Volcano—An Extensible and Parallel Query Evaluation System

Goetz Graefe

Abstract—**To investigate the interactions of extensibility and parallelism in database query processing, we have developed a new dataflow query execution system called Volcano. The Volcano effort provides a rich environment for research and education in database systems design, heuristics for query optimization, parallel query execution, and resource allocation.**

Volcano uses a standard interface between algebra operators, allowing easy addition of new operators and operator implementations. Operations on individual items, e.g., predicates, are imported into the query processing operators using *support functions*. The semantics of support functions is not prescribed; any data type including complex objects and any operation can be realized. Thus, Volcano is *extensible* with new operators, algorithms, data types, and type-specific methods.

Volcano includes two novel *meta-operators*. The *choose-plan* meta-operator supports *dynamic query evaluation plans* that allow delaying selected optimization decisions until run-time, e.g., for embedded queries with free variables. The *exchange* meta-operator supports *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.

All operators, with the exception of the exchange operator, have been designed and implemented in a single-process environment, and parallelized using the exchange operator. Even operators not yet designed can be parallelized using this new operator if they use and provide the interator interface. Thus, the issues of data manipulation and parallelism have become *orthogonal*, making Volcano the first implemented query execution engine that effectively combines extensibility and parallelism.

Index Terms—**Dynamic query evaluation plans, extensible database systems, iterators, operator model of parallelization, query execution.**

I. Introduction

IN ORDER to investigate the interactions of extensibility, efficiency, and parallelism in database query processing and to provide a testbed for databse systems research and education, we have designed and implemented a new query evaluation system called Volcano. It is intended to provide an experimental vehicle for research into query execution techniques and query optimization optimization heuristics rather than a database system ready to support applications. It is not a complete database system as it lacks features such as a user-friendly query language, a type system for instances (record definitions), a query optimizer, and catalogs. Because of this focus, Volcano is able to serve as an experimental vehicle for a multitude of purposes, all of them open-ended, which results in a combination of requirements that have not been integrated in a single system before. First, it is modular and extensible to enable future research, e.g., on algorithms, data models, resource allocation, parallel execution, load balancing, and query optimization heuristics. Thus, Volcano provides an infrastructure for experimental research rather than a final research prototype in itself. Second, it is simple in its design to allow student use and research. Modularity and simplicity are very important for this purpose because they allow students to begin working on projects without an understanding of the entire design and all its details, and they permit several concurrent student projects. Third, Volcano's design does not presume any particular data model; the only assumption is that query processing is based on transforming sets of items using parameterized operators. To achieve data model independence, the design very consistently separates set processing *control* (which is provided and inherent in the Volcano operators) from *interpretation and manipulation* of data items (which is imported into the operators, as described later). Fourth, to free algorithm design, implementation, debugging, tuning, and initial experimentation from the intricacies of parallelism but to allow experimentation with parallel query processing. Volcano can be used as a single-process or as a parallel system. Single-process query evaluation plans can already be parallelized easily on shared-memory machines and soon also on distributed-memory machines. Fifth, Volcano is realistic in its query execution paradigm to ensure that students learn how query processing is really done in commercial database products. For example, using temporary files to transfer data from one operation to the next as suggested in most textbooks has a substantial performance penalty, and is therefore used in neither real database systems nor in Volcano. Finally, Volcano's means for parallel query processing could not be based on existing models since all models explored to date have been designed with a particular data model and operator set in mind. Instead, our design goal was to make parallelism and data manipulation *orthogonal*, which means that the mechanisms for parallel query processing are independent of the operator set and semantics, and that all operators, including new

Manuscript received July 26, 1990; revised September 5, 1991. This work was supported in part by the National Science Foundation under Grants IRI-8996270, IRI-8912618, and IRI-9006348, and by the Oregon Advanced Computing Institute (OACIS).

The author is with the Computer Science Department, Portland State University, Portland, OR 97207-0751.

IEEE Log Number 9211308.

ones, could be designed and implemented independently of future parallel execution.

Following a design principle well established in operating systems research but not exploited in most database system designs, Volcano provides *mechanisms* to support *policies*. Policies can be set by a human experimenter or by a query optimizer. The separation of mechanisms and policies has contributed to the extensibility and modularity of modern operating systems, and may make the same contribution to extensible database systems. We will return to this separation repeatedly in this paper.

Since its very purpose is to allow future extensions and research, Volcano is continuously being modified and extended. Among the most important recent extensions were the design and implementation of two *meta-operators*. Both of them are not only new operators but also embody and encapsulate new concepts for query processing. They are meta-operators since they do not contribute to data manipulation, selection, derivation, etc., but instead provide additional control over query processing that cannot be provided by conventional operators like file scan, sort, and merge join. The *choose-plan* operator implements *dynamic query evaluation plans*, a concept developed for queries that must be optimized with incomplete information [17]. For example, it is not possible to reliably optimize an embedded query if one of the constants in the query predicate is actually a program variable and therefore unknown during compilation and optimization. Dynamic plans allow preparation for multiple equivalent plans, each one optimal for a certain range of actual parameter values. The *choose-plan* operator selects among these plans at runtime while all other operators in Volcano's operator set (present or future) are entirely oblivious to the presence and function of the *choose-plan* operator.

The second meta-operator, the *exchange* operator, implements and controls parallel query evaluation in Volcano. While operators can exchange data without the *exchange* operator, in fact within processes as easily as a single procedure call, this new operator exchanges data across process and processor boundaries. All other operators are implemented and execute without regard to parallelism; all parallelism issues like partitioning and flow control are encapsulated in and provided by the *exchange* operator. Thus, data manipulation and parallelism are indeed orthogonal in Volcano [20]. Beyond the cleanliness from a software engineering point of view, it is also very encouraging to see that this method of parallelizing a query processing engine does indeed allow linear or near-linear speedup [18].

This paper is a general overview describing the overall goals and design principles. Other articles on Volcano were written on special aspects of the system, e.g., [16]–[21], [25], [26]. These articles also include experimental performance evaluations of Volcano's techniques and algorithms, in particular [18], [21].

The present paper is organized as follows. In the following section, we briefly review previous work that influenced Volcano's design. A detailed description of Volcano follows in Section III. Section IV contains a discussion of extensibility in the system. Dynamic query evaluation plans and their implementation are described in Section V. Parallel processing encapsulated in the *exchange* module is described in Section VI. Section VII contains a summary and our conclusions from this effort.

II. Related Work

Since so many different systems have been developed to process large datesets efficiently, we only survey the systems that have significantly influenced the design of Volcano. Our work has been influenced most strongly by WiSS, GAMMA, and EXODUS. The Wisconsin Storage System (WiSS) [10] is a record-oriented file system providing heap files, B-tree and hash indexes, buffering, and scans with predicates. GAMMA [11] is a software database machine running on a number of general-purpose CPU's 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 had a local disk device, accessed using WiSS. The disks were accessible only locally, and update and selection operators used only these eight processors. The other, diskless processors were used for join processing. Recently, the GAMMA software has been ported to an Intel iPSC/2 hypercube with 32 nodes, each with a local disk drive. GAMMA uses hash-based algorithms extensively, implemented in such a way that each operator is executed on several (usually all) processors and the input stream for each operator is partitioned into disjoint sets according to a hash function.

The limited data model and extensibility of GAMMA led to the search for a more flexible but equally powerful query processing model. The operator design used in the GAMMA database machine software gives each operator control within its own process, leaving it to the networking and operating system software to synchronize multiple operators in producer–consumer relationships using flow-control mechanisms. This design, while working extremely well in GAMMA, does not lend itself to single-process query evaluation since multiple loci of control, i.e., multiple operators, cannot be realized inside a single process without special pseudo-multiprocess mechanisms such as threads. Therefore, GAMMA's operator and data transfer concepts are not suitable for an efficient query processing engine intended for both sequential and parallel query execution.

EXODUS [7] is an extensible database system with some components followiong the "tool-kit" approach, e.g., the optimizer generator [13], [14] and the E database implementation language [27], [28], and other components built as powerful but fixed components, e.g., the storage manager [5]. Originally, EXODUS was conceived to be data-model-independent, i.e., it was supposed to support a wide variety of data models, but later a novel, powerful, structurally object-oriented data model called Extra was developed [6]. The concept of data model

independence as first explored in EXODUS has been retained in the Volcano project and the design and implementation of its software. During the design of the EXODUS storage manager, many storage and access issues explored in WiSS and GAMMA were revisited. Lessons learned and trade-offs explored in these discussions certainly helped in forming the ideas behind Volcano. The design and development of E influenced the strong emphasis on iterators for query processing.

A number of further conventional (relational) and extensible systems have influenced our design. Ingres [32] and System R [9] have probably influenced most database systems, in particular their extensible follow-on projects Starburst [23] and Postgres [35]. It is interesting to note that independently of our work the Starburst group has also identified the demand-driven interator paradigm as a suitable basis for an extensible single-process query evaluation architecture after using it successfully in the System R relational system, but as yet has not been able to combine extensibility with parallelism. GENESIS [1] early on stressed the importance of uniform operator interfaces for extensibility and software reusability.

XPRS has been the first project aiming to combine extensibility with parallelism [34]. Its basic premise is to implement Postgres on top of RAID disk arrays and the Sprite operating system. XPRS and GAMMA basically differ in four ways. First, GAMMA supports a purely relational data model while XPRS supports an extensible relational model, Postgres. Second, GAMMA's main form of parallelism is intra-operator parallelism based on partitioned data sets. XPRS, on the other hand, will rely on bushy parallelism, i.e., concurrent execution of different subtrees in a complex query evaluation plan. Third, GAMMA relies heavily on hashing for joins and aggregations whereas XPRS will have a mainly sort-based query processing engine [33]. Fourth, GAMMA is built on the premise that distributed memory is required to achieve scalable linear speed-up while XPRS is being implemented on a shared-memory machine.

Both XPRS and Volcano combine parallelism and extensibility, but XPRS is a far more comprehensive project than Volcano. In particular, XPRS includes a data model and a query optimizer. On the other hand, Volcano is more extensible precisely because it does not presume a data model. Therefore, Volcano could be used as the query processing engine in a parallel extensible-relational system such as XPRS. Moreover, it will eventually include a data-model-independent optimizer generator to form a complete query processing research environment.

III. Volcano System Design

In this section, we provide an overview of the design of Volcano. At the current time, Volcano is a library of about two dozen modules with a total of about 15 000 lines of C code. These modules include a file system, buffer management, sorting, B^+-trees, and two algorithms each (sort- and hash-based) for natural join, semi-join, all three outer joins, anti-joint, aggregation, duplicate elimination, union, intersection, difference, anti-difference, and relational division. Moreover, two modules implement dynamic query evaluation plans and allow parallel processing of all algorithms listed above.

All operations on individual records are deliberately left open for later definition. Instead of inventing a language in which to specify selection predicates, hash functions, etc., functions are passed to the query processing operators to be called when necessary with the appropriate arguments. These *support functions* are described later in more detail. One common and repeating theme in the design of Volcano is that it provides *mechanisms* for query evaluation to allow selection of and experimentation with *policies*. The separation of mechanisms and policies is a very well-known and well-established principle in the design and implementation of operating systems, but it has not been used as extensively and consistently in the design and implementation of database systems. It has contributed significantly to the extensibility and modularity of modern operating systems, and may make the same contribution to extensible database systems.

Currently, Volcano consists of two layers, the file system layer and the query processing layer. The former provides record, file, and index operations including scans with optional predicates, and buffering; the latter is a set of query processing modules that can be nested to build complex query evaluation trees. Fig. 1 identifies Volcano's main modules. This separation can be found in most query evaluation systems, e.g., RSS and RDS in System R [9] and Core and Corona in Starburst [23]. System catalogs or a data dictionary are not included in Volcano since the system was designed to be extensible and independent from any particular data model. We start our description at the bottom, the file system, and then discuss the query processing modules.

A. The File System

Within our discussion of the Volcano file system, we also proceed bottom-up, from buffer management to data files and indices. The existing facilities are meant to provide a backbone of a query processing system, and are designed such that modifications and additions can easily be accomplished as the need arises.

The *buffer manager* is the most interesting part of the file system. Because buffer management is performance-critical in any database system, the Volcano buffer manager was designed to include mechanisms that can be used most effectively and efficiently in a large variety of contexts and with a wide array of policies. In consequence, its features include multiple buffer pools, variable-length units of buffering that are called *clusters* in Volcano, and replacement hints from the next higher software level.

The buffer manager's hint facility is an excellent example of Volcano's design principle to implement *mechanisms* to support multiple *policies*. The buffer manager only provides the mechanisms, i.e., pinning, page re-

Execute (driver)	Exchange	Choose-Plan
Hash One-to-One Match	Sort One-to-One Match	Sort
Hash One-to-Many Match	Sort One-to-Many Match	Index Maintenance
Scans	Functional Join	Filter
Files & Records	Devices	Indices (B^+-trees)
Physical I/O	Buffer Manager	Memory Manager

Fig. 1. Volcano's main modules.

placement, and reading and writing disk pages, while the higher level software determines the policies depending on data semantics, importance, and access patterns. It is surprising that database buffer managers derive replacement decisions from *observed* reference behavior in spite of the fact that this behavior is generated by higher level database software and thus known and foreseeable in advance within the same system, albeit different subcomponents.

Files are composed of records, clusters, and extents. Since file operations are invoked very frequently in any database system, all design decisions in the file module have been made to provide basic functionality with the highest attainable performance. A cluster, consisting of one or more pages, is the unit of *I/O* and of buffering, as discussed above. The cluster size is set for each file individually. Thus, different files on the same device can have different cluster sizes. Disk space for files is allocated in physically contiguous extents, because extents allow very fast scanning without seeks and large-chunk read-ahead and write-behind.

Records are identified by a *record identifier* (RID), and can be accessed directly using the RID. For fast access to a large set of records, Volcano supports not only individual file and record operations but also *scans* that support read-next and append operations. There are two interfaces to file scans; one is part of the file system and is described momentarily; the other is part of the query processing level and is described later. The first one has the standard procedures for file scans, namely *open*, *next*, *close*, and *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. Thus, getting the next record within the same cluster does not require calling the buffer manager and can be performed very efficiently.

For fast creation of files, scans support an *append* operation. It allocates a new record slot, and returns the new slot's main memory address. It is the caller's responsibility to fill the provided record space with useful information, i.e., the *append* routine is entirely oblivious to the data and their representation.

Scans also support optional *predicates*. The predicate function is called by the *next* procedure with the argument and a record address. 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 predicate function *imported* from a higher level.

Support functions are passed to an operation as a function entry point and a typeless pointer that serves as a predicate argument. Arguments to support functions can be used in two ways, namely in compiled and interpreted query execution. In compiled scans, i.e., when the predicate evaluation function is available in macvhine code, the argument can be used to pass a constant or a pointer to several constants to the predicate function. For example, if 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 query, they can be used to pass appropriate code to the interpreter. The interpreter's entry point is given as predicate function. Thus, both interpreted and compiled scans are supported with a single simple and efficient mechanism. Volcano's use of support functions and their arguments is another example for a mechanism that leaves a policy decision, in this case whether to use compiled or interpreted scans, open to be decided by higher level software.

Indices are implemented currently only in the form of B+-trees with an interface similar to files. 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 scan predicates. Permitting any information in the leaves gives more choices in physical database design. It is another example of Volcano providing a mechanism to allow a multitude of designs and usage policies. B+-trees support scans similar to files, including predicates and append operations for fast loading. In addition, B^+-tree scans allow seeking to a particular key, and setting lower and upper bounds.

For intermediate results in query processing (later called *streams*), Volcano uses special devices called *virtual devices*. The difference between virtual and disk devices is that 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. Thus, Volcano uses the same mechanisms and function calls for permanent and intermediate data sets, easing implementation of new operators significantly.

In summary, much of Volcano's file system is conventional in its goals but implemented in a flexible, efficient, and compact way. The file system supports basic abstractions and operations, namely devices, files, records, B^+-trees, and scans. It provides mechanisms to access these objects, leaving many policy decisions to higher level software. High performance was a very important goal in the design and implementation of these mechanisms since performance studies and parallelization only make sense if the underlying mechanisms are efficient. Furthermore, research into implementation and performance trade-offs for extensible database systems and new data models is only relevant if an efficient evaluation platform is used.

B. Query Processing

The file system routines described above are utilized by the query processing routines to evaluate complex queries. Queries are expressed as *query plans* or algebra expressions; the operators of this algebra are query processing algorithms and we call the algebra an *executable algebra* as opposed to *logical algebras*, e.g., relational algebra. We will describe the operations using relational terminology hoping that this will assist the reader. We must point out, however, that the operations can be viewed and are implemented as operations on sets of objects, and that Volcano does not depend on assumptions about the internal structure of such objects. In fact, we intend to use Volcano for query processing in an object-oriented database system [15]. The key to this use of Volcano is that set processing and interpretation of data items are separated.

In Volcano, all algebra operators are implemented as *iterators*, i.e., they support a simple *open-next-close* protocol. Basically, iterators provide the iteration component of a loop, i.e., initialization, increment, loop termination condition, and final housekeeping. These functions allow "iteration" over the results of any operation similar to the iteration over the result of a conventional file scan. Associated with each iterator is a *state record* type. A state record contains arguments, e.g., the size of a hash table to be allocated in the *open* procedure, and *state*, e.g., the location of a hash table. All state information of an iterator is kept in its state record and there are no "static" variables; thus, an algorithm may be used multiple times in a query by including more than one state record in the query.

All manipulation and interpretation of data objects, e.g., comparisons and hashing, is passed to the iterators by means of pointers to the entry points of appropriate *support functions*. Each of these support functions uses an argument allowing interpreted or compiled query evaluation, as described earlier for file scan predicates. Without the support functions, Volcano's iterators are empty *algorithm shells* that cannot perform any useful work. In effect, the split into algorithm shells and support functions separates control and iteration over sets from interpretation of records or objects. This separation is one of the cornerstones of Volcano's data model independent and extensibility, which will be discussed in Section IV.

Iterators can be nested and then operate similarly to coroutines. State records are linked together by means of *input* pointers. The input pointers are also kept in the state records. Fig. 2 shows two operators in a query evaluation plan. Purpose and capabilities of the *filter* operator will be discussed shortly; one of its possible functions is to print items of a stream using a function passed to the filter operator as one of its arguments. The structure at the top gives access to the functions as well as to the state record. Using a pointer to this structure, the filter functions can be called and their local state can be passed to them as a procedure argument. The functions themselves, e.g.,

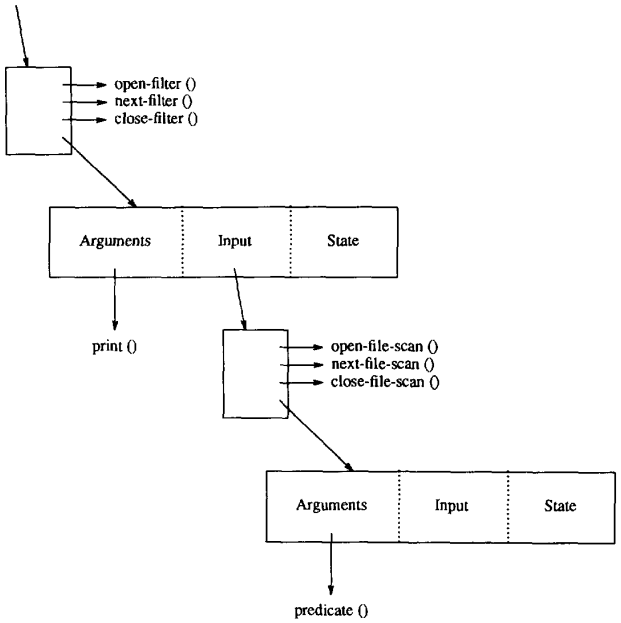


Fig. 2. Two operators in a query evaluation plan.

open-filter, can use the input pointer contained in the state record to invoke the input operator's functions. Thus, the filter functions can invoke the file scan functions as needed, and can pace the file scan according to the needs of the filter. In other words, Fig. 2 shows a complete query evaluation plan that prints selected records from a file.

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 and any kind of operators to evaluate a complex query, a second cornerstone to Volcano's extensibility. 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 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 ones being included in the E database implementation language in EXODUS and the query executor of the Starburst relational database system.

A number of query and environment parameters may influence policy decisions during *opening* 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 to pass information for policy decisions. Such policy decisions are implemented using support functions again. For example, the module implementing hash join allows dynamic determination of the size of a hash table—another example of the separation of mechanism and policy. This *bindings* parameter is particularly useful in dynamic query evaluation plans, which will be discussed later in Section V.

The tree-structured query evaluation plan is used to execute queries by demand-driven dataflow. The return value of a *next* operation is, besides a status indicator, a structure called *Next-Record*, which consists of an RID 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. Since this could result in a large number of buffer calls (one per record in each operator in the query), the interface to the buffer manager was recently redesigned such that it will require a total of two buffer calls per cluster on the procedure side (e.g., a file scan) independently of how many records a cluster contains, and only one buffer call per cluster on the consumer side.

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-Record* pairs, triples, etc., as intermediate results. Although this alternative results in less memory-to-memory copying, it is not implemented explicitly because Volcano already provides the necessary mechanisms, namely the *filter* iterator (see next subsection) that can replace each record in a stream by an RID-pointer pair or vice versa.

In summary, demand-driven dataflow is implemented by encoding operators as iterators, i.e., with *open*, *next*, and *close* procedures, since this scheme promises generality, extensibility, efficiency, and low overhead. The next few sections describe some of Volcano's existing iterators in more detail. In very few modules, the described operators provide much of the functionality of other query evaluation systems through generality and separation of mechanisms and policies. Furthermore, the separation of set processing control (iteration) from item interpretation and manipulation provides this functionality independently from any data model.

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 initiate a scan using the scan procedures of the file system level. The file name or closed file descriptor are given in the state record as are an optional predicate and bounds for B^+-tree scans. Thus, the two scan interfaces are functionally equivalent. Their difference is that the file system scan interface is used by various internal modules, e.g., by the device module for the device table of contents, while the iterator interface is used to provide leaf operators for query evaluation plans.

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 records retrieved using the RID's or it composes new records from the input records and the retrieved records, thus "joining" the B^+-tree entries and their corresponding data records.

B^+-tree scan and functional join are separated for a number of reasons. First, it is not clear that storing data in B^+-tree leaves never is a good idea. At times, it may be desirable to experiment with having other types of information associated with look-up keys. Second, this separation allows experimentation with manipulation of RID-lists for complex queries. Third, while functional join is currently implemented rather naively, this operation can be made more intelligent to assemble complex objects recursively. In summary, separating index search and record retrieval is another example for providing mechanisms in Volcano to allow for experiments with policies, a design principle employed to ensure that the Volcano software would be flexible and extensible.

The filter operator used in the example above performs three functions, depending on presence or absence of corresponding support functions in the state record. The *predicate* function applies a selection predicate, e.g., to implement bit vector filtering. 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, other changes in codes and representations, and reducing a stream of records to RID-pointer pairs. Finally, the *apply* function is invoked once on each record for the benefit of its side effects. Typical examples are updates and printing. Notice that updates are done within streams and query evaluation plans. Thus, Volcano plans are not only retrieval

but also *update* plans. The filter operator is also called the "side-effect operator." Another example is creating a filter for bit vector filtering. In other words, the filter operator is a very versatile single-input single-output operator that can be used for a variety of purposes. Bit vector filtering is an example for a special version of separation of policy and mechanism, namely the rule not to provide an operation that can be composed easily and efficiently using existing operations.

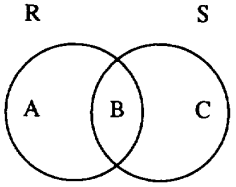
2) One-to-One Match: Together with the filter operator, the *one-to-one match* operator will probably be among the most frequently used query processing operators in Volcano as it implements a variety of set-matching functions. In a single operator, it realizes join, semi-join, outer joint, anti-joint, intersection, union, difference, anti-difference, aggregation, and duplicate elimination. The one-to-one match operator is a physical operator like sort, i.e., part of the executable algebra, not a logical operator like the operators of relational algebra. It is the operator that implements all operations in which an item is included in the output depending on the result of a comparison between a pair of items.

Fig. 3 shows the basic principle underlying the one-to-one match operator for binary operations, namely separation of the matching and non-matching components of two sets, called R and S in the Fig. 3, and producing appropriate subsets, possibly after some transformation and combination as in the case of a join. Since all these operations require basically the same steps, it was logical to implement them in one general and efficient module. The main difference between the unary and binary operations, e.g., aggregate functions and equi-join, is that the former require comparing items of the same input while the latter require comparing items of two different inputs.

Since the implementation of Volcano's one-to-one match is data-model-independent and all operations on data items are imported via support functions, the module is not restricted to the relational model but can perform set matching functions for arbitrary data types. Furthermore, the hash-based version provides recursive hash table overflow avoidance [12] and resolution similar to hybrid hash join [31] and can therefore handle very large input sizes. The sort-based version of one-to-one match is based on an external sort operator and can also operate on arbitrarily large inputs.

While there seems to be an abundance of join algorithms, our design goals of extensibility and limited system size led to the choice of only two algorithms (at the current time) to be implemented in Volcano, namely merge join and hybrid hash join. This choice will also allow experimental research into the duality and trade-offs between sort- and hash-based query processing algorithms.

The *classic hash join* algorithm (which is the in-memory component of hybrid hash join) 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 called the *flush phase*, which is needed for aggregate functions and some other operations.



Output	Full Match	Attribute match
A	Difference	Anti-semi-join
B	Intersection	Join, semi-join
C	Difference	Anti-semi-join
A, B		Left outer join
A, C	Anti-difference	Anti-join
B, C		Right outer join
A, B, C	Union	Symmetric outer join

Fig. 3. Binary one-to-one matching.

Since the one-to-one match operator is an interator like all Volcano operators, 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.

The build phase can be used to eliminate duplicates or to perform an aggregate function in the build input. 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. For aggregation, instead of inserting a new tuple into the hash table as in the classic 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.

While hash tables in main memory are usually quite fast, a severe problem occurs if the build input does not fit in main memory. 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 [12]. Second, overflow files can be created using *overflow resolution* after the problem occurs.

For Volcano's one-to-one match, we have adopted hybrid hash join. Compared to the hybrid hash algorithm used in GAMMA, our overflow resolution scheme has several improvements. Items can be inserted into the hash

table without copying, i.e., the hash table points directly to records in the buffer as produced by one-to-one match's build input. If input items are not densely packed, however, the available buffer memory can fill up very quickly. Therefore, the one-to-one match operator has an argument called the *packing threshold*. When the number of items in the hash table reaches this threshold, items are packed densely into overflow files. However, the clusters (pages) of these overflow files are not yet unfixed in the buffer, i.e., no *I/O* is performed as yet. Only when the number of items in the hash table reaches a second threshold called *spilling threshold* is the first of the partition files unfixed. The clusters of this file are written to disk and the count of items in the hash table accordingly reduced. When this number reaches the spilling threshold again, the next partition is unfixed, etc. If necessary, partitioning is performed recursively, with automatically adjusted packing and spilling thresholds. The unused portions of the hash table, i.e., the portions corresponding to spilled buckets, are used for bit vector filtering to save *I/O* to overflow files.

The fan-out of the first partitioning step is determined by the total available memory minus the memory required to reach the packing threshold. By choosing the packing and spilling thresholds, a query optimizer can avoid record copying entirely for small build inputs, specify overflow avoidance (and the maximum fan-out) for very large build inputs, or determine packing and spilling thresholds based on the expected build input size. In fact, because the input sizes cannot be estimated precisely if the inputs are produced by moderately complex expressions, the optimizer can adjust packing and spilling thresholds based on the esimated probability distributions of input sizes. For example, if overflow is very unlikely, it might be best to set the packing threshold quite high such that, with high probability, the operation can proceed without copying. On the other hand, if overflow is more likely, the packing threshold should be set lower to obtain a larger partitioning fan-out.

The initial packing and spilling thresholds can be set to zero; in that case, Volcano's one-to-one match performs overflow avoidance very similar to the join algorithm used in the Grace database machine. Beyond this parameterization of overflow avoidance and resolution, Volcano's one-to-one match algorithm also permits optimizations of cluster size and recursion depth similar to the ones used for sorting [4], [21] and for nonuniform hash value distributions, and it can operate on inputs with variable-length records.

The extension of the module 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 best implemented as semi-join. The union is the (double-sided) outer join of union-compatible relations. Difference and anti-difference of two sets can be computed using special settings of the algorithm's bells and whistles. Finally, a Cartesian product can be implemented by matching successfully all possible pairs of items from the two inputs.

A 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. Merge–join has been generalized similarly to hash–join to support semi-join, outer join, anti-join, and set operations. The sort operator has been implemented in such a way that it uses and provides the iterator interface. 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 also supports aggregation and duplicate elimination. It can perform these operations early, i.e., while writing temporary files [2]. The sort algorithm is described and evaluated in detail in [21].

In summary, Volcano's one-to-one match operators are very powerful parts of Volcano's query execution algebra. By separating the control required to operate on sets of items and the interpretation and manipulation of individual items it can perform a variety of set matching tasks frequently used in database query processing, and can perform these tasks for arbitrary data types and data models. The separation of mechanisms and policies for overflow management supports overflow avoidance as well as hybrid hash overflow resolution, both recursively if required. Implementing sort- and hash-based algorithms in a comparable fashion will allow meaningful experimental research into the duality and trade-offs between sort- and hash-based query processing algorithms. The iterator interface guarantees that the one-to-one match operator can easily be combined with other operations, including new iterators yet to be designed.

3) One-to-Many Match: While the one-to-one match operator includes an item in its output depending on a comparison of two items with one another, the one-to-many match operator compares each item with a number of other items to determine whether a new item is to be produced. A typical example is relational division, the relational algebra operator corresponding to universal quantification in relational calculus. There are two versions of relational division in Volcano. The first version, called *native division*, is based on sorting. The second version, called *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 [16] along with analytical and experimental performance comparisons and detailed discussions of two partitioning strategies for hash table overflow and multiprocessor implementations. We are currently studying how to generalize these algorithms in a way comparable with the generalizations of aggregation and join, e.g., for a *majority* function [8].

IV. Extensibility

A number of database research efforts strive for extensibility, e.g., EXODUS, GENESIS, Postgres, Starburst, DASDBS [30], Cactis [24], and others. Volcano is a very open query evaluation architecture that provides easy extensibility. Let us consider a number of frequently proposed database 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. To a certain extent, Volcano is incomplete (it is not a database system), but by separating set processing and instance interpretation and providing a well-defined interface between them, Volcano is inherently extensible on the level of instance types and semantics.

As a rule, data items that are transferred between operators using some *next* iterator procedure are records. For an extensible or object-oriented database system, this would be an unacceptable problem and limitation. The solution to be used in Volcano is to pass only the root component (record) between operators after loading and fixing necessary component records in the buffer and suitably swizzling inter-record pointers. Very simple objects can be assembled in Volcano with the functional join operator. Generalizations of this operator are necessary for object-oriented or non-first-normal-form database systems, but can be included in Volcano without difficulty. In fact, a prototype for such an *assembly* operator has been built [26] for use in the Revelation object-oriented database systems project [15].

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. In other words, the query processing routines are not affected by the semantics of the support functions as long as interface and return values are correct. The reason 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 interpreting and manipulating 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 [22], appropriate iterators have to be defined. Notice that it makes sense to perform not only retrieval but also maintenance of storage structures in the form of iterators. For example, if a set of items defined via a predicate (selection) needs to be updated, the iterator or query tree implementing the selection can "feed" its data into a maintenance iterator. The items fed into the maintenance operator should include a reference to the part of the storage structure to be updated, e.g., a RID or a key, and appropriate new values if they have been computed in the selection, e.g., new salaries from old salaries. Updating multiple structures (multiple indices) can be organized and executed very efficiently using nested iterators, i.e., a query evaluation plan. Furthermore, if ordering makes maintenance more efficient as for B-trees, an ordering or sort iterator can easily be included in the plan. In other words, it makes sense to think of plans not only as query plans used in retrieval but also as "update plans" or combinations of retrieval and update plans. 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, the algorithm needs to be coded in the iterator paradigm. In other words, the algorithm implementation must provide *open*, *next*, and *close* procedures, and must use these procedures for its input stream or streams. After an algorithm has been brought into this form, its integration with Volcano is trivial. In fact, as the Volcano query processing software became more complex and complete, this was done a number of times. For example, the one-to-many match or division operators [16] were added without regard to the other operators, and when the early in-memory-only version of hash-based one-to-one match was replaced by the version with overflow management described above, none of the other operators or meta-operators had to be changed. Finally, a complex object assembly operator was added recently to Volcano [26].

Extensibility can also be considered in a different context. In the long run, it clearly is desirable to provide an interactive front-end to make using Volcano easier. We are currently working on a two front-end, a nonoptimized command interpreter based on Volcano's executable algebra and an optimized one based on a logical algebra or calculus language including query optimization implemented with a new optimizer generator. The translation between plans as produced by an optimizer and Volcano will be accomplished using a module that "walks" query evaluation plans produced by the optimizer and Volcano plans, i.e., state records, support functions, etc. We will also use the optimizing front-end as a vehicle for experimentation with *dynamic query evaluation plans* that are outlined in the next section.

In summary, since Volcano is very modular in its design, extensibility is provided naturally. It could be argued that this is the case only because Volcano does not address the hard problems in extensibility. However, this argument does not hold. Rather, Volcano is only one component of a database system, namely the query execution engine. Therefore, it addresses only a subset of the extensibility problems and ignores a different subset. As a query processing engine, it provides extensibility of its set of query processing algorithms, and it does so in a way that matches well with the extensibility as provided by

query optimizer generators. It does not provide other database services and abstractions like a type system and type checking for the support functions since it is not an extensible database system. The Volcano routines assume that query evaluation plans and their support functions are correct. Their correctness has to be ensured before Volcano is invoked, which is entirely consistent with the general database systems concept to ensure correctness at the highest possible level, i.e., as soon as possible after a user query is parsed. The significance of Volcano as an extensible query evaluation system is that it provides a simple but very useful and powerful set of mechanisms for efficient query processing and that it can and has been used as a flexible research tool. Its power comes not only from the fact that it has been implemented following a few consistent design principles but also from its two *meta-operators* described in the next two sections.

V. 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 [17].*[1]

Volcano includes a *choose-plan* operator that allows realization of both multiplan access modules and dynamic plans. In some sense, it is not an operator as it does not perform any data manipulations. Since it provides control for query execution it is a *meta-operator*. 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*.

Fig. 4 shows a very simple dynamic plan. Imagine a selection predicate controlled by a program variable. The

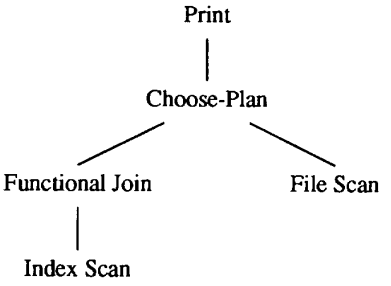


Fig. 4. A dynamic query evaluation plan.

index scan and functional join can be much faster than the file scan, but not when the index is nonclustering and a large number of items must be retrieved. Using the plan of Fig. 4, however, the optimizer can prepare effectively for both cases, and the application program using this dynamic plan will perform well for any predicate value.

The *choose-plan* operator allows considerable flexibility. If only one *choose-plan* operator is used as the top of a query evaluation plan, it implements a multiplan access module. If multiple *choose-plan* operators are included in a plan, they implement a dynamic query evaluation plan. Thus, all forms of dynamic plans identified in [17] can be realized with one simple and effective mechanism. Note that the *choose-plan* operator does not make the policy decision concerning which of several plans to execute; it only provides the mechanism. The policy is imported using a support function. Thus, the decision can be made depending on bindings for query variables (e.g., program variables used as constants in a query predicate), on the resource and contention situation (e.g., the availability of processors and memory), other considerations such as user priority, or all of the above.

The *choose-plan* operator provides significant new freedom in query optimization and evaluation with an extremely small amount of code. 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. The operator is another example for Volcano's design principle to provide mechanisms to implement a multitude of policies. We used the same philosophy when designing and implementing a scheme for parallel query evaluation.

VI. Multiprocessor Query Evaluation

A large number of research and development projects have shown over the last decade that query processing in relational database systems can benefit significantly from parallel algorithms. The main reasons parallelism is relatively easy to exploit in relational query processing systems are 1) query processing is performed using a tree of operators that can be executed in separate processes and processors connected with pipelines (inter-operator parallelism) and 2) each operator consumes and produces sets that can be partitioned or fragmented into disjoint subsets to be processed in parallel (intra-operator parallelism).

[1]This section is a brief summary of [17].

Fortunately, the reasons parallelism is easy to exploit in relational systems does not require the relational data model *per se*, only that queries be processed as sets of data items in a tree of operators. These are exactly the assumptions made in the design of Volcano, and it was therefore logical to parallelize extensible query processing in Volcano.

When Volcano was ported to a multiprocessor machine, it was desirable to use all single-process query processing code existing at that point *without any change*. The result is very clean, self-scheduling parallel processing. We call this novel approach the *operator model* of parallelizing a query evaluation engine [20].[2] In this model, all parallelism issues are localized in one operator that uses and provides the standard iterator interface to the operators above and below in a query tree.

The module responsible for parallel execution and synchronization is called the *exchange* iterator in Volcano. 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. Fig. 5 shows a complex query execution plan that includes data processing operators, i.e., file scans and joins, and exchange operators. The next two figures will show the processes created when this plan is executed.

This section describes how the *exchange* iterator implements vertical and horizontal parallelism followed by discussions of alternative modes of operation of Volcano's *exchange* operator and modifications to the file system required for multiprocess query evaluation. The description goes into a fair amount of detail since the *exchange* operator adds significantly to the power of Volcano. In fact, it represents a new concept in parallel query execution that is likely to prove useful in parallelizing both existing commercial database products and extensible single-process systems. It is described here for shared-memory systems only; considerations for the distributed-memory version are outlined as future work in the last section of this paper.

A. 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 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 rather than iterator (procedure) calls. 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 acknowledgment, and returns.

[2]Parts of this section have appeared in [20].

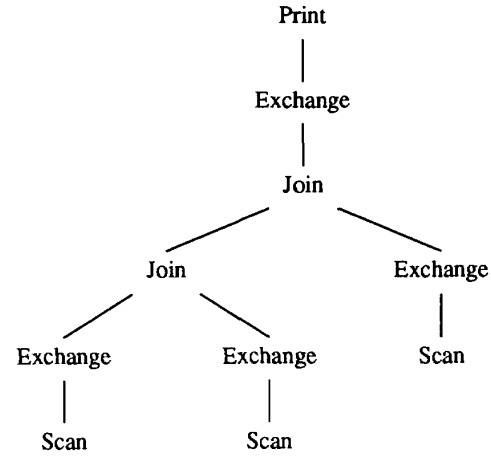


Fig. 5. Operator model of parallelization.

Fig. 6 shows the processes created for vertical parallelism or pipelining by the exchange operators in the query plan of the previous figure. The exchange operators have created the processes, and are executing on both sides of the process boundaries, hiding the existence of process boundaries from the "work" operators. The fact that the join operators are executing within the same process, i.e., the placement of the exchange operators in the query tree, was arbitrary. The exchange operator provides only the mechanisms for parallel query evaluation, and many other choices (policies) would have been possible. In fact, the mechanisms provided in the operator model tend to be more flexible and amenable to more different policies than in the alternative bracket model [20].

In the producer process, the exchange operator 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*, which are arrays of *Next-Record* structures. The packet size is an argument in the exchange iterator's state record, and can be set between 1 and 32 000 records. When a packet is filled, it is inserted into a linked list originating in the *port* and a semaphore is used to inform the consumer about the new packet. 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 in Volcano because files on virtual devices must not be closed before all their records are unpinned in the buffer. In other words, it is a peculiarity due to other design decisions in Volcano rather than inherent in the exchange iterator on the operator model of parallelization.

The alert reader has noticed that the exchange module uses a different dataflow paradigm than all other opera-

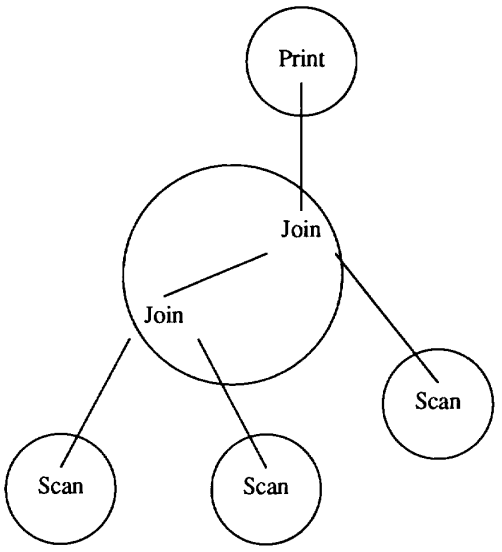


Fig. 6. Vertical parallelism.

tors. 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 reasons for this change in paradigms. First, we intend to use the exchange operator also for *horizontal parallelism*, to be described below, which is easier to implement with data-driven dataflow. 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, it would create unnecessary control overhead and delays. Since very-high degrees of parallelism and very-high-performance query evaluation require a closely tied network, e.g., a hypercube, of shared-memory machines, we decided to use a paradigm for data exchange that has been proven to perform well in a "shared-nothing" database machine [11].

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 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 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.

B. 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.

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.

Fig. 7 shows the processes created for horizontal parallelism or partitioning by the exchange operators in the query plan shown earlier. The join operators are executed by three processes while the file scan operators are executed by one or two processes each, typically scanning file partitions on different devices. To obtain this grouping of processes, the only difference to the query plan used for the previous figure is that the "degree of parallelism" arguments in the exchange state records have to be set to 2 or 3, respectively, and that partitioning support functions must be provided for the exchange operators that transfer file scan output to the joint processes. All file scan processes can transfer data to all join processes; however, data transfer between the join operators occurs only within each of the join processes. Unfortunately, this restriction renders this parallelization infeasible if the two joins are on different attributes and partitioning-based parallel join methods are used. For this case, a variant of exchange is supported in Volcano exchange operator called *interchange*, which is described in the next section.

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 *opened*, only one process is running, which is naturally the master. When a master forks a child process in a producer–consumer relationship, the child process becomes the master within its group. The first action of the master producer is to deter-

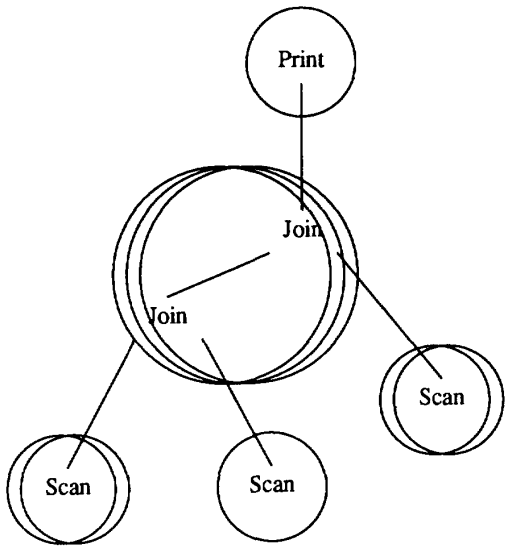


Fig. 7. Horizontal parallelism.

mine 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.

After all producer processes are forked, they run without further synchronization among themselves, with two exceptions. First, when accessing a shared data structure, e.g., the port to the consumers or a buffer table, short-term locks must be acquired for the duration of one linked-list insertion. 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 that 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 an orderly fashion, and the entire query evaluation is self-scheduling.

C. Variants of the Exchange Operator

There are a number of situations for which the *exchange* operator described so far required some modifications or extensions. In this section, we outline additional capabilities implemented in Volcano's exchange operator. All of these variants have been implemented in the *exchange* operator and are controlled by arguments in the state record.

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 [16] requires that the divisor be replicated and used with each partition of the dividend. Another example are fragment-and-replicate parallel join algorithms in which one of the two input relations is not moved at all while the other relation is sent to all processors. To support these algorithms, the exchange operator can be directed 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.

During implementation and benchmarking of parallel sorting [18], [21], 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. 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*. All these modifications were implemented in such a way that they support multilevel merge trees, e.g., a parallel binary merge tree as used in [3]. The merging paths are selected automatically such that the load is distributed as evenly as possible in each level.

Second, we implemented a sort algorithm that sorts data randomly partitioned (or "striped" [29]) over multiple disks into a range-partitioned file with sorted partitions, i.e., a sorted file distributed over multiple disks. When using the same number of processors and disks, two processes per CPU were required, one to perform the file scan and partition the records and another one to sort them. Creating and running more processes than processors can inflict a significant cost since these processes compete for the CPU's and therefore require operating system scheduling.

In order to make better use of the available processing power, we decided to redue the number of processes by half, effectively moving to one process per CPU. 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 *opened*, 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 was termed *interchange*, and was referred to earlier in the discussion of Fig. 7.

This mode of operation 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.

D. File System Modifications

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 each disk's volume table of contents. Furthermore, typically nonrepetitive actions like mounting a device must be invoked by the query root process before or after a query is evaluated by multiple processes.

The most intricate changes were required for the *buffer* module. In fact, making sure the buffer manager would not be a bottleneck in a shared-memory machine was an interesting subproject independent of database query processing [18]. Concurrency control in the buffer manager was designed to provide a testbed for future research with effective and efficient mechanisms, and not to destroy the separation of policies and mechanisms.

Using one exclusive lock is the simplest way to protect a buffer pool and its internal data structures. However, 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 each descriptor (page or cluster resident in the buffer). The buffer pool lock must be held while searching or updating the hash tables and bucket chains. It is never held while doing *I/O*; thus, it is never held for a long period of time. A descriptor or cluster lock must be held while doing *I/O* or while updating a descriptor in the buffer, e.g., to decrease its fix count.

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 that holds the lock might be 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*; Volcano's restart scheme does not satisfy the second condition. On the other hand, starvation is theoretically possible but has become extremely unlikely after buffer modifications that basically eliminated buffer contention.

In summary, the exchange module encapsulates parallel query processing in Volcano. It provides a large set of mechanisms useful in parallel query evaluation. Only very few changes had to be made in the buffer manager and the other file system modules 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 processng entirely self-scheduling, supports a variety of policies, e.g., partitioning schemes or packet sizes, 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 entirely separates data selection, manipulation, derivation, etc. from all parallelism issues, and may therefore prove useful in parallelizing other systems, both relational commercial and extensible research systems.

VII. Summary and Conclusions

We have described Volcano, a new query evaluation system that combines compactness, efficiency, extensibility, and parallelism in a dataflow query evaluation system. Compactness is achieved by focusing on few general algorithms. For example, the one-to-one match operator implements join, semi-join, our join, anti-join, duplication elimination, aggregation, intersection, union, difference, and anti-difference. 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., type and structure of their elements, are not part of the stream definition and its implementation, and can be determined at will, making Volcano a *data-model-independent set processor*. The separation of set processing control in *iterators* and object interpretation and manipulation through support functions contributes significantly to Volcano's extensibility.

The Volcano design and implementation was guided by a few simple but generally useful principles. First, Volcano implements *mechanisms* to support *policies* that can be determined by a human experimenter or a query optimizer. Second, operators are implemented as iterators to allow efficient transfer of data and control within a single process. Third, a uniform operator interface allows for integration of new query processing operators and algorithms. Fourth, the interpretation and manipulation of stream elements is consistently left open to allow supporting any data model and processing items of any type, shape, and representation. Finally, the encapsulated implementation of parallelism allows developing query processing algorithms in a single-process environment but executing them in parallel. These principles have led to a

very flexible, extensible, and powerful query processing engine.

Volcano introduces two novel *meta-operators*. Dynamic query evaluation plans are a new concept introduced in [17] that allow efficient evaluation of queries with free variables. The *choose-plan* meta-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.

Dataflow techniques are used within processes as well as between processes. Within a process, demand-driven dataflow is implemented by means of streams and iterators. Streams and iterators represent the most efficient execution model in terms of time and space for single-process query evaluation. Between processes, data-driven dataflow is used to pass data between producers and consumers efficiently. If necessary, Volcano's data-driven dataflow can be augmented with flow control or back pressure. Horizontal partitioning can be used both on stored and intermediate datasets to allow intra-operator parallelism. The design of the *exchange* meta-operator encapsulates the parallel execution mechanism for vertical, bushy, and intra-operator parallelism, and it performs the translations from demand-driven to data-driven dataflow and back [20].

Encapsulating all issues of parallelism control in one operator and thus orthogonalizing data manipulation and parallelism offers important extensibility and portability advantages. All data manipulation operators are shielded from parallelism issues, and have been designed, debugged, tuned, and preliminarily evaluated in a single-process environment. To parallelize a new operator, it only has to be combined with the exchange operator in a query evaluation plan. To port all Volcano operators to a new parallel machine, only the exchange operator requires appropriate modifications. At the current time, the exchange operator supports parallelism only on shared-memory machines. We are currently working on extending this operator to support query processing on distributed-memory machines while maintaining its encapsulation properties. However, we do not want to give up the advantages of shared memory, namely fast communication and synchronization. A recent investigation demonstrated that shared-memory architectures can deliver near-linear speed-up for limited degrees of parallelism; we observed a speed-up of 14.9 with 16 CPU's for parallel sorting in Volcano [18]. To combine the bets of both worlds, we are building our software such that it runs on a closely-tied group, e.g., a hypercube or mesh architecture, of shared-memory parallel machines. Once this version of Volcano's exchange operator and therefore all of Volcano runs on such machines, we can investigate query processing on hierarchical architectures and heuristics of how CPU and *I/O* power as well as memory can best be placed and exploited in such machines.

Most of today's parallel machines are built as one of the two extreme cases of this hierarchical design: a distributed-memory machine uses single-CPU nodes, while a shared-memory machine consists of a single node. Software designed for this hierarchical architecture will run on either conventional design as well as a genuinely hierarchical machine, and will allow exploring trade-offs in the range of alternatives in between. Thus, the operator model of parallelization also offers the advantage of architecture- and topology-independent parallel query evaluation [19].

A number of features make Volcano an interesting object for further 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, the duality and trade-offs between sort- and hash-based query processing algorithms and their implementations will be explored further. Fourth, Volcano allows measuring the performance of parallel algorithms and identifying bottlenecks on a shared-memory architecture, as demonstrated for instance in [18]. We intend to perform similar studies on distributed-memory and, as they become available, hierarchical architectures. Fifth, the advantages and disadvantages of a separate scheduler process in distributed-memory query processing (as used in GAMMA) will be evaluated. Finally, after data-driven dataflow has been shown to work well on a shared-nothing database machine [11], 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 extensions and improvements. First, Volcano currently does very extensive error detection, 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, even more so in a distributed-memory environment. Second, for a more complete performance evaluation, Volcano should be enhanced to a multiuser 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 operational query evaluation system that combines extensibility and parallelism. We believe that Volcano is 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 have given copies to selected outside organizations. We intend to use it in a number of further research projects, including research on the optimization and evaluation of dynamic query evaluation plans [17] and the REVELATION project on query optimization and execution in object-oriented database systems with encapsulated behavior [15].

ACKNOWLEDGMENT

The one-to-one match operators were implemented by Tom Keller starting with existing hash join, hash aggre-

gate, merge join, and sort code. Hash table overflow management was added by Mark Swain. Dynamic query evaluation plans and the *choose-plan* operator were designed and implemented by Karen Ward. We are also very much indebted to all members of the GAMMA and EXODUS projects. Leonard Shapiro contributed to the quality and clarity of the exposition with many insightful comments. David DeWitt, Jim Gray, David Maier, Bill McKenna, Marguerite Murphy, and Mike Stonebraker gave very helpful comments on earlier drafts of this paper. The anonymous referees gave some further helpful suggestions.

REFERENCES

[1] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise, "GENESIS: An extensible database management system," *IEEE Trans. Software Eng.*, vol. 14, p. 1711, Nov. 1988.

[2] D. Bitton and D. J. DeWitt, "Duplicate record elimination in large data files," *ACM Trans. Database Syst.*, vol. 8, p. 255, June 1983.

[3] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson, "Parallel algorithms for the execution of relational database operations," *ACM Trans. Database Syst.*, vol. 8, p. 324, Sept. 1983.

[4] K. Bratbergsengen, "Hashing methods and relational algebra operations," in *Proc. Conf. Very Large Data Bases*, Singapore, Aug. 1984, p. 323.

[5] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, "Object and file management in the EXODUS Extensible database system," in *Proc. Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, p. 91.

[6] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg, "A data model and query language for EXODUS," in *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, p. 413.

[7] 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*, D. M. S. Zdonik, Ed. San Mateo, CA: Morgan Kaufman, 1990.

[8] J. V. Carlis, "HAS: A relational algebra operator, or divide is not enough to conquer," in *Proc. IEEE Conf. Data Eng.*, Los Angeles, CA, Feb. 1986, p. 254.

[9] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolo, P. G. Selinger, M. Schkolnik, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, "A history and evaluation of system R," *Commun. Assoc. Comput. Mach.*, vol. 24, p. 632, Oct. 1981.

[10] H. T. Chou, D. J. DeWitt, R. H. Katz, and A. C. Klug, "Design and implementation of the Wisconsin storage system," *Software—Practice Experience*, vol. 15, no. 10, p. 943, Oct. 1985.

[11] D. J. DeWitt, S. Ghandeharadizeh, D. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The Gamma database machine project," *IEEE Trans. Knowledge Data Eng.*, vol. 2, p. 44, Mar. 1990.

[12] S. Fushimi, M. Kitsuregawa and H. Tanaka, "An overview of the system software of a parallel relational database machine GRACE," in *Proc. Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, p. 209.

[13] G. Graefe and D. J. DeWitt, "The EXODUS optimizer generator," in *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, p. 160.

[14] G. Graefe, "Rule-based query optimization in extensible database systems," Ph.D. dissertation Univ. Wisconsin-Madison, Aug. 1987.

[15] G. Graefe and D. Maier, "Query optimization in object-oriented database systems: A prospectus," in *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science, vol. 334, K. R. Dittrich, Ed. New York: Springer-Verlag, Sept. 1988, p. 358.

[16] G. Graefe, "Relational division: Four algorithms and their performance," in *Proc. IEEE Conf. Data Eng.*, Los Angeles, CA, Feb. 1989, p. 94.

[17] G. Graefe and K. Ward, "Dynamic query evaluation plans," in *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, p. 358.

[18] G. Graefe and S. S. Thakkar, "Tuning a parallel database algorithm on a shared-memory multiprocessor," *Software-Practice and Experience*, vol. 22, no. 7, July 1992, p. 485.

[19] G. Graefe and D. L. Davison, "Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Processing," to appear in *IEEE Trans. on Softw. Eng.*, vol. 19, no. 8, August 1993.

[20] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," in *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, p. 102.

[21] —, "Parallel external sorting in Volcano," CU Boulder Comput. Sci. Tech. Rep. 459, Feb. 1990.

[22] A. Guttman, "*R*-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, p. 47.

[23] L. Haas, W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, and M. J. Carey, and E. Shekita, "Starburst mid-flight: As the dust clears," *IEEE Trans. Knowledge Data Eng.*, vol. 2, p. 143, Mar. 1990.

[24] S. E. Hudson and R. King, "Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system," *ACM Trans. Database Syst.*, vol. 14, p. 291, Sept. 1989.

[25] T. Keller and G. Graefe, "The one-to-one match operator of the Volcano query processing system," Oregon Grad. Center, Comput. Sci. Tech. Rep., Beaverton, OR, June 1989.

[26] T. Keller, G. Graefe, and D. Maier, "Efficient assembly of complex objects," Proc. ACM SIGMOD Conf., Denver, CO, May 1991, p. 148.

[27] J. E. Richardson and M. J. Carey, "Programming constructs for database system implementation in EXODUS," in *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, p. 208.

[28] J. E. Richardson, "E: A persistent systems implementation language," Comput. Sci. Tech. Rep. 868, Univ. Wisconson—Madison, Aug. 1989.

[29] K. Salem and H. Garcia-Molina, "Disk Striping," in *Proc. IEEE Conf. Data Eng.*, Los Angeles, CA, Feb. 1986, p. 336.

[30] H. J. Schek, H. B. Paul, M. H. Scholl, and G. Weikum, "The DASDBS project: Objectives, experiences, and future prospects," *IEEE Trans. Knowledge Data Eng.*, vol. 2, p. 25, Mar. 199.

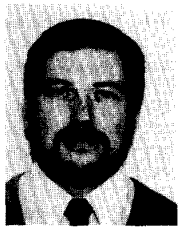
[31] L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Trans. Database Syst.*, vol. 11, p. 239, Sept. 1986.

[32] M. Stonebraker, "Retrospection on a database system," *ACM Trans. Database Syst.*, vol. 5, p. 225, June 1980.

[33] M. Stonebraker, P. Aoki, and M. Seltzer, "Parallelism in XPRS," UCB/Electronics Research Lab. Memo. M89/16, Berkeley, CA, Feb. 1989.

[34] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," in *Proc. Conf. Very Large Databases*, Long Beach, CA, Aug. 1988, p. 318.

[35] M. Stonebraker, L. A. Rowe, and M. Hirohama, "The Implementation of Postgres," *IEEE Trans. Knowledge Data Eng.*, vol. 2, p. 125, Mar. 1990.



Goetz Graefe was an undergraduate student in business administration and computer science in Germany before getting his M.S. and Ph.D. degree in computer science in 1984 and 1987, respectively, from the University of Wisconsin—Madison. His dissertation work was on the EXODUS Optimizer Generator under David DeWitt and Michael Carey.

In 1987, he joined the faculty of the Oregon Graduate Institute, where he initiated both the Volcano project and with David Maier, the REVELATION OODBMS project. From 1989 to 1992, he was with the University of Colorado at Boulder. Since 1992, he has been an Associate Professor with Portland State University. He is currently working on extensions to Volcano, including a new optimizer generator, on request processing in OODBMS's and scientific databases, and on physical database design.