function can be a generic predicate interpreter with the interpretable code as argument.

In queries involving more than one operator — 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 iterator concept in the E programming language design [33] and the algebraic query evaluation system of the Starburst extensible relational database system [21]. Table 2 gives a small set of

Iterator	*Open*	*Next*	*Close*
Print	*open* input	call *next* on input; format the record on screen	*close* input
Scan	open file	read next record	close file
Select	*open* input	call *next* on input until a record qualifies	*close* input
Hash join	allocate hash directory; *open* build input; build hash table calling *next* on build input; *close* build input; *open* probe input	call *next* on probe input until a match is found	*close* probe input; deallocate hash directory
Hash aggregation	allocate hash directory; *open* input; build hash table calling *next* on input, aggregating old and inserting new items; *close* input	return an item from hash table	deallocate hash directory

Table 2. Examples of Iterator Functions.

examples for what the *open*, *next*, and *close* functions of various operators might do.

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* consisting of a record identifier and the record's address where it is fixed in the buffer pool. The protocol for fixing and unfixing records is as follows. Each record fixed 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 anonymous inputs is that all parallelism issues can be encapsulated in a single module [19]. Volcano's "parallelism operator" is unique, and one reason for this study was to verify that the design and implementation of this operator allows efficient and effective parallel query processing. In the next section, we describe the parallelism operator, and report parallel performance results in Section 6.

3.2. Mechanisms for Multi-Processor Query Evaluation

After Volcano had been operational for a while as a single-process system, we considered porting Volcano to a multi-processor environment, and decided that it would be desirable to use the query processing code described above *without any change*. 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. In the consumer process, the exchange operator 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.

In the producer process, the exchange operator becomes the *driver* for the query tree below the exchange operator calling *open*, *next*, and *close* on its input. The output of *next* is collected in *packets*, which are arrays of *NEXT_RECORD* structures (record identifier - pointer pairs). When a packet is filled, it is inserted into the *port* and a semaphore informs 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.

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 fix 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. The initial value of the flow control semaphore determines how many packets the producers may get ahead of the consumers.

The second form of parallelism we wanted to support was *intra-operator parallelism*, which requires data partitioning. Partitioning of stored datasets is achieved by using multiple files, preferably on different devices. Partitioning of intermediate results is implemented by using multiple queues in a port. If there are multiple consumer processes, each uses its own queue. The producers can use round-robin-, key-range-, or hash-partitioning [10] in deciding to which of the queues an output record has to go.

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 subtree 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 or the buffer hash table, short-term latches must be acquired for the duration of the access. 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 master of the middle group creates a port and the processes in the middle group synchronize briefly to exchange the location of the new port.

We believe that Volcano's exchange iterator is a powerful encapsulation of process and flow control in parallel query processing. More details on this iterator, including additional modes of operation, can be found in [19]. In the next two sections, we describe Volcano's sort iterator followed by a discussion of how to implement various forms of parallel sorting by combining the sort and exchange iterators.

4. Single-Process External Sorting

In this section, we continue our taxonomy of sorting algorithms and describe the design choices made for Volcano's sort iterator. We describe the iterator in some detail because we believe there is no point in parallelizing a mediocre sequential program. We spent a fair amount of time on tuning the single-process sort operator, and believe that the good performance of the parallel version is also due to the fact that the single-process version was carefully design and tuned.

External sorting is known to be an expensive operation, and a large number of algorithms has been devised [27]. In all of our sort algorithms, we try to exploit the duality between main memory mergesort and quicksort. Both of 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 algorithm property.

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. Volcano's sort module does not restrict the size of the sort space, the fan-in of the merge phase, or the number of merge levels.

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, e.g., a merge join. Similarly, the input to Volcano's sort module must be an iterator, and sort uses *open*, *next*, and *close* procedures to request its input.

Quicksort is only one alternative for generating initial runs. We also considered using a heap in memory, i.e., replacement selection, because a heap allows creating initial runs twice as large as memory on the average after a small start-up period (two runs) [27]. The basic idea is that after a 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, the new record can be included in the current run (assuming an ascending output sequence). 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 smaller.

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 can 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 should be half full. In order not to overcommit the sort space, the heap size must be reduced to half, i.e., $\frac{1}{2}BR$. However, cutting the heap size in half exactly offsets the advantage of creating runs twice as long as the heap. In other words, nothing is gained[3].

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[4], and abandoned the idea of using heaps for creating initial runs. Furthermore, this technique does not work easily for variable-length records.

As a final remark about heap-based run creation, using heaps results in a different I/O pattern than using quicksort. In the quicksort scheme, there is a cycle of reading several input pages, sorting them, and writing them into a run file. If heaps are used, reading an input page and writing a run page alternate. This can be either very bad or very good. If input and run files reside on a single disk, this results in many disk seeks. Multiple disks, on the other hand, may be utilized effectively by using one for input and another for output. In the experiments described in Section 6, we used one disk per process, and therefore would not have gained from heap-based run generation. We claim that on today's multi-processor machines, it is more likely to use more processes than disks rather than the other way around.

After the input has been written into initial runs, the next step is merging these level-0 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. The *maximal merge fan-in* can be determined

---

[3] This calculation is not entirely correct. Using a simulation program that determines the maximum buffer size for a given heap size or the average run length for a fixed buffer size, we have observed that pages are typically less than half full. In fact, our simulation results indicate that even if heap growth is driven by buffer deallocation, i.e., the heap is grown immediately by one page each time a buffer frame is freed, the fraction of the heap that is really used will be only 35% to 45% *on the average*, meaning that replacement selection creates even shorter and more runs than quicksort with the same amount of sort space. The exact fraction depends on the blocking factor (records per page), the heap size, and to a small extent on the file size.

[4] Note that in many computer systems, memory-to-memory copying is about as fast (or slow) as disk transfer.

by dividing the memory size by the input runs' cluster size.

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 at any point of time 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 and may, depending on the disk space allocation scheme, create significant disk space fragmentation. However, lazy merging has a significant advantage. Consider a situation 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 as well as Horowitz and Sahni describe and analyze many optimizations in much more detail [22, 27].

From a different viewpoint, merging is used to reduce the number of sorted runs. The goal of merging is to reduce this number to one, the sorted output. In order to make best use of the final merge step when the largest runs are merged, 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 new 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. We will demonstrate the savings that can be achieved after we have discussed Volcano's merge strategy.

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 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.

To visualize the advantages of lazy and semi-eager merging over lazy merging, 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[5]. 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 resembling more closely an $N \log N$ function, and is therefore lower than the cost of eager merging.

Table 3 demonstrates this effect. We ran Volcano's sort iterator with a number of input sizes with and without merge optimization. The initial run size was 400 records, the merge fan-in was 10. The average depth indicates the number of merge levels. In other words, the depth columns give an indication of the I/O cost per record. For unoptimized merging, this number is always an integer. For optimized merging, fractions are possible because not all records go through the same number of merge levels. It is obvious that the cost of unoptimized merging grows in steps for larger and larger files while the cost of optimized merging grows more smoothly. The difference for 4,000 records is due to the fact that Volcano's sort iterator in eager

[5] Actually, only 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.

		Unoptimized Merging		Optimized Merging		
Input Records	Initial Runs	Records Merged	Average Depth	Records Merged	Average Depth	Percent Savings
1000	3	1000	1	1000	1	0
2000	5	2000	1	2000	1	0
3000	8	3000	1	3000	1	0
4000	10	8000	2	4000	1	50
5000	13	10000	2	6400	1.28	36
6000	15	12000	2	8400	1.40	30
7000	18	14000	2	10400	1.49	25
8000	20	16000	2	12800	1.60	20
9000	23	18000	2	14800	1.64	18
10000	25	20000	2	16800	1.68	16
11000	28	22000	2	18800	1.71	15
12000	30	24000	2	21200	1.77	12
13000	33	26000	2	23200	1.78	11
14000	35	28000	2	25200	1.80	10
15000	38	30000	2	27600	1.84	8
16000	40	32000	2	29600	1.85	7.5
17000	43	34000	2	31600	1.86	7.1
18000	45	36000	2	33600	1.87	6.7
19000	48	38000	2	36000	1.89	5.3
20000	50	40000	2	38000	1.90	5

Table 3. Effect of Lazy or Semi-Eager Merging.

merging first merges the 10 runs into a temporary file and only then detects that the input is exhausted.

Volcano's sort module includes further optimizations concerning reduced fan-in for the final merge, aggregation and duplicate elimination, sorting to find only the top-most N elements, using alternating devices to eliminate disk seeks while writing a merge output, automatic adjustment of cluster sizes to reduce disk seeks at the expense of smaller fan-in and more data transfer, and writing the last level-0 run in decreasing (reverse) order to allow reclaiming some pages during the following merge step. We will describe several of these optimizations in the remainder of this section.

The *semi-eager* merging scheme allows choosing the fan-in of the final merge separately from the default or maximal fan-in. There are two cases in which this is beneficial. First, if two sort operations feed a merge join operator, each of the sort modules should use only half the memory for the final merge. This can be accomplished by setting the *final-fan-in* argument in the state record to half the value of the *max-fan-in* argument. If the output of the merge join is to be sorted immediate on a new key, e.g., for a subsequent merge join on a different join attribute, it might make sense to set the final fan in to third of the maximum to divide the memory equally over three active sort operators. Second, if the consumer of the sorted output is rather slow, for instance an interactive user who "pages" through the output, it might be desirable to write the entire sorted file to disk and free up all sort memory before presenting the output to the consumer. This can be accomplished by setting the *final-fan-in* argument to one. If the entire input fits into the sort memory, this argument value forces that a run be written immediately and read as demanded by the consumer, rather than leaving the entire input in the buffer as described earlier.

Since sorting is frequently used for aggregation (grouping) and duplicate elimination [4, 12, 26], we included these operations in the sort module. Notice that from an algorithmic standpoint, aggregation and duplicate elimination 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 1,000,000 input records into 1,000 groups. 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 have to be created. Bitton and DeWitt presented an external sort algorithm using early duplicate elimination and demonstrated its superior performance when compared to sorting with subsequent duplicate elimination [4]. Early duplicate elimination or aggregation generally pays off if the reduction factor is greater than or equal to the fan-in during the final merge.

When sorting is used to determine the top N values, e.g., in a database query to find the ten highest salaries, a similar argument holds for the intermediate result files. Therefore, Volcano's sort iterator can be limited to produce only the first N records and to never create intermediate files with more than N records. The last two options can be combined, e.g., to find the ten departments with the highest average salaries.

During merging, half of the I/O's operations will be input and half will be output operations. The input operations refer to a number of files and are unlikely to exhibit much locality, but the output operations are basically sequential. To benefit from this fact, Volcano's sort module can use two alternating 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 speed-up 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 alternating devices is left for future analysis.

The cluster size (i.e., the unit of I/O and buffering) can be set for each file individually in Volcano. 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. However, since the maximal merge fan-in is the quotient of buffer size and cluster size, if the cluster size is too large, and if the input is sufficiently large, additional merge levels might be necessary. What, then, is the best cluster size? When deriving the formula that determines the I/O cost for merging from the input size, memory size, cluster size, and the ratio of disk access (seek and latency) and transfer costs, we found that the input size has no effect on the optimal cluster size[6]. Therefore, it is possible to determine the optimal cluster size before the input is processed, independently of the accuracy of the query optimizer's estimate for the input cardinality. Volcano's sort operator includes an optimization procedure that determines the optimal cluster size on the fly. The formulas for this consideration are given in the Appendix.

To demonstrate the effect of cluster size optimizations, we sorted 100,000 100-byte records, about 10 MB, with merge optimization enabled using a sort space of forty pages (160 KB) within a fifty-page (200 KB) I/O buffer, varying the cluster size from one page (4 KB) to fifteen pages (60 KB). The initial run size was 1,600 records, for a total of 63 initial runs. For the device used for all run files (and only for those), we counted the number of I/O operations and the transferred pages, and calculated the total I/O cost by charging 25 ms per I/O operation (for seek and rotational latency) and 2 ms for each transferred page (assuming 2 MB/s transfer rate). As can be seen in Table 4, there is an optimal cluster size with minimal cost. It is clearly suboptimal to always choose the smallest cluster size (one page) to obtain the largest fan-in and fewest merge levels. Volcano's automatic cluster size optimization would have chosen clusters of eight pages for this sort space size; this is slightly off the optimum because the calculation in the sort operator uses an approximation of the actual merge pattern.

We have not considered read-ahead in our I/O considerations. Read-ahead clearly has large potential benefits, as shown very succinctly by Salzberg [35]. An interesting special method is to control read-ahead by the smallest largest key in all pages in the merge input buffer, a technique usually called *forecasting*. If read-ahead without forecasting is used, the fan-in is reduced to one half since half the memory is used as read-ahead buffers (double buffering) [35]. If forecasting is used, the amount of memory dedicated to read-ahead can be varied between a single page and half

[6] This statement is not entirely correct. In our derivation, we did not round up the number of required merge levels since Volcano's semi-eager merge scheme allows "half" levels. The input size does indeed define borderline cases and small quanta in the cost function.

Cluster Size [×4 KB]	Fan-in	Average Depth	Disk Operations	Pages Transferred [×4 KB]	Total I/O Cost [sec]
1	40	1.376	6874	6874	185.598
2	20	1.728	4298	8596	124.642
3	13	1.872	3176	9528	98.456
4	10	1.936	2406	9624	79.398
5	8	2.000	1984	9920	69.440
6	6	2.520	2132	12792	78.884
7	5	2.760	1980	13860	77.220
8	5	2.760	1718	13744	70.438
9	4	3.000	1732	15588	74.476
10	4	3.000	1490	14900	67.050
11	3	3.856	1798	19778	84.506
12	3	3.856	1686	20232	82.614
13	3	3.856	1628	21164	83.028
14	2	5.984	2182	30548	115.646
15	2	5.984	2070	31050	113.850

Table 4. Effect of Cluster Size Optimizations.

the memory. This parameter should be set such that merge (CPU) speed and read-ahead (I/O) speed match to achieve what Salzberg calls "perfect overlapping" [35]. Although Volcano's design and implementation include a buffer demon for read-ahead and write-behind, we left the additional complexity of analyzing read-ahead performance and experimental results using read-ahead for the future. For the performance measurements reported in Section 6 the buffer demon was disabled.

We would like to remark briefly on record vs. key sorts because key sorting was used by Lorie and Young to reduce communication costs in parallel sorting [29]. Volcano's sort module uses record sorting but does not preclude key sorting. Recall that the input to 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, respectively, 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.

5. Parallel Sorting

Much work has been dedicated to parallel sorting, but 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 [3,5,6,30,36]. In a highly parallel architecture, any centralized component that has to process all data is bound to be a severe bottleneck. Therefore, we focus on divide-and-conquer algorithms that avoid such bottlenecks.

For parallel sorting, we have essentially the same choices as for any divide-and-conquer sort algorithm. This fact has been previously observed and used by Baer *et al.* [2]. 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 between processors is required, and we wish to avoid transferring each data item more than once because of its overhead and synchronization delays. Therefore, each 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 the data have been sorted locally on each node, all sort-nodes start shipping their data with the lowest keys to the receiving node for the first key range, and the receiving node merges all incoming data streams. This node 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[7]. 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 sites locally sort the received data. 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 determine, prior to the redistribution step, the key values that will divide the entire data set into ranges with equal numbers of records. *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[8]. 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 [7]; 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 quantile estimates instead of exact values as could be gathered through sampling [32].

Both of these sorting methods, partitioning followed by local sorts and local sorts followed by redistribution, are implemented in Volcano. The first method could readily be implemented using the methods and modules described so far, namely the *exchange* and *sort* iterators. Interestingly, the data exchange and the sort phase overlap naturally due to the iterator behavior of the algorithms. For the second method, local sorts followed by merges at the destination site, we needed to implement 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 iterator. 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 or sent from, and all inputs can be accumulated in one input stream. For a merge operation, it is essential to distinguish the input records by their producer in order to merge multiple sorted streams correctly.

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

---

[7] This is not a problem for CPU scheduling in a shared-memory system that uses one central run queue as our system does, unless process migration between CPUs is expensive, e.g., due to cache migration costs. Depending on the disk configuration, however, it might be a problem due to uneven disk load. In a shared-nothing (distributed memory) architecture it always is a problem.

[8] 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.

6. Performance Evaluation

In this section, we present experimental performance results to answer the questions put forward in the introduction: Can parallel sorting be implemented efficiently by combining a single-process sort with a generic "parallelization" module? Can linear or at least near-linear speed-ups be obtained? Can parallel sorting provide substantial benefits even if slow data input or output limit the sort performance?

The measurements were obtained on a Sequent Symmetry with 8 CPUs connected via a 80 MB/s bus, 80 MB of shared memory, 2 dual-channel disk controllers, and 8 disk drives, two per channel. The CPUs were 16 MHz Intel 80386 CPUs with 64 KB cache using a write-behind cache protocol. Each channel can transfer data independently of other channels. 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 100 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 I/O buffer. The large physical memory ensured that this buffer was memory-resident without virtual memory page faults. If multiple processes competed for the sort space, it was divided equally among them. The record length was uniformly 100 bytes to make our performance measurement comparable to other benchmarks [1]. The keys are four-byte integers randomly chosen from the range 0 to $2^{21}-1$.

Of the four combinations of single and multiple input and output we measured the three most useful ones: single-input single-output (1:1), multiple-input multiple-output (M:N) with an equal number of input and output partitions (M=N), and multiple-input single-output (M:1). The first case is measured and reported as a special case of the second. We considered medium size files (100,000 records, about 10 MB) and large files (1,000,000 records, about 100 MB). 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. We experimented with both speed-up (constant total data volume) and scale-up (constant ratio of data volume to resources).

6.1. Multiple-Input Multiple-Output Sorting

For data partitioned over multiple disks, we started the experiments with randomly partitioned input, and required range-partitioning for the output file. We report on only one of the algorithms, data exchange followed by local sorts. Records were exchanged between processes using quantiles known a priori and then sorted using the sort operator. We measured the performance of the other alternative, local sorting followed by data exchange and merging, but did not obtain competitive results due to the bottleneck reasons put forth in Section 5.

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 CPUs and therefore required operating system scheduling. Ideally, the scheduling overhead will not be considered significant. However, in our environment, a shared-memory multi-processor with one central run queue, processes will migrate among CPUs, and considering the large cache associated with each CPU, frequent 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 on a process boundary. 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, i.e., the producer subtree in its own process, possibly sending them off to other processes in the group, until a record for its own consumer was found. In effect, in this mode the exchange operator multiplexes an OS process between producer and consumer, thereby eliminating the need for both OS scheduling and flow control.

Degree of Parallelism	Elapsed Time [sec]	Speed-up	Parallel Efficiency [%]
1	136.26	1	100
2	67.70	2.01	100
3	62.25	2.19	73
4	47.01	2.90	72
5	33.76	4.04	81
6	31.20	4.37	73
7	27.13	5.02	72
8	24.99	5.47	68

Table 5. M:N Sort, 100,000 Records.

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. Table 5 shows the elapsed times and speed-up for sorting 100,000 100-byte records, or about 10 MB of data. As the number of processes and disks increased, the elapsed time decreased from 136 to 25 seconds. The speed-up column shows these improvements, but also shows that the speed-up falls short of the ideal, linear speed-up. Another column, parallel efficiency, indicates how well the parallel program makes use of its resources. It is calculated by dividing the single-process time over the degree of parallelism times the parallel elapsed time. 100% parallel efficiency is equivalent to linear speed-up.

So far, we have identified three reasons for the speed-up results. First, we increased only the number of processes and disks, but not sort space and buffer space. As memory is divided over more processes, the initial runs get shorter, and more of the sort effort is shifted into the merge phase. The number of comparisons is not affected by this shift, but comparisons are done more efficiently in Volcano during quicksort than during merging. Looking at the problem more generally, one cannot really expect the performance of an operation to improve linearly if only some but not all resources are increased simultaneously. Second, as the memory per process and the initial run size shrink, less I/O is sequential for writing level-0 runs as opposed to random for reading and writing during the merge phase. Third, and most importantly, the more processes are involved, the more concurrent demands are made on shared resources. Other experiments indicated that Volcano's buffer organization used in these experiments did not support very many processes. With 16 processes, we observed more wait time than usage time for the latch protecting the buffer hash table.

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

Table 6 shows that the elapsed times, between 362 and 195 seconds, were a little less than 10 times those for 100,000 records, even though one would have expected them to be a little more. We suspect that this is an effect of the burst pattern of the sort input requests. Recall that the sort space is loaded without any record processing, 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

Degree of Parallelism	Elapsed Time [sec]
4	361.60
5	291.23
6	244.70
7	212.08
8	195.16

Table 6. M:N Sort, 1,000,000 Records.

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 their local inputs. The speed-up for 1,000,000 is quite close to the speed-up for 100,000 records, about 1.9 times for the increase from 4 to 8.

The next experiment was a capacity scaling experiment, i.e., we increased data volume and resources proportionally. We assigned 100,000 records to each disk. Table 7 shows the performance of sorting between 100,000 and 800,000 records, or about 10 to 80 MB. It would have been desirable if the response time had been constant, but it is apparent that this is not the case. Elapsed times increased moderately as more disks, CPUs, and records were added, from 137 for 100,000 on one to 159 seconds for 800,000 records on eight CPUs and disks. Thus, the added capacity does help to sort the added data volume, but the scale-up is not linear.

We have drawn two conclusions from these speed-up and scale-up experiments. First, parallelizing a single-process sort operator with a generic "parallelization" operator does yield substantial speed-ups. Second, linear speed-ups and scale-ups are hard to obtain, and speed-up and parallel efficiency decrease as bus saturation is approached in a shared-memory system. In these experiments, we observed a parallel efficiency of about 70%, which is reasonable but far from ideal.

Since the experiments reported above were completed, we had the opportunity to use a larger machine with 16 CPUs and 16 disk drives. With modified Volcano software, in particular concerning fixing and unfixing records in the buffer, the best performance we observed for sorting 1,000,000 100-byte records using the multiple-input multiple-output strategy were 623.42 sec for 2, 158.12 sec for 8, and 83.66 sec for 16 disks. To the best of our knowledge, this is the best measured disk-based sort performance to-date. Notice that the parallel efficiency was (2×623.42) / (8×158.12) = 98.5% for eight and (2×623.42) / (16×83.66) = 93.1% for sixteen processors and disks, i.e., Volcano's parallel sort algorithm combining the single-process sort operator and the exchange operator exhibited almost linear speed-up for 16 CPUs and disks. Unfortunately, for time reasons we were unable to complete a comparative study of different sorting scenarios, degrees of parallelism, and file sizes with the modified software on the large machine. Therefore, we continue reporting performance results obtained before the software modification.

6.2. Multiple-Input Single-Output Sorting

The following experiments show the performance of parallel sorting if the sort processes are restrained by the performance of operators on their input or output side. Since the case of limited input has been explored elsewhere [36], we consider here the case of merging the output of concurrent sorts into a single stream. 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 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.

In these experiments, the processor running the merge process is obviously a potential bottleneck. We included this process' time in user mode in the following tables. This time indicates what percentage of a single CPU's power was spent on the merge process, independent of which CPUs actually ran this process. This time was gathered by the operating system, DYNIX. The system time was only about 5% of the user time.

Degree of Parallelism	Elapsed Time [sec]
1	137.05
2	141.38
3	147.80
4	151.43
5	148.68
6	150.81
7	154.53
8	158.58

Table 7. M:N Sort, 100,000 Records per Process.

Degree of Parallelism	Elapsed Time [sec]	Parallel Efficiency [%]	CPU Utilization [%]
1	100.79	100	12.8
2	51.25	98	27.9
3	47.10	71	33.2
4	40.70	62	40.2
5	38.01	53	46.9
6	35.15	48	51.3
7	34.00	42	54.2
8	32.40	39	58.4

Table 8. M:1 Sort, 100,000 Records.

Table 8 shows elapsed times for sorting and merging data from one to eight disks into an application program. The merge process could easily keep up with very few sort processes, and we observed almost linear speed-up, 101 to 51 seconds for doubling the number of CPUs and disks. This is also reflected in the parallel efficiency of 98%. As more disks and processes were added, however, the merge process became a bottleneck. Toward the end of the experiment, 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. The low parallel efficiency of only 39% indicates the problem as well.

Table 9 shows the elapsed times for sorting and merging 1,000,000 records into 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 table, additional resources for the sort phase did not improve performance as much as desirable, only from 351 seconds for 4 disks to 289 seconds for 8 disks. The bottleneck, obviously, was the merge process, which required from 46% to 64% of one processor's power.

We conclude from these experiments that making effective use of parallel sorting is not possible when "feeding" sorted records into a single-thread application program or consumer operator, e.g., a single-process merge join. The same statement is true for a slow input stream. Two alternative solutions exist. Either we restrict ourselves to low degrees of parallelism, as was presumed in the design of FastSort [36], or we focus on parallelizing not only the database engine but also the application programs as well as the interface between database and application, as suggested in [16].

7. Summary and Conclusions

Because of its modularity and extensibility, 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 and intra-operator parallelism, and it performs the transitions from demand-driven to data-driven dataflow and back.

Degree of Parallelism	Elapsed Time [sec]	CPU Utilization [%]
4	351.20	46.1
5	326.88	53.5
6	305.48	58.0
7	297.24	61.2
8	288.99	64.4

Table 9. M:1 Sort, 1,000,000 Records.

Volcano's sort operator not only includes aggregation and duplicate elimination, it also uses a number of performance enhancements. Most important among them is the semi-eager merging scheme and the automatic adjustment of cluster sizes. Using streamed 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 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 estimated from samples, range-partitioning and subsequent local sort give very good performance and near-linear speed-up. For the standard database sorting benchmark, sorting 1,000,000 100-byte records, we observed 14.9 times speed-up on a shared-memory machine using 16 processors and disks. However, if the sort performance is limited by input or output bandwidth, only rather limited speed-ups are obtainable.

In the experiments reported here, we did not see a detrimental effect of parallelism. If we assume (or are given) a fixed total memory size, the available memory has to be divided among multiple sort processes for sorting and merging. As the number of processes increases, there is less space for each process. Consequently, the initial runs are shorter, the fan-in in each merge is reduced, and the required merge levels may increase. 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 the large set of parameters to always guarantee optimal sort performance in Volcano, in particular tradeoffs of parallelism during run generation, parallelism during merging, read-ahead, and cluster size.

Acknowledgements

David DeWitt sparked my interest in parallel sorting — most of the taxonomy was developed with him for the Gamma project. Leonard Shapiro was a great sounding board for ideas and clarified many points in this paper. Frank Symonds and Gary Kelley helped to shape the exchange operator with their interest in shared-memory parallel query processing. Mike Mozer and Jonathan Bein carefully read drafts of this paper and provided many insightful comments. Sequent Computer Systems generously provided machine time for the experiments reported here which is gratefully acknowledged. Current work on Volcano is supported by the National Science Foundation with two grants, Cogent Research Inc., and the Oregon Advanced Computing Institute.

Appendix

Consider sorting an input file of I pages using M pages of memory and the problem of determining the optimal cluster size in pages, called C. Assume that only a single device is used without read-ahead. Since Volcano's sort operator optimizes the merging process, we do not round quotients up or down in the following calculations.

First, we define some simple derived variables, namely the size of level-0 runs generated using quicksort $R = M$, the number of level-0 runs $Q = I/R$, and the maximal fan-in $F = M/C$.

Next, we determine the number of merge levels.

$$L = \log_F (Q)$$

$$= \log_F (I/R) = \log_F (I/M)$$

$$= \ln (I/M) / \ln (F) = \ln (I/M) / \ln (M/C)$$

However, even if merge optimization is used as in Volcano's semi-eager merging scheme, L must be rounded up if this formula results in less than 1. In this case, the minimal reasonable value for C can be determined by making it as large as possible with a single-level merge. However, it is possible that using larger clusters and more merge levels will result in better performance.

Knowing the number of merge levels, we can determine the transfer cost T for the entire merge process as

$$T = 2\,I\,L$$

because each page is written and read once in each merge level. In order to minimize T, we must reduce L, the number of merge levels; therefore, C should be as small as possible. Note that this is only the transfer cost; seek and latency costs must be determined separately.

Let us consider the number of I/O operations which determines both seek and latency costs. The number of I/O operations S can be determined as

$$S = T / C$$

$$= 2\,I / C \ln (I/M) / (\ln (M/C))$$

$$= (I \ln (I/M)) / (C \ln (M/C))$$

In order to minimize S, we must maximize $(C \ln (M/C))$. Considering that C's impact on the product is larger through the first than the second factor, we must maximize, up to a certain point, C to reduce seek costs.

Unfortunately, the goals to minimize S and T oppose each other; therefore, their tradeoff must be considered. Assume that seek and latency cost per I/O operation is s, e.g., 25 ms, and that the transfer time per page is t, e.g., 2 ms. The optimization problem is to minimize

$$Cost = s\,S + t\,T.$$

Substituting the formulas developed above for S and T, this is

$$Cost = \left(t + s/C\right) T = \left(t + s/C\right) 2\,I\,L$$

$$= \left(t + s/C\right) 2\,I \ln (I/M) / \ln (M/C)$$

$$= \left(2I \ln (I/M)\right) \left(t + s/C\right) / \left(\ln(M/C)\right)$$

This can be seen as a one-dimensional non-linear optimization problem, but an iteration over the physically possible range of C is probably a faster and less complex solution than a numerical optimization algorithm would be. Notice that the first factor is not dependent on C, making it is sufficient to minimize

$$\left(t + s/C\right) / \left(\ln(M/C)\right)$$

This expression is independent of the input size I which means that the optimal cluster size can be determined *a priori*, i.e., before sorting begins, even if the sort input size is not known[9].

When parallelism and read-ahead with forecasting are considered, the problem becomes a multi-dimensional optimization problem. In addition to the independent variable C, there are new independent variables for the degree of parallelism P and the fraction of memory dedicated to read-ahead K. K must be between zero and one, with practical values probably between zero and one half. Furthermore, different degrees of parallelism might be optimal during run generation and merging, say P_Q and P_M. In this case, the problem is to minimize the cost function using the variables P_Q, P_M, C, and K.

Most variables used above and the cost function become significantly more complex, starting with $R = M/P_Q$ and $F = M(1-K)/(P_M C)$. The number of runs remains $Q = I/R$, but notice that several processes will merge these runs. Therefore, the number of level-0 runs per process is the same as in the single-process case, namely

$$Q / P_Q = (I/R) / P_Q = (I / (M/P_Q)) / P_Q = I / M.$$

The number of merge levels increases, however, since the fan-in decreases as memory is divided

[9] In most database systems, the query optimizer makes a rough estimate, but such an estimate can be off by a factor of two or more.

into space for merging and read-ahead and among merge processes. Note that each merging process only merges its share of runs which is Q/P_M runs. The number of merge levels is

$$L = \log_F (Q/P_M)$$

$$= \ln\left(Q\ P_Q\right) / \left(M\ P_M\right) / \ln\left(M\,(1-K) / (P_M\ C)\right).$$

The total amount of data transfer to and from disk as well as the number of I/O operations can be computed with the same formulas as above giving

$$T = 2\,I\,L \quad \text{and} \quad S = T\,/\,C.$$

The actual cost function depends on the cost of each I/O operation and their distribution over the actual I/O devices. Thus, if N independent devices were used, we would estimate the cost as

$$Cost = s\ S\ /N + t\ T\ /N$$

$$= \left(t + s/C\right) T\ /N$$

This formula computes the time spent on I/O, it does not include consideration of read-ahead effectiveness and therefore will not directly relate to elapsed time. Since the positive influence of read-ahead is not considered in the cost formula, an optimization using the above formula will set K to zero. Finally, if in fact multiple devices are used, in particular more devices than sort processes ($N > P_Q$), strategies overlapping input and output while creating level-0 runs deserve consideration, including heap-based schemes that also generate longer runs.

In summary, there are very many considerations that influence sorting and parallel sorting costs. Their multitude prohibits to decide easily, using one simple heuristic, which parameter setting will result in the optimal sort cost for a particular system and input file.

References

1. Anon. et al., "A Measure of Transaction Processing Power", *Datamation*, April 1, 1985, 112-118.
2. J. Baer, S. C. Kwan, G. Zick and T. Snyder, "Parallel Tag-Distribution Sort", *Computer Sciences Technical Report*, Seattle, WA., January 1985.
3. M. Beck, D. Bitton and W. K. Wilkinson, "Sorting Large Files on a Backend Multiprocessor", *IEEE Transactions on Computers 37* (1988), 769-778.
4. D. Bitton and D. J. DeWitt, "Duplicate Record Elimination in Large Data Files", *ACM Transactions on Database Systems 8*, 2 (June 1983), 255-265.
5. 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.
6. D. Bitton Friedland, "Design, Analysis, and Implementation of Parallel External Sorting Algorithms", *Computer Sciences Technical Report 464* (January 1982), University of Wisconsin.
7. 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 (1972), 448-461.
8. H. Boral and D. J. DeWitt, "Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines", *Proceeding of the International Workshop on Database Machines*, Munich, 1983.
9. H. T. Chou, D. J. DeWitt, R. H. Katz and A. C. Klug, "Design and Implementation of the Wisconsin Storage System", *Software - Practice and Experience 15*, 10 (October 1985), 943-962.
10. 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.

11. 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).

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

13. G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems", *Ph.D. Thesis*, Madison, August 1987.

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

15. 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.

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

17. 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.

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

19. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", *Proceedings of the ACM SIGMOD Conference*, Atlantic City, NJ., May-June 1990.

20. S. L. Graham, P. B. Kessler and M. K. McKusick, "gprof: A Call Graph Execution Profiler", *SIGPLAN Notices Notices 17*, 6 (June 1982), 120-126.

21. 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.

22. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD., 1978.

23. B. R. Iyer and D. M. Dias, "System Issues in Parallel Sorting for Database Systems", *Technical Report RJ6585* (November 30, 1988), IBM Almaden Research Lab.

24. M. Kitsuregawa, W. Yang, S. Fushimi, H. Kimura, J. Shinano and Y. Kasahara, "Implementation of LSI Sort Chip for Bimodal Sort Memory", *Proc. Int'l. VLSI Conf.*, Munich, FRG, August 16-18, 1989.

25. M. Kitsuregawa, W. Yang and S. Fushimi, "Evaluation of 18-state Pipeline Hardware Sorter", *Proc. Int'l Workshop on Database Machines*, Deauville, France, June 19-21, 1989.

26. A. Klug, "Access Paths in the 'ABE' Statistical Query Facility", *Proceedings of the ACM SIGMOD Conference*, Orlando, FL., June 1982, 161-173.

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

28. G. S. Liu and H. H. Chen, "Parallel merging of lists in database management system", *Information Systems 13*, 4 (1988), 429.

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

30. 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.

31. M. Muraszkiewicz, "Concepts of Sorting and Projection in a Cellular Array", *Proceeding of the Conference on Very Large Data Bases*, Cannes, France, September 1981, 76-80.

32. 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.

33. 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.

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

35. B. Salzberg, "Merging Sorted Runs Using Large Main Memory", *Acta Informatica 27* (1989), 195-215, Springer International.

36. 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-June 1990.

37. H. Schweppe, H. Zeidler, W. Hell, H. Leilich, G. Stiege and W. Teich, "RDBM – A dedicated multiprocessor system for database management", in *Advanced Database Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1983, 36-86.

38. Y. Tanaka, "Bit-sliced VLSI Algorithm for Search and Sort", *Proceedings of the Conference on Very Large Data Bases*, Singapore, August 1984, 225-234.