Volcano[1]: An Extensible and Parallel Query Evaluation System

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 89-006

July, 1989

Volcano[1]:
An Extensible and Parallel Query Evaluation System

Goetz Graefe

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

Abstract

We present a new dataflow query evaluation system developed for database systems research and education. Volcano is *extensible* with new operators and algorithms because it uses a standard interface between procedures. It supports *dynamic query evaluation plans* that allow delaying some optimization decisions until run-time, e.g., for embedded queries with free variables. It includes an *exchange* operator that allows intra-operator parallelism on partitioned datasets and both vertical and horizontal inter-operator parallelism, translating between demand-driven dataflow within processes and data-driven dataflow between processes.

1. Introduction

In order to provide a testbed for database systems education and research we decided to design and implement a modular, high-performance query evaluation system with our limited resources. We spent a fair amount of time thinking about making our software flexible without sacrificing efficiency. The result is a compact system, consisting of fewer than two dozen core modules with a total of about 12,000 lines of C code. These modules includes a file system, buffer management, sorting, B^+-trees, and two algorithms each for natural join, semi-join, outer join, anti-join, aggregation, duplicate elimination, division, union, intersection, difference, anti-difference, and Cartesian product. Moreover, two modules implement dynamic query evaluation plans and allow parallel processing of all algorithms listed above.

Volcano is an operational query evaluation system implemented 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

[1] The name *Volcano* was born when David Maier asked about some drawings of index structures on the author's white board, left over from a conversation with Leonard Shapiro about the query evaluation system, whether they were pictures of "data volcanos." For a system developed in Oregon, not far from Mt. Hood and Mt. St. Helens, the name seemed just fine. The artwork on the cover was done by Kelly Atkinson from a photograph of Mt. Hood, Oregon.

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.

Many design decisions in Volcano were deliberately left open, and typically can be expressed using run time variables, e.g., the number and sizes of buffer pools, the unit of buffering and I/O for each file, the unit of disk space allocation, the degree of parallelism in query evaluation, etc. All operations on records are left open for later definition. Instead of inventing a language in which to specify selection predicates, hash functions, etc., we prefered to pass functions to the appropriate operators which are called when necessary with the right arguments. These *support functions* are explained in more detail in the text. The common theme is that Volcano provides *mechanisms* for query evaluation in order to allow the user of Volcano to experiment with *policies*.

This paper is a general overview, it describes the Volcano system without special attention to any particular aspect. Other papers on Volcano were written for this purpose, e.g., [3,4,5,6,7,8,9,10]. These papers also include experimental performance evaluations of Volcano's algorithms.

In the following section, we briefly review previous work that influenced our design. In Section 3, we provide a more detailed description of Volcano. Section 4 is a discussion of extensibility in the system. Dynamic query evaluation plans and their implementation are described in Section 5. Parallel processing is encapsulated in the *exchange* module described in Section 6. In Section 7, we describe how the *exchange* module can be used to efficiently exploit common subexpressions in a complex query. Section 8 describes alternative strategies and implementations of parallel sorting for very large files. Section 9 contains a summary and our conclusions from this effort.

2. Related Work

Since so many different systems have been developed to process large datasets efficiently, we only survey the systems that have strongly influenced the design of Volcano. The system grew in pieces, with first ideas developed at the University of Wisconsin without a clear design for Volcano. The ideas for dynamic query evaluation plans and for parallel execution were developed at the Oregon Graduate Center [3,11].

Our work is most strongly influenced by WiSS and GAMMA. The Wisconsin Storage System (WiSS), is a record-oriented file system providing heap files, B-tree and hash indices, buffering, and scans with predicates. GAMMA is a software database machine running on a number of general purpose computers as a backend to a UNIX host machine. It was developed on 17 VAX 11/750's connected with each other and the VAX 11/750 host via a 80 Mb/s token ring. Eight GAMMA processors have a local disk device, accessed with WiSS. The disks are accessible only locally, and update and selection operators use only these eight processors. The other, diskless processors are used for join processing. GAMMA extensively uses hash-based algorithms, implemented in such a way that each operator is executed on several (or all) processors and the input stream for each operator is partitioned into disjoint sets according to a hash function [12,13,14]. A detailed description and a preliminary performance evaluation of GAMMA can be found in [13], more information on its performance in [15]. The GAMMA software is currently being ported to an Intel iPSC/2 hypercube with local disk drives.

In early summer of 1987, there was only an impression that some decisions in WiSS [16] and GAMMA [13] were not optimal for performance, generality, or both. For instance, the decisions to protect WiSS's buffer space by copying a data record in or out for each request and to re-request a buffer page for every record during a scan seemed to inflict too much overhead[2]. The desire to design and build a better file system drove the first effort towards Volcano's file system layer.

During the design of the EXODUS storage manager [17], many of these issues were revisited. Lessons learned and tradeoffs explored in these discussions certainly helped in forming the ideas behind Volcano. The development of E [18] influenced the strong emphasis on iterators for query processing.

Finally, a number of conventional (relational) and extensible systems have influenced our design. Without further discussion, we mention Ingres [19], System R [20], GENESIS [21], Starburst [22], Postgres [23], and XPRS [24]. It is interesting to note that independently of our work the Starburst group has also identified the demand-driven iterator paradigm as a suitable basis for an extensible, single process query evaluation architecture [25]. Furthermore, there has been a large amount of research and development in the database machine

[2] This statement only pertains to the original version of WiSS as described in [16]. Both decisions were reconsidered for the version of WiSS used in GAMMA.

area, such that there is an annual international workshop on the topic. Almost all database machine proposals and implementations utilize parallelism in some form. We certainly have learned from this work, in particular from GAMMA, and tried to include these lessons in the design and implementation of Volcano. In particular, we have strived for simplicity and symmetry in the design, mechanisms for well-balanced parallelism, and efficiency in all details.

3. Volcano System Design

In this section, we provide an overview of the modules in Volcano. The file system layer provides record, file, and index operations, including scans with predicates, and buffering. The query processing layer is a library of query processing modules that can be nested to build complex query evaluation trees. This separation can be found in most query evaluation systems, e.g., RSS and RDS in System R [20]. System catalogs or a data dictionary are not included in Volcano since we did not want to commit ourselves to a particular data model. We start our description at the bottom, the file system, and then discuss the query processing modules.

3.1. The File System

Within our discussion of the Volcano file system, we also proceed bottom-up, from devices and memory management through buffer management to data files and B^+-trees.

Volcano stores data on disk devices. Devices are viewed as arrays of fixed-sized pages. Several devices of different sizes can be used concurrently. The page size is a compile-time constant, currently 1 KB. We use UNIX files to simulate devices, similar to WiSS. It is quite straightforward to port Volcano to using raw devices or other primitive OS services[3]. We assume that we can read or write a contiguous array of pages (up to the maximum cluster size, see below) with a single read or write system call.

Page allocation on a device is done using a contiguous bit map. The bit map is sized to contain one bit for each page on the device. The bit map and the volume header are read into buffer space when a device is mounted, and kept there until it is dismounted. Mounting, page allocation, and actual I/O functions are performed by the *physio* module.

[3] In fact, this has been done successfully as reported in the performance section below.

A simple memory manager is used by all other system components, e.g., to allocate buffer space or hash tables. It allocates a large chunk of memory when Volcano is initialized and satisfies requests from this chunk. Request are expressed in number of pages. It uses a bit map and linked lists of small chunks to allocate and deallocate pages.

The buffer manager is one of the more interesting parts of the system. It includes a number of novel features and mechanisms. Buffer pools can be created and destroyed dynamically, and multiple buffer pools can exist concurrently. When a device is mounted, i.e., created or opened, the device is assigned to a buffer pool. When a pool is created, it is created with a *norm size*, e.g., 512 KB, and an *actual size* of 0. The buffer pool grows with each new request that cannot be satisfied with the resident clusters until the actual size reaches the norm size, as will be described shortly.

When pages are requested from the buffer and result in a buffer fault, the buffer manager allocates space for the new page using the memory manager described above. The reason for this strategy is that we wanted to support variable-size chunks, which we call *clusters*, within devices without the problems of *buffer shuffling*, i.e., moving pages around in the buffer in order to find contiguous pages. Buffer shuffling is undesirable for two reasons. First, it is slow and expensive to do. In many current systems, e.g., a DEC microVAX II, memory to memory copying is not much faster than typical disk transfer rates. Second, in order to avoid copying data between the buffer space and query processing algorithms we needed to pin buffer pages, i.e., guarantee the location of a cluster in the buffer until the cluster is explicitly unpinned. We opted against using double indirection as used in the EXODUS storage system [17,26] because it is too complex to implement and introduces unnecessary performance penalties.

A number of investigations concerned database buffer replacement strategies. For an overview, see [27]. Buffer replacement in systems with variable size pages was investigated by Sikeler [28]. To comment only on one, Chou performed an extensive comparison using simulations driven by real database system traces to compare a number of replacement strategies [29,30]. When examining Chou's work, we noticed that most of the benefit of Chou's algorithm DBMIN could be realized by using *hints* from higher level software, e.g., the query

processing routines, for each cluster when a cluster is pinned[4]. Thus, when pinning or unpinning a cluster in the buffer, a hint must be given to the buffer manager whether to *KEEP* or to *TOSS* this cluster. This hint is used by the buffer manager to decide whether to insert the cluster at the top or at the bottom of a LRU list. In effect, the replacement policy is a switched LRU/MRU policy. The LRU policy is implemented in such a way that dirty clusters are paged out slower than clean ones since their replacement cost is higher. Incidentally, this strategy is similar to the *LOVE/HATE* hint used in Starburst's buffer manager [32]. However, while the default in Starburst is *LOVE* (LRU), the default in Volcano is *TOSS* (MRU). The MINIX operating system [33] also inserts pages either at the top or the bottom of a LRU stack for data and indirect pages. The decision is static, however, and does not accept hints from the user level.

When the actual size reaches the norm size, page replacement commences. In the case of a buffer fault, it first is checked whether the cluster at the bottom of the LRU stack is of the same size as the requested cluster. If so, it is replaced. If not, or if there is no free cluster in the stack because the buffer is overcommitted, space for the new cluster is allocated from the memory manager. For this purpose, there are more cluster descriptors in the buffer pool than the norm size of the pool, and the buffer pool can grow beyond its norm size. After the requested cluster is loaded into the buffer, the buffer deallocates clusters from the bottom of the LRU stack until the actual size is below or equal to the norm size, or until the LRU stack is empty.

Files are composed of records, clusters, and extents. Clusters are the unit of I/O and of buffering, as discussed above. The cluster size is fixed for each file individually. Thus, different files on the same device can have different cluster sizes. Records must fit into clusters to avoid the difficulties and inefficiencies of spanning records, i.e., records that are divided over multiple pages. Since clusters can be very large, this is hardly a restriction. At the current time, the record length is also fixed for each file, i.e., all records within one file must have the same length. We realize that this is a severe restriction for supporting complex objects with Volcano and will remedy this problem shortly. Disk space for file is allocated in extents, one primary and up to thirty

[4] Chou's algorithm also uses a hint when a scan is opened about which of a set of access patterns can be expected. Chou's algorithm also includes load control by detecting an overcommitted buffer and the danger of thrashing. However, we believe that load control can easily be added to Volcano simply by estimating hot set sizes [31] and keeping track of their sum.

secondary extents. An extent contains multiple clusters. The extent sizes are declared when a file is created.

Records can be accessed directly using a *record identifier* (RID) or through a scan. A RID consists of device number, page number, cluster size, and record number within the cluster, and it is 8 bytes long. Thus, there can be 256 devices, 2^{32} pages per device, 256 pages per cluster, and 2^{16} records per cluster. The cluster size is included the enable the buffer and I/O modules to perform all necessary operations using only a RID.

There are two modes of record insertion. If a *near* RID is provided with the insertion request, the new record is inserted into the old record's cluster. If this cluster is already full, the default strategy is used. By default, the record is inserted into an arbitrary cluster of the file. For this purpose the file descriptor includes a hint pointing to a cluster that has an open record slot. If no such cluster can be found, a new cluster is appended to the file.

One special file is kept on each device called the *volume table of contents* (VTOC). It implements a single level directory, and is accessed with the standard file and record routines. Originally, this file was intended to serve the purpose of inodes in the UNIX file system [34]. Later on we decided that we also needed a mapping from names to VTOC entries. The name service in Volcano is rather simple. Instead of "going the whole nine yards" and implementing tree-structured directories, we opted for the quicker solution of including name strings in the VTOC entries. However, there is nothing that prevents us from revisiting this decision. The name look-up service is encapsulated in a single procedure, thus changing this aspect of Volcano's design would be a very localized change.

Internally, files are identified by a *closed file descriptor* which is the RID of the file's entry in the VTOC. When a file is opened, the VTOC cluster containing the file's entry is fixed in the buffer pool. An *open file descriptor* is a pointer to a file's entry fixed in the buffer pool. While this is not the safest way to implement handles to open files, it probably is the most efficient one, which is why we chose to use it. The closed file descriptor of the VTOC is kept in the volume header; a device's VTOC is kept open as long as the device is mounted.

There are two interfaces to file scans; one is part of the file system and is the described momentarily; the other is part of the query processing level and is decribed later. The first one has the standard procedures for

file scans *open, next, close, rewind.* The *next* procedure returns the main memory address of the next record. This address is guaranteed (pinned) until the next operation is invoked on the scan. A file scan supports the notion of a *current record* and *current cluster,* the cluster which includes the current record. The current record is the record returned by the last *next* call. The current cluster is fixed in the buffer while the scan is open. Thus, getting the next record within the same cluster does not require calling the buffer manager and can be performed in about 100 instructions. The current record's RID can be extracted from the scan descriptor if it is needed.

It has been pointed out to us that for some applications, bi-directional scans are desirable. Currently, Volcano does not provide reverse scans or a *previous* operation. We believe, however, that such an operation is easy to add if necessary. We did not implement it because Volcano's query processing modules in their current form have no use for reverse scans.

For fast creation of files, scans support an *append* operation. It allocates a new record slot, either in the current cluster or in a new cluster appended to the end of the file, fixes the new cluster in the buffer pool, and returns the new slot's main memory address. Since it can use the current cluster in most cases (without invoking the buffer manager) and does not perform any copying, it is about as fast as the *next* operation.

In addition, there is a *marker* associated with each file scan that can be used with the procedures *set-mark* and *goto-mark.* The former puts the marker on the current record. The latter returns to the marker, i.e., the first call to *next* after *goto-mark* returns the same record as the last call to *next* before the last *set-mark.* This marking mechanism was designed for use by merge join.

Finally, scans in Volcano support optional predicates. Predicates are passed to the scan by means of a function entry point and a typeless pointer which serves as a predicate argument. The predicate function is called by the *next* procedure with the argument and a record address. Only records for which the predicate evaluates to *TRUE* are returned by the scan. Selective scans are the first example of *support functions* mentioned briefly in the introduction. Instead of determining a qualification itself, the scan mechanism relies on a function *imported* from a higher level.

Arguments to support functions can be used in two ways. In compiled scans, i.e., when the predicate evaluation function is available in machine code, they can be used to pass a constant or a pointer to several constants to the predicate function. For example, while the predicate consists of comparing a record field with a string, the comparison function is passed as predicate function while the search string is passed as predicate argument. In interpreted scans, i.e., when a general interpreter is used to evaluate all predicates in a query, they can be used to pass appropriate code for interpretation to the interpreter. The interpreter is given as predicate function. Thus, both interpreted and compiled scans are supported with a single simple and efficient mechanism.

Indices are implemented currently only in the form of top-down B^+-trees. A leaf entry consists of a key and information. The information part typically is a RID, but it could include more or different information. The key and the information can be of any type; a comparison function must be provided to compare keys. The comparison function uses an argument equivalent to the one described for file scan predicates. Currently both the key and the information must be of fixed size. Thus, some space may be wasted for keys of type string as the maximum size must always be reserved and keys may appear repeatedly if the indexed key is not unique. The size of internal index nodes and leaf clusters can be different multiples of a page. We hope that by using larger nodes and leaves we will be able to provide adequate performance even for large keys.

B^+-trees are implemented on top of devices, not on top of files. Therefore, B^+-tree nodes and leaves are not allocated using the extent mechanisms described above for files.

B^+-trees support scans similar to files, including predicates. In addition, B^+-tree scans allow seeking to a particular key, and setting lower and upper bounds. Finally, B^+-trees also support an *append* operation designed for fast loading. Its design assumes that entries are appended in key order. Root-to-leaf traversals are necessary only when appending a new leaf to the B^+-tree. The *append* operation includes parameters for free space in B^+-tree nodes and leaves.

For testing purposes, we implemented *memory devices*, which simulate a disk device in process (virtual) memory. Obviously, their contents are lost when the program that created and used them terminates. I/O on memory devices is translated to copies between the "device" space and the buffer pool.

For intermediate results in query processing (later called ***streams***), we implemented special devices called ***virtual devices***. A virtual device has a volume header and a page allocation map just like a real device. However, data pages of virtual devices only exist in the buffer. As soon as such data pages are unpinned, they disappear and their contents are lost. Files on virtual devices are called ***virtual files***.

In summary, most of Volcano's file system is rather conventional. Two aspects are novel and warrant further performance studies. First, the buffer manager with multiple buffer pools, dynamic growth in case of overcommitment, and its simple but effective replacement hint promises most of the performance advantages of much more complex replacement strategies [35], but requires more thorough experimental analysis. Second, using clusters of different sizes on the same device and in the same buffer pool support small and large files and objects with one simple, efficient mechanism.

3.2. Query Processing

The basic routines above are utilized by the query processing routines to evaluate complex query plans. Queries are expressed as complex algebra expressions; the operators of this algebra are query processing algorithms. We will describe the operations using relational terminology since we hope that this will assist the reader. We want to point out, however, that the operations can be viewed and are implemented as operations on sets of structured objects, and that Volcano is not dependent on assumptions about the structure of such objects[5].

Volcano does not include mechanisms to create such algebra expressions. In a complete database system, these expressions are created in a sequence of steps illustrated in Figure 1. After parsing, a query or command is first verified against the catalogs to determine whether or not the query is legal, e.g., whether it references non-existing attributes. Second, in most relational systems, the query is modified to reflect views and integrity constraints [37]. This step can be seen as a translation from an external level of a database to the conceptual level [38]. Third, the query optimizer maps a query against the conceptual database to a program accessing the physical database. This program is called ***query evaluation plan***, ***access plan***, or simply ***plan***, and can be

[5] In fact, we intend to use Volcano as query processing system for an object-oriented database system [36].

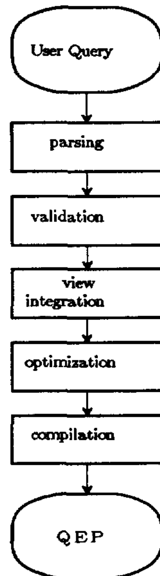

Figure 1: Query Preparation

expressed in many languages, e.g., in ASL in System R [39] or as a set of state records as in Volcano. Since most queries can be mapped to many different, functionally equivalent plans, the optimizer includes an elaborate search engine, e.g., [40,1]. Finally, in most database systems, the query evaluation plan is compiled into machine code, called the *access module*, either by a special purpose compiler or a standard programming language compiler such as the C compiler in the case of Volcano.

Volcano, as mentioned in the introduction, is not a database system. Rather, it provides mechanisms for query evaluation in database systems. Its "programming interface" are state records, currently built manually, but intended to be created by an optimizer. Multiple state records are linked together to form a complex query; each state record represents one algebra operator.

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., (pointers to) 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. Each of these functions uses an argument allowing interpreted or compiled query evaluation, as described earlier for file scan predicates.

Each iterator has an associated state record type. A state record contains the arguments and the state of one operation. For example, the size of a hash table to be allocated in *open* is an argument, and its location is part of the state. All state information of 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.

In queries involving more than one operator (i.e., almost all queries), state records are linked together by means of *input* pointers, as shown in Figure 2. The input pointers are also kept in the state records. They are pointers to a *QEP* structure that includes four pointers. These are to the input operator's state record and to the entry points of the three procedures implementing the operator (*open*, *next*, and *close*).

Using Volcano's standard form of iterators, an operator does not need to know what kind of operator produces its input, or whether its input comes from a complex query tree or from a simple file scan. We call this concept *anonymous inputs* or *streams*. Streams are a simple but powerful abstraction that allows combining any number of operators to evaluate a complex query. Together with the iterator control paradigm, streams represent the most efficient execution model in terms of time (overhead for synchronizing operators) and space (number of records that must reside in memory concurrently) for single process query evaluation.

Calling *open* for the top-most operator results in instantiations for the associated state record's state, e.g., allocation of a hash table, and in *open* calls for all inputs. In this way, all iterators in a query are initiated

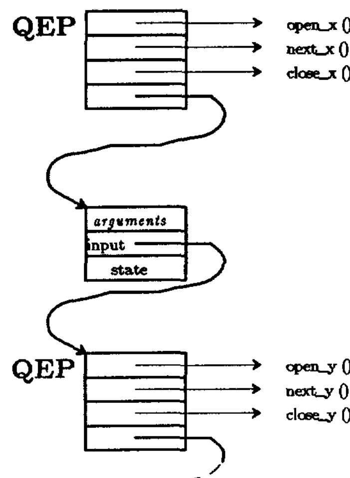

Figure 2: A Query Evaluation Plan

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. The top-most operator calls the *next* procedure of its input if it needs more input data to produce an output record. 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 [18] and the query executor of the Starburst relational database system [25].

A number of query and environment parameters may influence policy decisions during *open*ing a query evaluation plan, e.g., query predicate constants and system load information. Such parameters are passed between all *open* procedures in Volcano with a parameter called *bindings*. This is a typeless pointer that can be used for policy decisions in support functions, e.g., to dynamically determine the degree of parallelism. This parameter is particularly useful in dynamic query evaluation plans, which are described later in a separate section.

The tree-structured query evaluation plan is used to execute queries by demand-driven dataflow. The return value of a *next* operation is, besides an error indicator, 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 about fixing and unfixing records is as follows. Each record pinned in the buffer is *owned* by exactly one operator at any point in time. After receiving a record, the operator can hold on to it for a while, e.g., in a hash table, unfix it, e.g., when a predicate fails, or pass it on to the next operator. Complex operations that create new records, e.g., join, have to fix their output records in the buffer before passing them on, and have to unfix their input records.

A *NEXT_RECORD* structure can point to one record only. All currently implemented query processing algorithms pass complete records between operators, e.g., join creates new, complete records by copying fields from two input records. It can be argued that creating complete new records and passing them between operators is prohibitively expensive. An alternative is to leave original records in the buffer as they were retrieved from the stored data, and compose *NEXT_REC* pairs, triples, etc., for intermediate results. The advantage of this alternative is less memory-to-memory copying. While this may or may not translate to significant performance improvements in single-processor or shared-nothing systems, it may provide substantial gains for shared-

memory multi-processor systems in which the bus as a central resource can become the bottleneck. We are currently working on quantifying the performance impact.

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*.

We could have chosen to implement demand-driven dataflow in a different way. Three methods were considered. First, a central executor module can be used to schedule the progress in each query processing module, basically reinventing coroutines [41]. Each iterator is triggered by the executor and returns control to the executor either when it needs an input record or when it has produced an output record. We felt, however, that writing operators in the coroutine paradigm is less natural than in the iterator paradigm.

Second, instead of one iterator directly calling its input iterator using a function entry point included in a *QEP* structure, we could have used a one-level executor that uses a large *case* statement to decide which iterator to call. We did not see an advantage in this scheme since it introduces additional overhead (two procedure calls for each record passed between iterators instead of one) but still requires that each *QEP* structure have some knowledge (address, name) of its input operator.

Third, we could have used rewriting techniques as proposed by Freytag [42,43,44] to avoid all overhead of procedure calls between iterators and the overhead of invoking support functions. This schemes has three drawbacks, however. First, we felt that it is harder to extend with new algorithms, e.g., for complex object retrieval. Second, in order to remove the overhead of calling support functions, the form and the types of predicates must be incorporated in the rewriting system. Thus, a type system is introduced into Volcano, something we wanted to avoid to keep it more flexible and extensible. Third, using rewriting techniques requires that the entire code be compiled; we felt that we should leave the decision open for the user of Volcano whether predicates and other functions on individual objects are compiled or interpreted.

In summary, we chose to implement demand-driven dataflow by encoding operators as iterator, i.e., with *open*, *next*, and *close* procedures, since this scheme promises generality, extensibility, and low overhead.

3.2.1. Scans, Functional Join, and Filter

The first scan interface was discussed with the file system. The second interface to scans, both file scans and B^+-tree scans, provides an iterator interface suitable for query processing. The *open* procedures open the file or B^+-tree and a scan on it (as described above). The file name or closed file descriptor are given in the state record as are an optional predicate and bounds for B^+-tree scans.

Typically, B^+-tree indices hold keys and RID's in their leaves. In order to use B^+-tree indices, the records in the data file must be retrieved. In Volcano, this look-up operation is split from the B^+-tree scan iterator and is performed by the *functional join* operator. This operator requires a stream of records containing RID's as input and either outputs the data file records retrieved using the RID's or it composes new records from the input records and the retrieved data file records, thus "joining" the B^+-tree entries and their corresponding data records.

We separated B^+-tree scan and functional join for a number of reasons. First, it is not clear that storing data in B^+-tree leaves never is a good idea. This was done, for example, in the original Ingres design [19], but abandoned in the commercial version of Ingres. At times, it is desirable to have other types of information associated with look-up keys, e.g., primary keys in secondary indices in ARBRE [45].

Second, we wished to allow experimenting with manipulation of RID-lists for complex queries. While Blasgen and Eswaran [46] did not find RID-list joins preferable over nested loops join and merge join, their conclusion may not hold for other operations and database systems, e.g., bibliographic search and retrieval systems.

Third, while functional join is currently implemented rather naively, we can make this operation more intelligent. If the functional join operator considers a window of RID's at a time instead of only a single RID, it may be possible to avoid some buffer faults and to save on disk seek operations if the records are retrieved in an organized way. It is not clear whether there is a payoff in this technique or not, but we wanted to leave the option for later experimentation.

Finally, we intend to generalize functional join in a later project to intelligently assemble complex objects consisting of many records in multiple nesting levels [36] A number of object ID's or RID's will be used, selectively based on field values of the input record or of retrieved component objects [9].

Typically, selection predicates are applied by the file or B^+-tree scan operators. Selections that cannot be applied within scans are performed by the *filter* operator. The filter operator can actually perform three functions, switched by the presence or absence of corresponding support functions in the state record. The *predicate* function applies a selection predicate. If this function is present, only records for which the predicate returns *TRUE* are passed on. The *transform* function creates a new record, typically of a new type, from each old record. An example would be a relational projection (without duplicate elimination). More complex examples include compression and decompression, changes in codes and representations, and arithmetic. Finally, the *apply* function is invoked once on each record for the benefit of its side effects. Typical examples are updates and printing. Both the *transform* and *apply* functions have access to the current RID through one of their arguments.

3.2.2. One-to-One Match

The *one-to-one match* operator will probably be among the most frequently used query processing algorithm of Volcano. Originally, it was conceived after the observation that many aggregate functions require a subsequent join on the attributes of the *by*-list (grouping attributes). When using hash-based algorithms, implementing aggregate functions and equi-join in two separate modules results in the same hash table being built twice, once for computing aggregates for each group and once in preparation for the join.

As an example for a query in which the same hash table is build twice, consider a (relational) university database and a request to

find the students who have taken all courses in their major department.

This query can be evaluated using aggregate functions. Focus on the department relation. We will first count the courses for each department, which is an aggregate function grouping on *department*, and then join the output relation (with the attributes *department,number−of−courses*) to the student relation or an intermediate result relation using the *department* attribute. If the aggregate function was performed using a hash table on *department*, the same hash table can be used for the subsequent join.

This observation sparked the definition of one-to-one match. Aggregate functions and join have in common that a tuple is included in the output depending on the result of a comparisons between a pair of tuples.

The opposite is relational division where a tuple is included in the output depending on a set of matches with the tuples of the divisor relation. The main difference between aggregate functions and equi-joins is that the former requires comparing tuples of the same input while the latter requires comparing tuples of two different inputs.

In order to avoid unnecessary memory-to-memory copying and creation of redundant duplicates, it seemed necessary to differentiate between semi-joins and joins that need to compose new intermediate relations. Furthermore, set operations between union-compatible relations can be viewed as variants of join and one-to-one match [47]; therefore, we wanted to implement them with the same algorithm. In our environment, it was a natural choice to implement all these variants using a single module with numerous "bells and whistles." The extensions for set operations such as intersection were trivial, and are outlined below.

The *simple hash join* algorithm as described in [48,12] proceeds in two phases. In the first phase, a hash table is built from one input; it is therefore called the *build phase*. In the second phase, the hash table is probed using tuples from the other input to determine matches and to compose output tuples; it is called the *probe phase*. After the probe phase, the hash table and its entries are discarded. Instead, our one-to-one match operator uses a third phase, which we call the *flush phase*, and which is needed for aggregate functions and some other operations.

Since the one-to-one match operator is also an iterator, the three phases are assigned to the *open*, *next*, and *close* functions. *Open* includes the build phase, while the other two phases are included in the *next* function. Successive invocations of the *next* function automatically switch from the probe phase to the flush phase when the second input is exhausted. Closing the build and probe inputs is delayed until one-to-one match is closed because the file system does not allow closing a file on a virtual device until all its records are unpinned.

The state to be saved between invocations of *next* includes indicators for probe or flush phase, the current build record, and the current probe record. The current build record is required to locate further build records in the same bucket chain which must also matched against the current probe record.

If the one-to-one match operator is used to implement relational join, it proceeds exactly like the simple hash join. The build phase requests input tuples from the *build input* and builds hash bucket chains. The probe

phase determines matches by looping over members of a bucket and produces output tuples where appropriate. The flush phase unfixes the tuples in the hash table in the buffer and deallocates the hash table without producing any output. Support functions are used to calculate a record's hash value, to compare two records, and to compose an output record from two input records.

A semi-join is a join in which records of one input are selected depending on matches with records of the other input. In order to implement semi-join efficiently, the one-to-one match operator preserves rather than copies records of the first input. When performing a semi-join that preserves the probe input, a probe input tuple is matched only until a match is found. The probe tuple is then passed on as output. Thus, memory-to-memory copying is avoided. The module performs a semi-join instead of a join if the support function to compose output tuples is omitted.

The opposite semi-join (which preserves build input tuples) requires a more sophisticated implementation. In our implementation, a bit is initialized to *FALSE* for each tuple in the hash table. When a match is found, this bit is set to *TRUE*. The flush phase scans through all buckets and outputs the tuples for which a match was found.

Finally, outer-join and anti-join can be implemented using Volcano's one-to-one match module. Two additional support functions compose result tuples from build or probe input tuples respectively. Input tuples that do not participate in the natural join (and hence have to be augmented with *NULLs*) are identified using the same mechanisms used for semi-joins.

The build phase can be used to eliminate duplicates or to perform an aggregate function in the build input. Instead of inserting a new tuple into the hash table as in simple hash join, an input tuple is first matched with the tuples in its prospective hash bucket. If a match is found, the new tuple is discarded or its values are aggregated into the existing tuple. The one-to-one match module determines which variant of the algorithm is requested by the presence or absence of the three support functions used to compare two records from the build input, to initialize an aggregation (e.g., set a counter to 0), and to aggregate two records. A Boolean argument switch controls whether build input records are inserted into the hash table, or a new file and new records are created. The latter may be necessary if new fields are required for the aggregation, e.g., a sum *and* and a count

for calculating an average.

The one-to-one match module does not require a probe input; if only an aggregation is required without subsequent join, the absence of the probe input in the state record signals to the module that the probe phase should be skipped.

It is possible to improve the operator's dataflow behavior for duplicate elimination by producing output records immediately when encountered. However, a copy would have to be held back to enable duplicate detection. We felt that allowing upper level operators to proceed while keeping the record pinned in the buffer for duplicate detection would increase buffer contention without significant benefits. Therefore, we did not implement this variant.

While hash tables in main memory are usually quite fast, a severe problem occurs if the build input does not fit in main memory[6]. This situation is called ***hash table overflow.*** There are two ways to deal with hash table overflow. First, if a query optimizer is used and can anticipate overflow, it can be avoided by partitioning the input(s). This ***overflow avoidance*** technique is the basis for the hash join algorithm used in the Grace database machine [49]. Second, overflow files can be used to resolve the problem after it occurs. Several ***overflow resolution*** schemes have been designed and compared [48,12,50,51]. At the current time, we are studying how best to implement hash table overflow avoidance and resolution for the rather complex one-to-one match operator in Volcano.

The extension of the code described so far to set operations started with the observation that the intersection of two union-compatible relations is the same as the natural join of these relations, and can be implemented as semi-join. The union is the (two-sided) outer join of union-compatible relations. The difference and anti-difference of two sets can be computed using special settings of the algorithm's bells and whistles. Finally, Cartesian product can be implemented by matching successfully all possible pairs of records from the two inputs.

---

[6] Notice that if an aggregate function or duplicate elimination is performed on the build input, only the *output* of this operation must fit in main memory. In particular when memory is scarce, aggregating into a new, temporary file pays off since it avoids internal fragmentation in the buffer, i.e., the records are packed densely into clusters.

The second version of one-to-one match is based on sorting. Its two modules are a disk-based merge-sort and the actual merge-join. Opening the *sort* iterator prepares sorted runs for merging. If the number of runs is larger than the maximal fan-in, runs are merged into larger runs until the remaining runs can be merged in a single step. The final merge is performed on demand by the *next* function. If the entire input fits into the sort buffer, it is kept there until demanded by the *next* function.

The *sort* operator has been implemented in such a way that it supports aggregation[7] and duplicate elimination. It can perform these operations early, i.e., while writing temporary files [52,53]. Merge-join has been generalized similarly to hash-join to support semi-join, outer join, anti-join, and set operations. The sort algorithm is explained in detail in [7].

Hash-based one-to-one match, sort-based one-to-one match, and sorting all create new output records and files from input streams, with the exception of semi-join. Since records must be copied in any case, and since copying is always done by invoking support functions, these iterators trivially support record reformatting, e.g., relational projection (without duplicate elimination). For performance reasons, it is very important that projection be included in join operations since copying can contribute significantly to overall query processing costs.

Considering performance results for relational join reported in [13,15,51,14], one might wonder why we did not include bit vector filtering [54] in the one-to-one match operator. We omitted this feature for five reasons. First, it is not obvious that the performance gain would be significant in a shared-memory multiprocessor. Second, it would further complicate the operator, its control logic, code, and support functions. Third, if it is really needed, it is straightforward to build the functionality with two *filter* operators. The first *filter* operator invokes an *apply* support function to build the bit vector. The second operator invokes a *predicate* function to eliminate tuples. The address of the filter in shared memory is passed to the support functions using the arguments. Fourth, detaching bit vector filtering allows using it with merge join. A bit vector filter cannot be included in the merge join operation because the filter must be present before tuples can be eliminated. However, if bit vector building is detached, the filter can be created before or while sorting the first input, and

[7] To be precise, we mean aggregate functions here. For example, the "sum of salaries by department" can be computed by sorting employee records on their department field and adding salaries within each department.

therefore be used to eliminate tuples from the second input before they are passed to the *sort* operator. Thus, bit vector filters can be used in Volcano not only to reduce join costs, including merge join, but also to reduce sort costs. Finally, separating bit vector filtering from one-to-one match allows using it with a single mechanism for both one-to-one match and one-to-many match.

3.2.3. One-to-Many Match

There are two versions of relational division in Volcano, probably the most typical and most frequently used operator of the *one-to-many match* variety. The first version, which we call *naive division*, is based on sorting. The second version, which we call *hash-division*, utilizes two hash tables, one on the divisor and one on the quotient. An exact description of the two algorithms and alternative algorithms based on aggregate functions can be found in [4] along with analytical and experimental performance comparisons and detailed discussions of two partitioning strategies, hash-table overflow, and multi-processor implementations. We are currently studying how to generalize these algorithms in a way comparable with our generalizations of aggregation and join, e.g., for a *majority* function.

4. Extensibility

A number of database research efforts strive for extensibility, e.g., EXODUS, GENESIS, Postgres, Starburst, DASDBS, and others. We believe that Volcano is a very open query evaluation architecture that provides easy extensibility. Let us consider a number of frequently proposed extensions and how they can be accommodated in Volcano.

First, when extending the object type system, e.g., with a new abstract data type (ADT) like *date* or *box*, the Volcano software is not affected at all because it does not provide a type system for objects. All manipulation of and calculation based on individual objects is performed by support functions. Volcano solves some problems in composing complex objects with the functional join operator. Generalizations of this operator are probably necessary for an object-oriented or non-first-normal-form database system, but can be included in Volcano without difficulty.

Second, in order to add new functions on individual objects or aggregate functions, e.g., geometric mean, to the database and query processing system, the appropriate support function is required and passed to a query

processing routine. The reason why Volcano software is not affected by extensions of the functionality on individual objects is that Volcano's software only provides abstractions and implementations for dealing with and sequencing *sets* of objects using streams, whereas the capabilities for individual objects are *imported* in the form of support functions.

Third, in order to incorporate a new access method, e.g., multidimensional indices in form of R-trees [55], appropriate iterators have to be defined. The stream concept is very open; in particular, anonymous inputs shield existing query processing modules and the new iterators from one another.

Fourth, to include a new query processing algorithm in Volcano, e.g., an algorithm for transitive closure or *nest* and *unnest* operations for nested relations, we need to code the algorithm in the iterator paradigm. In other words, we have to write the algorithm in such a way that it provides and uses for its input *open*, *next*, and *close* procedures. After an algorithm has been brought into this form, its integration with Volcano is trivial. In fact, as the Volcano query processing software grew, we did this a number of times.

Extensibility can also be considered in a different context. In the long run, it clearly is desirable to provide a front-end to make using Volcano easier. In particular, we are currently exploring interfacing the EXODUS query optimizer generator [2,1] with Volcano. We are developing a module that "walks" query evaluation plans produced by an optimizer and generates C programs with embedded Volcano code, i.e., state records, support functions, etc. We will use this front-end as a vehicle for experimentation after we have modified the EXODUS software to create *dynamic query evaluation plans*, as outlined in the next section.

In summary, since Volcano is limited in scope, extensibility is provided naturally. In can be argued that this is the case only because Volcano does not address the hard problems in extensibility. We believe that this argument does not hold. Rather, Volcano addresses one subset of the extensibility problems and ignores a different subset. While it provides extensibility of its set of query processing algorithms, it does not provide other essential services like a type system and type checking for the support functions and is therefore not an extensible database system. The Volcano routines assume that the query evaluation plans and their support functions are correct. Their correctness has to be ensured before Volcano is invoked. The significance of Volcano as an extensible query evaluation system is that it provides a simple but very useful set of mechanisms for

efficient query processing and that it can and will be used as a flexible research tool.

5. Dynamic Query Evaluation Plans

In most database systems, a query embedded in a program written in a conventional programming language is optimized when the program is compiled. The query optimizer must make assumptions about the values of the program variables that appear as constants in the query and the data in the database. These assumptions include that the query can be optimized realistically using guessed "typical" values for the program variables and that the database will not change significantly between query optimization and query evaluation. The optimizer must also anticipate the resources that can be committed to query evaluation, e.g., the size of the buffer or the number of processors. The optimality of the resulting query evaluation plan depends on the validity of these assumptions. If a query evaluation plan is used repeatedly over an extended period of time, it is important to determine when reoptimization is necessary. We are working on a scheme in which reoptimization can be avoided by using a new technique called *dynamic query evaluation plans* [3].

Volcano includes a *choose-plan* operator to realize both multi-plan access modules and dynamic plans. This operator provides the same *open-next-close* protocol as the other operators and can therefore be inserted into a query plan at any location. The *open* operation decides which of several equivalent query plans to use and invokes the *open* operation for this input. *Open* calls upon a support function for this policy decision, passing it the *bindings* parameter described above. The *next* and *close* operations simply call the appropriate operation for the input chosen during *open*.

The *choose-plan* operator allows considerable flexibility. If only one *choose-plan* operator is used as top of a query evaluation plan, it implements a multi-plan access module. If multiple *choose-plan* operators are included in a plan, they implement a dynamic query evaluation plan.

The *choose-plan* operator provides significant new freedom in query optimization and evaluation. Since it is compatible with the query processing paradigm, its presence does not affect the other operators at all, and it can be used in a very flexible way. We used the same philosophy when designing and implementing a scheme parallel for query evaluation.

6. Multi-Processor Query Evaluation

The multi-processor implementation grew out of a desire to leverage as much of the effort as possible when the Oregon Graduate Center acquired an eight-processor shared-memory computer system. 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.

This section describes vertical and horizontal parallelism followed by an example, a discussion of variations and variants of the *exchange* operator, an overview of modifications to the file system required for parallel processing, and a comparison of Volcano's exchange operator with GAMMA's mechanisms for parallelism.

6.1. Vertical Parallelism

The first function of exchange is to provide *vertical parallelism* or pipelining between processes. The *open* procedure creates a new process after creating a data structure in shared memory called a *port* for synchronization and data exchange. The child process, created using the UNIX *fork* system call, is an exact duplicate of the parent process. The exchange operator then takes different paths in the parent and child processes.

The parent process serves as the *consumer* and the child process as the *producer* in Volcano. The exchange operator in the consumer process acts as a normal iterator, the only difference from other iterators is that it receives its input via inter-process communication. 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 1 KB which contain 83 *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[8]. Records in packets are fixed in the shared buffer and must be unfixed by a consuming operator.

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

The alert reader has noticed that the exchange module uses a different dataflow paradigm than all other operators. While all other modules are based on demand-driven dataflow (iterators, lazy evaluation), the producer-consumer relationship of exchange uses data-driven dataflow (eager evaluation). There are two very simple reasons for this change in paradigms. First, we intend to use the exchange operator also for ***horizontal parallelism***, to be described below. Second, this scheme removes the need for request messages. Even though a scheme with request messages, e.g., using a semaphore, would probably perform acceptably on a shared-memory machine, we felt that it creates unnecessary control overhead and delays. Since we believe that very high degrees of parallelism and true high-performance query evaluation requires a closely tied network, e.g., a hypercube, of shared-memory machines, we decided to use a paradigm for data exchange that has has been proven to perform well in a shared-nothing database machine [13].

A run-time switch of exchange enables ***flow control*** or ***back pressure*** using an additional semaphore. If the producer is significantly faster than the consumer, the producer may pin a significant portion of the buffer, thus impeding overall system performance. If flow control is enabled, after a producer has inserted a new packet into the port, it must request the flow control semaphore. After a consumer has removed a packet from the port, it releases the flow control semaphore. The initial value of the flow control semaphore, e.g., 4, determines how many packets the producers may get ahead of the consumers.

Notice that flow control and demand-driven dataflow are not the same. One significant difference is that flow control allows some "slack" in the synchronization of producer and consumer and therefore truly overlapped

[8] 83 records is the standard packet size. The actual packet size is an argument in the state record, and can be set between 1 and 255 records.

execution, while demand-driven dataflow is a rather rigid structure of request and delivery in which the consumer waits while the producer works on its next output. The second significant difference is that data-driven dataflow is easier to combine efficiently with horizontal parallelism and partitioning.

6.2. Horizontal Parallelism

There are two forms of horizontal parallelism which we call *bushy parallelism* and *intra-operator* parallelism. In bushy parallelism, different CPU's execute different subtrees of a complex query tree. Bushy parallelism and vertical parallelism are forms of *inter-operator* parallelism. Intra-operator parallelism means that several CPU's perform the same operator on different subsets of a stored dataset or an intermediate result[9].

Bushy parallelism can easily be implemented by inserting one or two exchange operators into a query tree. For example, in order to sort two inputs into a merge-join in parallel, the first or both inputs are separated from the merge-join by an exchange operation. The parent process turns to the second sort immediately after forking the child process that will produce the first input in sorted order. Thus, the two sort operations are working in parallel.

Intra-operator parallelism requires data partitioning. Partitioning of stored datasets is achieved by using multiple files, preferably on different devices. Partitioning of intermediate results is implemented by including multiple queues in a port. If there are multiple consumer processes, each uses its own input queue. The producers use a support function to decide into which of the queues (or actually, into which of the packets being filled by the producer) an output record must go. Using a support function allows implementing round-robin-, key-range-, or hash-partitioning.

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

[9] A fourth form of parallelism is inter-query parallelism, i.e., the ability of a database management system to work on several queries concurrently. In the current version, Volcano does not support inter-query parallelism. A fifth and sixth form of parallelism that can be used for database operations involve hardware vector processing [56] and pipelining in the instruction execution. Since Volcano is a software architecture and following the analysis in [57], we do not consider hardware parallelism further.

within its group. The first action of the master producer is to determine how many slaves are needed by calling an appropriate support function. If the producer operation is to run in parallel, the master producer forks the other producer processes.

Gerber pointed out that such a centralized scheme is suboptimal for high degrees of parallelism [14]. When we changed our initial implementation from forking all producer processes by the master to using a *propagation tree* scheme, we observed significant performance improvements. In such a scheme, the master forks one slave, then both fork a new slave each, then all four fork a new slave each, etc. This scheme has been used very effectively for broadcast communication and synchronization in binary hypercubes.

Even after optimizing the forking scheme, its overhead is not negligible. We are considering using *primed processes*, i.e., processes that are always present and wait for work packets. Primed processes are used in GAMMA [13] and in many commercial database systems. Since the distribution of compiled code for support functions is not trivial in our environment (Sequent Dynix), we delayed this change and plan on using primed processes only when we move to an environment with multiple shared-memory machines[10].

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 the tree and reaches the first exchange operator, the master consumer's *close_exchange* procedure informs all producer processes that they are allowed to close down using the semaphore mentioned above in the discussion on vertical parallelism. If the producer processes are also consumers, the master of the process group informs its producers, etc. In this way, all operators are shut down in

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

an orderly fashion, and the entire query evaluation is self-scheduling.

6.3. An Example

Let us consider an example. Assume a query with four operators, A, B, C, and D such that A calls B's, B calls C's, and C calls D's *open*, *close*, and *next* procedures. Now assume that this query plan is to be run in three process groups, called A, BC, and D. This requires an exchange operator between operators A and B, say X, and one between C and D, say Y. B and C continue to pass records via a simple procedure call to the C's *next* procedure without crossing process boundaries. Assume further that A runs as a single process, A_0, while BC and D run in parallel in processes BC_0 to BC_2 and D_0 to D_3, for a total of eight processes.

A calls X's *open*, *close*, and *next* procedures instead of B's (Figure 3a), without knowledge that a process boundary will be crossed, a consequence of anonymous inputs in Volcano. When X is *open*ed, it creates a port with one input queue for A_0 and forks BC_0 (Figure 3b), which in turn forks BC_1 and BC_2 (Figure 3c). When the BC group *opens* Y, BC_0 to BC_2 synchronize, and wait until the Y operator in process BC_0 has initialized a port with three input queues. BC_0 creates the port and stores its location at an address known only to the BC processes. Then BC_0 to BC_2 synchronize again, and BC_1 and BC_2 get the port information from its location. Next, BC_0 forks D_0 (Figure 3d) which in turn forks D_1 to D_3 (Figure 3e).

When the D operators have exhausted their inputs in D_0 to D_3, they return an ***end-of-stream*** indicator to the driver parts of Y. In each D process, Y flags its last packets to each of the BC processes (i.e., a total of

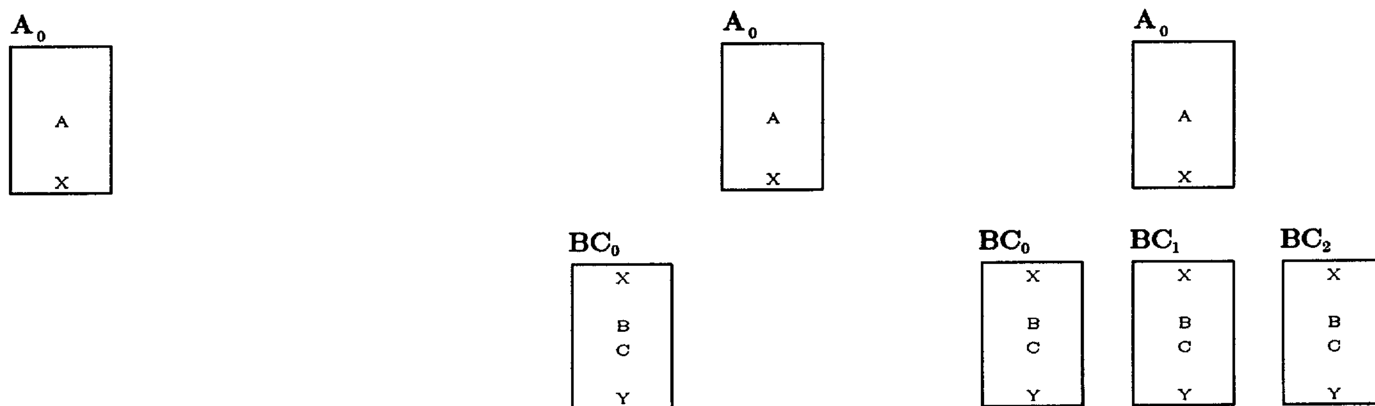

Figure 3a-c. Creating the BC processes.

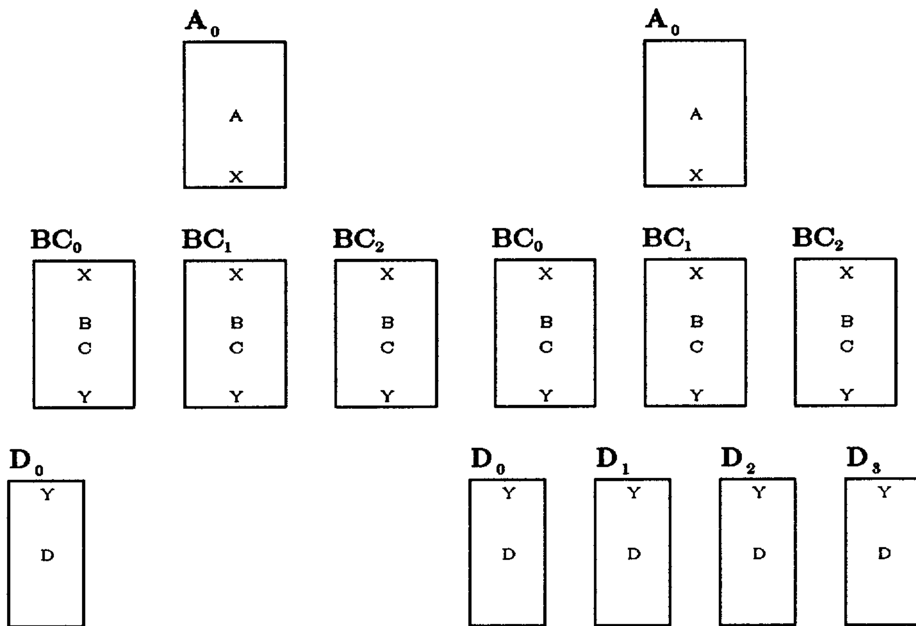

Figure 3d-e. Creating the D processes.

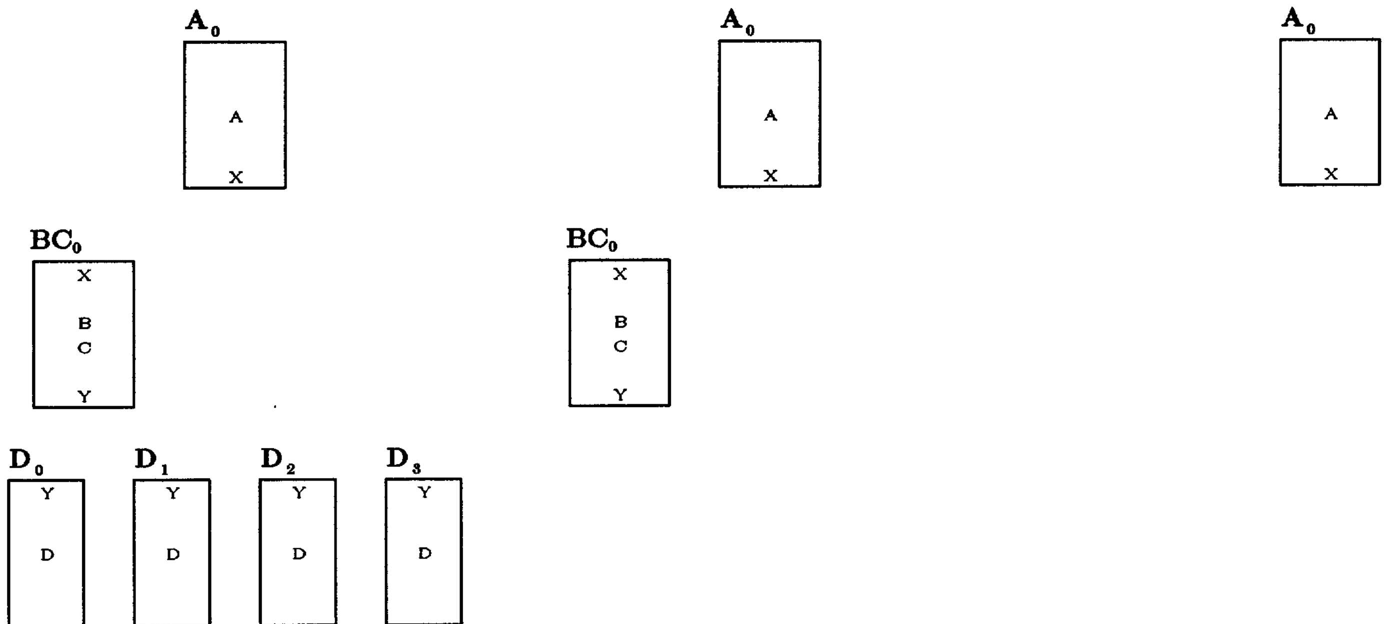

Figure 3f-h. Closing all processes down.

3×4=12 flagged packets) with an *end-of-stream* tag and then waits on a semaphore for permission to *close*. The copies of the Y operator in the BC processes count the number of tagged packets; after four tags (the number of producers or D processes), they have exhausted their inputs, and a call to Y's ***next*** procedure will return an

end-of-stream indicator. In effect, the *end-or-stream* indicator has been propagated from the D operators to the C operators. In due turn, C, B, and then the driver part of X will receive an *end-of-stream* indicator. After receiving three tagged packets, X's *next* procedure in A_0 will indicate *end-of-stream* to A.

When *end-of-stream* reaches the root operator of the query, A, the query tree is *close*d. Closing the exchange operator X includes releasing the semaphore that allows the BC processes to shut down (Figure 3f). The X driver in each BC process *close*s its input, operator B. B *close*s C, and C *close*s Y. Closing Y in BC_1 and BC_2 is an empty operation. When the process BC_0 *close*s the exchange operator Y, Y permits the D processes to shut down by releasing a semaphore. After the processes of the D group have closed all files and deallocated all temporary data structures, e.g., hash tables, they indicate the fact to Y in BC_0 using another semaphore, and Y's *close* procedure returns to its caller, C's *close* procedure, while the D processes terminate (Figure 3g). When all BC processes have *close*d down, X's *close* procedure indicates the fact to A_0 and query evaluation terminates (Figure 3h).

6.4. Variants of the Exchange Operator

For some operations, it is desirable to *replicate* or *broadcast* a stream to *all* consumers. For example, one of the two partitioning methods for hash-division [4] requires that the divisor be replicated and used with each partition of the dividend. Another example is Baru's parallel join algorithm in which one of the two input relations is not moved at all while the other relation is sent through all processors [58]. To support these algorithms, the exchange operator can be directed (by setting a switch in the state record) to send all records to all consumers, after pinning them appropriately multiple times in the buffer pool. Notice that it is not necessary to copy the records since they reside in a shared buffer pool; it is sufficient to pin them such that each consumer can unpin them as if it were the only process using them. After we implemented this feature, parallelizing our hash-division programs using both divisor partitioning and quotient partitioning [4] took only about three hours and yielded not insignificant speedups.

When we implemented and benchmarked parallel sorting [7], we added two more features to *exchange*. First, we wanted to implement a merge network in which some processors produce sorted streams merge concurrently by other processors. Volcano's *sort* iterator can be used to generate a sorted stream. A *merge* iterator

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

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

Second, we implemented a sort algorithm that sorts data randomly partitioned over multiple disks into a range-partitioned file with sorted partitions, i.e., a sorted file distributed over multiple disks. Using the same number of processors and disks, we used two processes per CPU, one to perform the file scan and partition the records and another one to sort them. We realized that creating 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 can migrate. Considering that there is a large cache associated with each CPU, the cache migration adds a significant cost.

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

This mode of operation[11] also makes flow control obsolete. A process runs a producer (and produces input for the other processes) only if it does not have input for the consumer. Therefore, if the producers are in danger of overrunning the consumers, none of the producer operators gets scheduled, and the consumers consume the available records.

6.5. File System Modifications

Clearly, the file system required some modifications to serve several processes concurrently. In order to restrict the extent of such modifications, Volcano currently does not include protection of files and records other than the volume table of contents (VTOC). Furthermore, typically non-repetitive actions like mounting a device must be invoked by the query root process before or after a query is evaluated by multiple processes. The following few paragraphs list the changes that were required in the file system to allow parallel execution.

The *memory* module allocates space in a shared segment rather than a private segment, thus buffer space is also shared among all processes. In order to protect the memory allocation map, a single exclusive lock is held during the short periods of time while the allocation map is searched or updated.

The *physical I/O* module uses two exclusive locks per device. First, *device busy* lock is held while calling UNIX's *lseek*, *read*, and *write* system calls. This is necessary because otherwise two processes could get into a race-condition in which one process's seek operation determines the location of the other process's write. Second, the *map busy* lock protects the free space bit map.

Changes to the *device* module were restricted to protecting the volume table of contents. An exclusive lock is held while an entry is inserted or deleted or while the VTOC is scanned for the descriptor for an external file.

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 query processing. Therefore, the buffer uses a two-level scheme. There is a lock for each buffer pool and one for

[11] Whether exchange forks new producer processes or uses the existing process group to execute the producer operations is a run-time switch.

each descriptor (cluster in the buffer). The buffer pool lock must be held while searching or updating the hash tables and bucket chains. It is never held while doing I/O; thus, it is never held for a long period of time. A 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.

Other buffer managers do not use a pool lock but lock each search bucket and the free chain individually, e.g., the buffer manager of Starburst [59]. The advantage is increased concurrency, while the disadvantage is increased number of locks and lock operations. We are currently working on quantifying this tradeoff for our environment.

If a process finds a requested cluster in the buffer, it uses an atomic test-and-lock operation to lock the descriptor. If this operation fails, the pool lock is released, the operation delayed and restarted. It is necessary to restart the buffer operation including the hash table lookup because the process which holds the lock might be reading or replacing the requested cluster. Therefore, the requesting process must wait to determine the outcome of the prior operation.

Using this restart-scheme for descriptor locks has the additional benefit of avoiding deadlocks. The four conditions for deadlock are *mutual exclusion, hold-and-wait, no preemption,* and *circular wait* [60,61]. Volcano's restart-scheme does not satisfy the second condition.

While the locking scheme avoids deadlocks, it does not avoid convoys [62]. If a process exhausts its CPU time-slice while holding a "popular" exclusive lock, e.g., on a buffer pool, probably all other processes will block in a convoy until the lock-holding process is re-scheduled and releases the lock. However, since we do not use a "fair" scheduling policy that does not allow reacquiring a lock before all waiting processes held and released the lock, we expect that convoys will quickly evaporate [62]. We intend to investigate the special problem of convoys in shared-memory multi-processors further.

It is interesting to note that spin-locks are quite effective in a multi-processor environment. For instance, the pool is locked typically for about 100 instructions. If a process finds the pool locked, it is cheaper to waste 100 instructions spinning than it is to reschedule the CPU and to perform a context switch.

After the buffer manager and the other file system modules were modified to serve multiple processes, it was straightforward to include a *read-ahead/write-behind daemon*. One or more copies of this daemon process are forked when the buffer manager is initialized, and accept work requests on a queue and semaphore similar to the one used within the exchange module. There are three kinds of work requests, the first two are accompanied by a cluster identifier. First, *FLUSH* writes a cluster if it is in the buffer and dirty. Second, *READ_AHEAD* reads a cluster and inserts it at the top of the *LRU* chain. The cluster remains in the buffer using the normal aging process. If it is not fixed and removed from the free list before it reaches the bottom of the free list, it is replaced. Third, a *QUIT* request terminates the daemon.

6.6. Review and Comparison with GAMMA

In summary, the exchange module encapsulates parallel processing in Volcano. Only very few changes had to be made to the buffer manager and the other modules of the file system in order to accommodate parallel execution. The most important properties of the exchange module are that it implements three forms of parallel processing within a single module, that it makes parallel query processing entirely self-scheduling, and that it did not require any changes in the existing query processing modules, thus leveraging significantly the time and effort spent on them and allowing easy parallel implementation of new algorithms.

It might be interesting to compare Volcano and GAMMA [13] query processing in some detail. We only want to point out differences; we do not claim that the design decisions in Volcano are superior to those in GAMMA. First, Volcano runs on a shared-memory multi-processor, whereas GAMMA runs on a shared-nothing architecture. This difference made Volcano easier to implement but will prevent very large configurations due to bus contention. We are currently investigating where the limit is for our software and hardware architecture, and how we can push it as far as possible. Second, GAMMA is a complete system, with query language, system catalogs, query optimization, concurrent transactions, etc., whereas Volcano in its current form only provides mechanisms for single-user query evaluation. Third, Volcano schedules complex queries without the help of a scheduler process. Operators are scheduled and activated top-down using a tree of iterators. In GAMMA, on the other hand, operators are activated bottom-up by a scheduler process associated with the query. Fourth, GAMMA uses only left-deep query trees, i.e., the probing relation in a hash join [48, 12] must be a stored rela-

tion or the result of a selection. In Volcano, both join inputs can be intermediate results. In fact, since Volcano uses anonymous inputs, a join operator has no way of knowing how the inputs were generated. Clearly, the decision whether to use bushy query trees or only left-deep trees has to be made very carefully since the composite resource consumption may lead to thrashing. Fifth, Volcano can execute two or more operators within the same process. In other words, vertical parallelism is optional. In the GAMMA design it is assumed that data have to be repartitioned between operators.

7. Multi-plan Query Evaluation

Frequently, it is desirable to share intermediate results among two queries, or to share two subqueries of the same query. In the query optimization literature, common subexpression detection and global query optimization are popular topics [63,64,65,66,67,68]. However, none of the above query optimizers considers the problem of *scheduling* queries with common subexpressions. In this section, we briefly illustrate how the mechanisms provided in Volcano can be used to execute common subexpressions efficiently.

Consider two queries with a common subexpression. Let us call the common subexpression C and the query-specific tree components A and B respectively. In order to execute A, B, and C, we use one query expression. At the top is an *exchange* module which is used to fork two processes. The top-most operator in each of these process is a *choose-plan* operator which chooses between A and B using the *get_my_id* function provided by the *exchange* module.

This mechanism allow two processes to execute two different plans concurrently. Their common subexpression C is connected to A and B with another *exchange* module. When A and B need input from C, they call on this exchange module. Recall that multiple processes calling on the same exchange module synchronize at this point. Next, a third process is forked to execute C. The *replicate* or broadcast switch described above ensures that all of C's output is delivered to both A and B.

Unfortunately, Volcano's exchange mechanisms do not provide all desired functionality. Since the design of the exchange operators was intended for intra-operator parallelism rather than execution of different plans, the subqueries A and B must obey a significant restriction. In particular, A and B must not include additional exchange modules. If A tried to execute another subquery, say D, in a separate process, the exchange operator

connecting A and D would try to synchronize with the other processes in its process group, namely the process executing B, which would leave A in an infinite wait. If, however, D is a second common subexpression of A and B, and if A and B open C and D in the same order, the exchange operators will connect the four processes correctly.

8. 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, e.g., [69,70,71,72]. In a highly parallel architecture, any centralized component that has to process all data is bound to be a severe bottleneck.

In our 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 this the logical and the physical phase. 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 parallel sorting, we have essentially the same choices. 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[12]. 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[13]. The described 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

---

[12] 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 database 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, e.g., in form of a hypercube, in which data transfer can be a significant cost.

[13] 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 thán a certain fraction of key values in the distribution, e.g., the median is the 50% or 0.5 quantile[14]. 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 [73]. 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 [74].

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. 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*. Differently from all 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 streams correctly.

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

---

[14] 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.

A more detailed description of Volcano's sort iterator and its parallel sort algorithms can be found in [7], along with an experimental performance evaluation.

9. Summary and Conclusions

We have described Volcano, a new query evaluation system that combines compact, extensible, dynamic, and parallel algorithms in a dataflow query evaluation system. Compactness is achieved by focusing on few general algorithms. The one-to-one match operator implements join, semi-join, outer join, anti-join, intersection, union, difference, anti-difference, and Cartesian product. Extensibility is achieved by implementing only one essential abstraction, streams, and by relying on imported support functions for object interpretation and manipulation. The details of streams, e.g., the types and structure of their elements, are not part of the stream definition and its implementation, and can be determined at will.

Dynamic query evaluation plans are a new concept introduced in [3] that allow efficient evaluation of queries with free variables. The ***choose-plan*** operator at the top of a plan or a subplan makes an efficient decision which alternative plan to use when the plan is invoked. Dynamic plans have the potential of increasing the performance of embedded and repetitive queries significantly.

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.

A number of features make Volcano an interesting object of performance studies. First, the LRU/MRU buffer replacement strategy switched by a keep-or-toss hint needs to be evaluated. Second, using clusters of different sizes on a single device and avoiding buffer shuffling by allocating buffer space dynamically instead of statically require careful evaluation. Third, Volcano allows measuring the performance of parallel algorithms and identifying bottlenecks on a shared-memory architecture. Fourth, we will investigate the frequency and the

effect of convoys in multi-processor query evaluation. Fifth, the advantages and disadvantages of a separate scheduler process should be evaluated. Sixth, after data-driven dataflow has been shown to work well on a shared-nothing database machine [13], the combination of demand- and data-driven dataflow should be explored on a network on shared-memory computers.

While Volcano is a working system in its current form, we are considering several improvements. First, Volcano currently does very extensive error detection, including a number of self-tests, but it does not encapsulate errors in ***fail-fast*** modules. It would be desirable to modify all modules such that they have all-or-nothing semantics for all requests. This might prove particularly tricky for the exchange module. Second, for a more complete performance evaluation, Volcano should be enhanced to a multi-user system that allows inter-query parallelism. Third, to make it a complete data manager and query processor, transactions semantics including recovery should be added.

Volcano is the first implemented query evaluation system that combines extensibility and parallelism. We believe that in Volcano we have a powerful tool for database systems research and education. We are making it available for student use, e.g., for implementation and performance studies, and we intend to use it in a number of research projects. First among those are research on the optimization and evaluation of dynamic query evaluation plans [3] and the *REVELATION* project on query optimization in object-oriented database systems with behavioral encapsulation [36].

Acknowledgements

The one-to-one match operators were implemented by Tom Keller of OGC starting from existing hash join, hash aggregate, merge join, and sort code. Dynamic query evaluation plans and the ***choose-plan*** operator were designed and implemented by Karen Ward. The performance measurements were performed by Frank Symonds of Sequent Computer Systems. We are also very much indebted to the members of the GAMMA and EXODUS projects. David DeWitt, Leonard Shapiro, and David Maier contributed to the quality and clarity of the exposition with insightful comments. Jim Gray, Mike Stonebraker, and Marguerite Murphy gave very helpful comments on earlier drafts of this paper.

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. G. Graefe, "Relational Division: Four Algorithms and Their Performance," *Proceedings of the IEEE Conference on Data Engineering*, pp. 94-101 (February 1989).
5. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-007)(June 1989).
6. G. Graefe, F. Symonds, and G. Kelley, "Shared-Memory Dataflow Query Processing in Volcano," *Oregon Graduate Center, Computer Science Technical Report*, (89-010)(June 1989).
7. G. Graefe, "Parallel External Sorting in Volcano," *Oregon Graduate Center, Computer Science Technical Report*, (89-008)(June 1989).
8. T. Keller and G. Graefe, "The One-to-One Match Operator of the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-009)(June 1989).
9. G. Graefe, "Set Processing and Complex Object Assembly in Volcano and the REVELATION Project," *Oregon Graduate Center, Computer Science Technical Report*, (89-013)(June 1989).
10. G. Graefe and S.S. Thakkar, "Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor," *in preparation*, (July 1989).
11. 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).
12. D.J. DeWitt and R.H. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the Conference on Very Large Data Bases*, pp. 151-164 (August 1985).
13. 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*, pp. 228-237 (August 1986).
14. R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," *Ph.D. Thesis*, University of Wisconsin, (October 1986).
15. D.J. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A Performance Analysis of the GAMMA Database Machine," *Proceedings of the ACM SIGMOD Conference*, pp. 350-360 (June 1988).
16. 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) pp. 943-962 (October 1985).
17. M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita, "Object and File Management in the EXODUS Extensible Database System," *Proceedings of the Conference on Very Large Data Bases*, pp. 91-100 (August 1986).
18. 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).
19. M. Stonebraker, E. Wong, P. Kreps, and G.D. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems* **1**(3) pp. 189-222 (September 1976).
20. M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson, "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems* **1**(2) pp. 97-137 (June 1976).

21. D.S. Batory, "GENESIS: A Project to Develop an Extensible Database Management System," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems,* pp. 207-208 (September 1986).

22. P. Schwarz, W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst Database System," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems,* pp. 85-92 (September 1986).

23. M. Stonebraker and L.A. Rowe, "The Design of POSTGRES," *Proceedings of the ACM SIGMOD Conference,* pp. 340-355 (May 1986).

24. M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," *Proceedings of the Conference on Very Large Databases,* pp. 318-330 (August 1988).

25. 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).

26. M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuh, E.J. Shekita, and S. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview," in *Readings on Object-Oriented Database Systems,* ed. D. Maier and S. Zdonik,Morgan Kaufman, San Mateo, CA. (1989).

27. W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems* **9**(4) pp. 560-595 (December 1984).

28. A. Sikeler, "VAR-PAGE-LRU: A Buffer Replacement Algorithm Supporting Different Page Sizes," *Lecture Notes in Computer Science* **303** p. 336 Springer Verlag, (April 1988).

29. H.T. Chou, "Buffer Management of Database Systems," *Ph.D. Thesis,* University of Wisconsin, (May 1985).

30. H.T. Chou and D.J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proceedings of the Conference on Very Large Data Bases,* pp. 127-141 (August 1985).

31. G.M. Sacco and M. Schkolnik, "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," *Proceeding of the Conference on Very Large Data Bases,* pp. 257-262 (September 1982).

32. J. McPherson, M. Lee, and B. Lindsay, *Personal Communication.* December 1988.

33. A.S. Tanenbaum, *Operating Systems – Design and Implementation,* Prentice Hall, Englewood Cliffs, NJ (1987).

34. M.K. McKusick, W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX," *ACM Transactions of Computer Systems* **2**(3) pp. 181-197 (August 1984).

35. H.T. Chou, *Personal Communication.* May 1988.

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

37. M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," *Proceedings of the ACM SIGMOD Conference,* (June 1975).

38. ANSI, "The ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems," *AFIPS Press,* (1977).

39. R.A. Lorie and J.F. Nilsson, "An Access Specification Language for a Relational Database Management System," *IBM Journal of Research and Development* **23**(3) pp. 286-298 (May 1979).

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

41. M. Conway, "A Multiprocessor System Design," *Proceedings of the AFIPS Fall Joint Computer Conference,* pp. 139-146 (1963).

42. J.C. Freytag, "Translating Relational Queries into Iterative Programs," *Ph.D. Thesis,* Harvard University, (September 1985).

43. J.C. Freytag and N. Goodman, "Rule-Based Transformation of Relational Queries into Iterative Programs," *Proceedings of the ACM SIGMOD Conference,* pp. 206-214 (May 1986).

44. J.C. Freytag and N. Goodman, "Translating Aggregate Queries into Iterative Programs," *Proceeding of the Conference on Very Large Data Bases,* pp. 138-146 (August 1986).

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

46. M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases," *IBM Systems Journal* **16**(4)(1977).

47. *This was observed in the Peterlee Relational Test Vehicle project; unfortunately, the author was unable to locate the exact reference..*

48. D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD Conference,* pp. 1-8 (June 1984).

49. S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An Overview of The System Software of A Parallel Relational Database Machine GRACE," *Proceeding of the Conference on Very Large Data Bases,* pp. 209-219 (August 1986).

50. R.H. Gerber, "The Hash-Partitioned Algorithms," *Preliminary Proposal,* University of Wisconsin, (January 1985).

51. D. DeWitt and D. Schneider, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proceedings of the ACM SIGMOD Conference,* p. 110 (May-June 1989).

52. A. Klug, "Access Paths in the 'ABE' Statistical Query Facility," *Proceedings of the ACM SIGMOD Conference,* pp. 161-173 (June 1982).

53. 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).

54. E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *ACM Transactions on Database Systems* **4**(1) pp. 1-29 (March 1979).

55. A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proceedings of the ACM SIGMOD Conference,* pp. 47-57 (June 1984).

56. S. Torii, K. Kojima, Y. Kanada, A. Sakata, S. Yoshizumi, and M. Takahashi, "Accelerating Nonnumerical Processing by an Extended Vector Processor," *Proceedings of the IEEE Conference on Data Engineering,* pp. 194-201 (February 1988).

57. 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,* Springer, (1983).

58. C.K. Baru, O. Frieder, D. Kandlur, and M. Segal, "Join on a Cube: Analysis, Simulation, and Implementation," *Proceedings of the 5th International Workshop on Database Machines,* (1987).

59. B. Lindsay, *Personal Communication.* February 1989.

60. E.G. Coffman, Jr., M.J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys* **3**(2) pp. 67-78 (June 1971).

61. R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys* **4**(3) pp. 179-196 (September 1972).

62. M. Blasgen, J.N. Gray, M. Mitoma, and T. Price, "The Convoy Phenomenon," *Operating Systems Review* **13**(2) pp. 20-25 (April 1979).

63. M. Jarke, "Common Subexpression Isolation in Multiple Query Optimization," pp. 191-205 in *Query Processing in Database Systems*, ed. W. Kim, D.S. Reiner, and D.S. Batory,Springer, Berlin (1985).

64. W. Kim, "Global Optimization of Relational Queries: A First Step," pp. 206-216 in *Query Processing in Database Systems*, ed. W. Kim, D.S. Reiner, and D.S. Batory,Springer, Berlin (1985).

65. J. Park and A. Segev, "Using Common Subexpressions To Optimize Multiple Queries," *Proceedings of the IEEE Conference on Data Engineering*, pp. 311-319 (February 1988).

66. A. Rosenthal and U. Chakravarthy, "Anatomy of a Modular Multiple Query Optimizer," *Proceedings of the Conference on Very Large Databases*, pp. 230-239 (August 1988).

67. K. Satoh, M. Tsuchida, F. Nakamura, and K. Oomachi, "Local and Global Query Optimization Mechanisms for Relational Databases," *Proceedings of the Conference on Very Large Data Bases*, pp. 405-417 (August 1985).

68. T.K. Sellis, "Multiple-Query Optimization," *ACM Transaction on Database Systems* **13**(1) pp. 23-52 (March 1988).

69. D. Bitton Friedland, "Design, Analysis, and Implementation of Parallel External Sorting Algorithms," *Computer Sciences Technical Report* **464**University of Wisconsin, (January 1982).

70. 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).

71. M. Beck, D. Bitton, and W.K. Wilkinson, "Sorting Large Files on a Backend Multiprocessor," *Department of Computer Science Technical Report*, (March 1986).

72. 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).

73. 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).

74. G. Graefe, "Selectivity Estimation Using Moments and Density Functions," *Oregon Graduate Center, Computer Science Technical Report*, (87-012)(November 1987).

Abstract 1
1. Introduction 1
2. Related Work 2
3. Volcano System Design 4
3.1. The File System 4
3.2. Query Processing 10
3.2.1. Scans, Functional Join, and Filter 15
3.2.2. One-to-One Match 16
3.2.3. One-to-Many Match 21
4. Extensibility 21
5. Dynamic Query Evaluation Plans 23
6. Multi-Processor Query Evaluation 24
6.1. Vertical Parallelism 24
6.2. Horizontal Parallelism 26
6.3. An Example 28
6.4. Variants of the Exchange Operator 30
6.5. File System Modifications 32
6.6. Review and Comparison with GAMMA 34
7. Multi-plan Query Evaluation 35
8. Parallel Sorting 36
9. Summary and Conclusions 39
Acknowledgements 40