Extensible Query Optimization and Parallel Execution in Volcano

Goetz Graefe

CU-CS-548-91 October 1991

University of Colorado at Boulder

DEPARTMENT OF COMPUTER SCIENCE

Extensible Query Optimization and Parallel Execution in Volcano

Goetz Graefe

CU-CS-548-91 October 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-7514
(303) 492-2844 Fax
graefe@cs.colorado.edu

Extensible Query Optimization and Parallel Execution in Volcano

Goetz Graefe, University of Colorado at Boulder

Abstract

The Volcano project focuses on data model-independent and architecture-independent optimized parallel query processing over large data sets using multiple operators or data processing steps. This overview describes the new rule-based Volcano optimizer generator and Volcano's architecture-independent parallel processing capabilities.

1. Introduction

Among the numerous current research projects in the areas of extensible and object-oriented database management, none attempts to combine extensibility and data model-independence with efficient, optimized, parallel request execution. Considering that efficient execution of requests specified in high-level languages has contributed significantly to the success of relational systems, we feel it is very important to combine extensibility, query optimization, and parallelism in powerful next-generation database management systems. The currently dominant direction in database research are object-oriented systems, and many researchers have decided to work on complex objects, their representation, storage, retrieval, maintenance, or clustering. While managing complex objects and their behavior, i.e., moving data structures and logic from the application program into the database management system, are important and interesting research topics, we feel that manipulation of large data volumes and derivation of new information from large data sets should not be ignored because it could cost object-oriented systems their well-deserved success in real applications.

The Volcano project at the University of Colorado at Boulder explores query processing techniques, both query optimization and query execution, that can support modern data models and, at the same time, exploit modern and forthcoming parallel hardware architectures. We do not propose to reintroduce relational query processing into next-generation database systems; instead, we work on a new kind of query processing engine that is independent of any data model. The basic assumption is that high-level languages are and will continue to be based on sets, predicates, and operators. Therefore, the only assumption we make in our research is that operators consuming and producing sets or sequences of items are the fundamental building blocks of next-generation query and request processing systems. In other words, we assume that some algebra of sets is the basis of query processing, and our research tries to support any algebra of sets. Fortunately, algebras and algebraic equivalence rules are a very suitable basis for database query optimization. Moreover, sets (permitting definition and exploitation of subsets) and operators with data passed (or pipelined) between them are also the foundations for parallel algorithms for database query processing.

To investigate query and request processing in next-generation database management systems, we have built an extensible and parallel query processing prototype called Volcano [26]. The important differences to all other extensible database projects are that Volcano does not rely on the relational model (though it would be perfectly reasonable to implement a relational query processing system using Volcano) and that it is more than yet another storage server for efficient storage and retrieval of records and objects. Instead, we strictly maintain model-independence and focus on manipulation of large sets of items, using as the only assumption that query and request processing is based on sets and sequences of objects. Volcano consists of an optimizer generator, an extensible set of operators, and a file system.

The optimizer generator is based on experiences building and using the EXODUS optimizer generator and uses very similar input syntax but employs a more efficient search algorithm and provides significantly better support for physical properties such as sort order, compression status, and location/partitioning.

The execution engine already includes sort- and hash-based matching functions such as join, semi-join (existential quantification), division (universal quantification), intersection, union, etc., but the collection of operators is also very extensible, both at the algorithm level (new operators and algorithms) and at the instance level (new data types and functions on them). Operators can be designed and implemented in a single-process system and later parallelized by a novel "exchange" operator. This operator encapsulates all information about parallelism, process and flow control, and even the underlying architecture.

Database query processing can be divided into query optimization and query execution. The former is based on techniques of dynamic programming, numerical optimization, artificial intelligence (planning and search), and statistics (selectivity and cost estimation). The latter is most closely related to data (storage) structures, algorithm design, operating systems, and computer architecture. Our research in the Volcano project and this paper follow this division.

2. Query Optimization

Query optimization in Volcano is based on a new optimizer generator. The basic idea of an optimizer generator, shown in Figure 1, is taken in principal from the EXODUS project [5]. When the DBMS software is built using an extensible toolkit, the optimizer source code is generated from a data model description, i.e., algebra and rule set are translated into pattern matching code and integrated with the search engine. After the DBMS software has been built, it can be used efficiently for all queries and requests without further code generation and compilation.

The other prominent extensible optimization projects, the EXODUS optimizer generator [11-13] and the Starburst join enumerator [27, 35], focus on algorithms for query optimization. In contrast, the Volcano system determines what facts and knowledge about expressions and plans are relevant for the request at hand and uses a tightly guided strategy to derive this knowledge efficiently. This approach makes it fairly easy to decide which derived knowledge to retain in a global variable (a fairly complex hash table) and which intermediate optimization results to discard immediately. This question was unresolved in the EXODUS optimizer generator and made it very CPU- and memory-intensive. Furthermore, because of the focus on interesting knowledge rather than algorithms, we plan on exploiting the approach for global (multi-query) optimization and

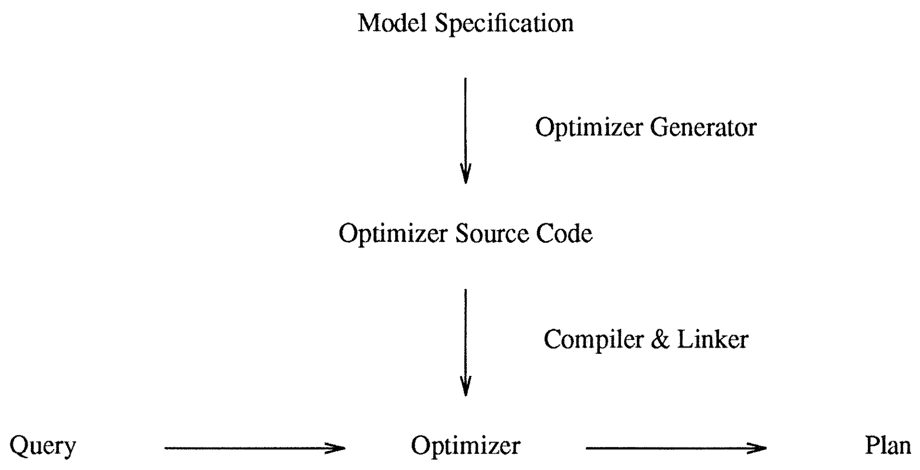

Figure 1. The Optimizer Generator Paradigm.

for dynamic query evaluation plans [16].

At this point, we are prototyping a small recursive algorithm that develops what knowledge about expressions and subexpressions is interesting and useful, and then derives such knowledge as efficiently as possible. The algorithm is controlled by a logical algebra and its transformation rules (e.g., join, select; join commutativity), a physical algebra and its operators' association with logical algebra operators (e.g., merge join, hash join; they implement join), selectivity functions, cost functions, etc. similar to the EXODUS optimizer generator. The input into the algorithm is an expression of the logical algebra, i.e., the query to be optimized, plus specific requirements for physical properties such as sort order or partitioning in parallel and distributed systems. The output is a plan expressed with operators of the physical algebra. The search strategy is much more directed, controllable, and efficient than the one used in EXODUS. The search strategy, i.e., the heuristic guidance of the algorithm, is modularized such that the algorithm can be used as a highly extensible experimental vehicle. Furthermore, a new class of algorithms used in query processing is introduced, called *enforcers* in Volcano, that enforce physical properties. Typical enforcer operators are sort (for sort order), exchange (for partitioning) [21, 22], decompression (for databases using automatic data compression [20, 24]), and choose-plan (for plan robustness under estimation errors during query optimization [16]).

Figure 2 shows an outline of the search algorithm used by the Volcano optimizer generator. The original invocation of the FindBestPlan procedure indicates the logical expression passed to the optimizer as the query to be optimized, physical properties as requested by the user (for example, sort order as in the SORT BY clause of SQL), and a cost limit of ∞. The set of possible physical properties is encoded in the *physical property vector*, which is an ADT defined by the optimizer implementor; the optimizer inspects and manipulates physical property vectors only via functions provided by the optimizer implementor. The cost limit can be given in any unit since all cost computations including comparisons and additions are performed in functions provided by the optimizer implementor.

The FindBestPlan procedure is broken into two parts. First, if a plan for the expression satisfying the physical property vector can be found in the hash table, either a plan and its cost or a failure indication are returned depending on whether or not the found plan satisfies the given cost limit. If the expression cannot be found in the hash table, or if the expression has been optimized before but not for the presently required physical properties, actual optimization is begun.

There are three possible "moves" the optimizer can explore at any point. First, the expression can be transformed using a transformation rule. Second, there might be some algorithms that can deliver the logical expression with the desired physical properties, e.g., hash join for unsorted output merge join for join output sorted on the join attribute. Third, an enforcer might be useful to permit additional algorithm choices, e.g., a sort operator to permit using hash join even if the final output is to be sorted.

After generating all possible moves, the optimizer assesses their promise of leading to the optimal plan. Clearly, good estimations of a move's promise are useful for finding the best plan fast. Considering that we are building an optimizer generator with as few assumptions as possible with respect to the logical and physical algebras, it is not possible to estimate a move's promise without assistance by the optimizer implementor. Thus, the optimizer implementor must provide estimation functions which will have a significant impact on an optimizer's performance.

The most promising moves are then pursued. For exhaustive search, these will be all moves. Otherwise, a subset of the moves determined by another function provided by the optimizer implementor are selected. Pursuing all moves or only a selected few ones is the second major heuristic that is placed into the hands of the optimizer implementor. Using this control function, an optimizer implementor can choose to transform a logical expression without any algorithm selection and cost analysis, which covers the optimizations that in Starburst are separated into the QGM (Query Graph Model) level [27, 28]. The difference between Starburst's two-level approach and Volcano's approach is that this separation is mandatory in

```
FindBestPlan (LogExpr, PhysProp, Limit)
if LogExpr & PhysProp is in the hash table
   if cost in hash table < Limit
      return plan and cost
   else
      return failure

else /* optimization required */
   create the set of possible "moves" from
      applicable transformations
      algorithms that give the required PhysProp
      enforcers for required PhysProp

   order the set of moves by promise

   for the most promising moves
      if the move is a transformation
         apply the transformation creating NewLogExpr
         call FindBestPlan (NewLogExpr, PhysProp, Limit)
      else if the move is an algorithm
         TotalCost := cost of the algorithm
         for all inputs I while TotalCost < Limit
            determine required phys. prop. for I
            find cost using FindBestPlan ()
            add cost to TotalCost
      else /* move uses enforcer */
         TotalCost := cost of the enforcer
         modify PhysProp for enforced property
         call FindBestPlan for LogExpr w/ modified PhysProp

   /* maintain hash table of explored facts */
   if LogExpr is not in hash table
      insert LogExpr into hash table
   insert PhysProp and best plan found into hash table
```

Figure 2. Outline of the Search Algorithm.

Starburst while it is a choice to be made by the optimizer implementor using Volcano.

The cost limit is used to improve the search algorithm using branch-and-bound pruning. Once a complete plan is known for a logical expression and a physical property vector, no plan or partial plan with higher cost can be part of the optimal query evaluation plan. Therefore, it is important (for optimization speed, not for correctness) that a relatively good plan be found fast, even if the optimizer uses exhaustive search. Furthermore, cost limits are passed down in the optimization of subexpressions, and tight upper bounds also speeds their optimization.

If a move to be pursued is a transformation, the new expression is formed and optimized using FindBestPlan. In order to detect the case that two (or more) rules are inverses of each other, the current expression and physical property vector is marked as "in progress." If a newly formed expression already exists in the hash table and is marked as "in progress," it is ignored because its optimal plan will be considered when it is finished.

If a move to be pursued is a normal query processing algorithm such as merge join, its cost is calculated by a cost function provided by the optimizer implementor. All its input (logical) expressions and their physical property vectors are found by another function provided by the optimizer implementor, and their costs are determined using FindBestPlan. Finally, all costs are added together using another optimizer implementor function. In this process, the Limit passed to FindBestPlan is the original Limit minus costs already computed. For example, if the total cost limit for a join expression is 10 cost units, the join algorithm takes 3 cost units and the left input takes 4 cost units, the limit for optimizing the right input is 3 cost units.

If the move to be pursued is an enforcer such as sort, its cost is estimated by a cost function provided by the optimizer implementor and the original logical expression with a suitably modified physical property vector is optimized using FindBestPlan. When optimizing the logical expression with the modified (i.e., relaxed) physical property vector, algorithms that already applied before relaxing the physical properties must not be explored again. For example, if a join result is required sorted on the join column, merge join (an algorithm) and sort (an enforcer) will apply. When optimizing the sort input, i.e., the join expression without sort condition, hash join should apply but merge join should not. To ensure this, FindBestPlan uses an additional parameter not shown in Figure 2 called the excluding physical property vector that is used only when inputs to enforcers are optimized. In the example, the excluding physical property vector would contain the sort condition, and since merge join is able to satisfy the excluding properties, it would not be considered a suitable algorithm for the sort input.

At the end of (or actually already during) the optimization procedure FindBestPlan, newly derived interesting facts are captured in the hash table. "Interesting" is defined with respect to possible future use. This can include both plans optimal for given physical properties as well as failures that prevent future attempts to optimize an logical expression and physical properties again with the same or even lower cost limits.

This description of the search algorithm used in Volcano does not cover all details for two reasons. First, further details would make the algorithm description even harder to follow, while the present level of detail presents the basic structure. Second, the algorithm implementation has only recently become operational, and we are still experimenting with it and tuning it to make more effective use of the hash table of explored expressions and optimized plans.

The algorithm used in the Volcano optimizer generator differs significantly from the one in the EXODUS optimizer generator in a number of important aspect. First, Volcano makes a distinction between logical expressions and physical expressions. In EXODUS, only one type of node existed in the hash table called MESH which contained both a logical operator like join and a physical algorithm like hash join. To retain equivalent plans using merge join and hash join, the logical expression (or at least one node) had to be kept twice resulting in a large number of nodes in MESH.

Second, the Volcano algorithm is driven top-down; subexpressions are optimized only if warranted. In the extreme case, it is possible to transform a logical expression without ever optimizing its subexpressions by selecting only one move, a transformation. In EXODUS, transformation were always followed immediately by algorithm selection and cost analysis. Moreover, transformations were explored whether or not they were part of the currently most promising logical expression and physical plan for the overall query. Worst of all for optimizer performance, however, was the decision to perform transformations with the highest expected cost improvement first. Since the cost improvement was calculated as product of a factor associated with the transformation rule and the current cost before transformation, nodes at the top of the expression (with high total cost) were preferred over lower expressions. When the lower expression were finally transformed, all consumer nodes above (of which there were many at this time) had to be reanalyzed creating even more MESH nodes.

Third, physical properties were handled only very haphazardly in EXODUS. If the algorithm with the lowest cost happened to deliver results with useful physical properties, this was recorded in MESH. Otherwise, the cost of enforcers (although this is a Volcano term) had to be included in the cost function of other algorithms such as merge join. In other words, the ability to specify required physical properties and let them, together with the logical expression, drive the optimization process as is done in Volcano was entirely absent in EXODUS.

Fourth, cost is defined in much more general terms in Volcano than in the EXODUS optimizer generator. In Volcano, cost is an abstract data type (ADT) that is handled only by invoking functions provided by the DBI. It can be a simple number, e.g., estimated elapsed seconds, a structure, e.g., a record consisting of CPU cost and I/O cost similar to the cost measures in System R [37], or a function. A function offers entirely new possibilities for query optimization. For example, it can be a function of the size of the available memory. This would allow optimizing plans for any run-time situation with respect to available memory and memory contention. Of course, it is not always possible to compare functions, i.e., to determine which of two functions is "less" than the other. In some situations, it might be possible, e.g., if one plan dominates another one independently of memory availability. If two functions cannot be compared because neither dominates the other in all situations, we propose using a *choose-plan* operator suggested in [16] which permits including several equivalent subplans in a single plan and delaying an optimization decision until query execution time.

Finally, we believe that the Volcano optimizer generator is more extensible than the EXODUS one, in particular with respect to the search strategy. The hash table that holds logical expressions and physical plans and operations on this hash table are quite general, and would support a variety of search strategies, not only the procedure outlines in Figure 2. We are still modifying (extending) the search strategy, and plan to modify it further in subsequent years and to use the Volcano optimizer generator for further research.

Algebraic transformation systems always include the possibility and danger of deriving the same expression in two different ways. Therefore, duplicate expression detection is a concern in algebraic query optimization. The Volcano optimization system uses a hash table of explored expressions to identify such duplicates. For global or multi-query optimization, duplicate expression detection is the major concern. We plan on extending this duplicate detection scheme from equivalent to subsumed expressions which will significantly increase the power of the optimizer generator for global query optimization.

Since query optimization can be a very time-consuming task, as shown many times for "simple" relational join optimization [32, 36], all means that can speed the process should be explored. Beyond traditional methods, we will explore storing logical expressions and their optimal plans for later reuse. While an optimal plan may not be optimal for an expression if the expression appears later in a different context (query), stored subplans hold promise for two reasons. First, they provide an upper bound; equivalent plans explored while optimizing the larger query can be safely abandoned if their cost surpasses that of the known plan. Second, since the optimizer is based on algebraic transformations, the stored optimal plan can be used as a basis from which to search for an optimal plan within the new context. Both reasons allow us to speculate that storing optimized plans may be a very useful concept, and we feel it can be explored most effectively in an extensible context such as the Volcano optimizer generator.

While dynamic query evaluation plans, global query optimization, and storage and reuse of optimized subplans are interesting and challenging research topics in the context of relational systems, we plan on pursuing them within an extensible system. There are two reasons for our approach. First, an extensible query optimization environment modularizes the various components of query optimizers [13], e.g., selectivity estimation, cost calculation, query transformation, search heuristics, etc. This greatly facilitates change, modification, and experimentation, and may indeed be a requirement for successful pursuit of our research goals. Second, none of these research problems is restricted to the relational domain, and our results will benefit relational, extensible, and object-oriented systems alike.

3. Query Execution

Volcano's query execution work is complementary to the optimizer research. It provides a wide variety of mechanisms for query execution parameterized and implemented in such a way that policies can be set by an optimizer or a human experimenter. Like the optimizer generator, the execution module was designed for extensibility. Another important design goal was high performance including the effective use of parallel execution on a variety of computer architectures.

Volcano's execution module is fairly complete as it already includes operators for file and B^+-tree scan and maintenance [26], sorting [19, 23], sort- and hash-based versions of a "one-to-many match" operator for matching operations such as join, semi-join, all outer joins, intersection, union, difference, aggregation, and duplicate elimination [26, 29], and universal quantification [17]. However, extension of the Volcano execution module is a continuous process.

Strict separation of set iteration and instance interpretation (operations on instances are imported into the operators as functions) enables all these operators to manipulate records and tuples as well as complex objects consisting of multiple, shared components. Furthermore, a simple and uniform interface between operators allows easy addition of new operations if required. In other words, the execution module is extensible both on the instance level (through imported functions on items) and on the operator level (through a uniform operator interface).

The most recent extensions were those required for distributed-memory and hierarchical machines (multiple shared-memory nodes) [22] and for fast assembly of sets of complex objects in memory [30]. Moreover, we are currently redesigning the hash-based one-to-many match operator which is based on hybrid hash join join [8] to exploit hash value skew in its input data for better performance rather than becoming slower under skew and uneven partition sizes.

Hybrid hash join partitions both inputs, called the build and probe inputs according to their use of the hash table, according to a hash function into multiple pairs of partition files. If the build input is small enough that it does not require the entire available memory as output buffers for the partitioning files, a hash table is kept in memory during the partitioning process and a part of the join is performed immediately. However, if the inputs are large and the fan-out is small (which it should be to permit fast, large-chunk I/O [4, 25]), recursive partitioning is required. In Volcano's new, skew-resistant one-to-many match operator, statistics about hash value distributions are gathered in one recursion level and used in the next level to assign hash buckets to the in-memory hash table and to partition files and to decide whether and when switching to nested loops join is required due to excessive duplication of an individual join key. In detailed simulation experiments, we found that not only can the detrimental effects of skew be completely avoided but skew can also, with proper handling, improve the performance over hybrid hash join with uniformly distributed input data.

Volcano's execution module includes two novel "meta-operators" that control query execution without directly contributing to data manipulation. First, the *choose-plan* operator implements dynamic query evaluation plans for embedded queries optimized at compile time with incomplete knowledge (e.g., predicate constants, current system load) as described in [16]. Second, the *exchange* operator allows parallelizing Volcano operators that were designed and implemented in single-process environments [21]. We are currently considering additional meta-operators for common subexpressions, data flow reversal for real-time databases, and extensions to the choose-plan operator to be used in very complex queries that do not permit complete optimization before run-time because of selectivity estimation errors.

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. Figure 3 shows a complex query execution plan that includes data processing operators, i.e., file scans and joins, and exchange operators.

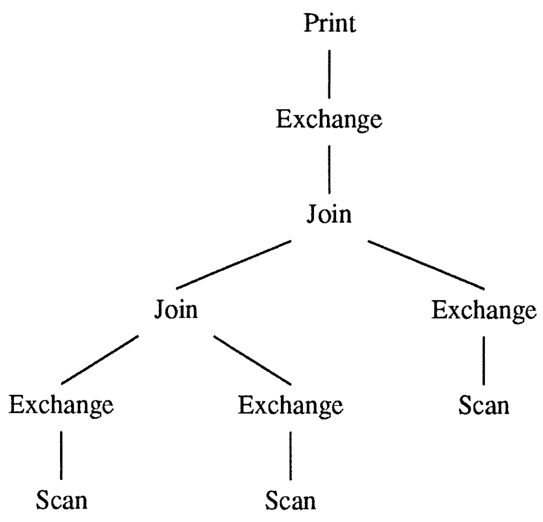

Figure 3. Operator Model of Parallelization.

Figure 4 shows the processes created when this plan is executed, including both vertical parallelism (pipelining) and horizontal parallelism (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 "degree of parallelism" arguments in the exchange state

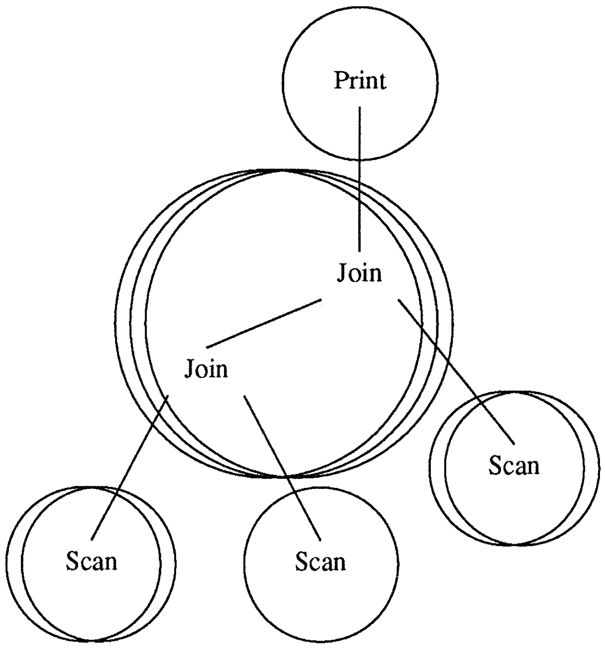

Figure 4. Horizontal Parallelism.

records have to be set to 2 or 3, respectively, and partitioning functions must be provided for the exchange operators that transfer file scan output to the join 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*. This and other variants of Volcano's exchange operator are described in [26].

The first version of Volcano's exchange operator supported only shared memory. When extending the Volcano and its exchange operator to support query processing on distributed-memory machines, we did not want to give up the advantages of shared memory, namely efficient communication, synchronization, and load balancing. A recent investigation demonstrated that shared-memory architectures can deliver near-linear speed-up for modest degrees of parallelism; we observed a speed-up of 14.9 with 16 CPUs and disks for parallel sorting in Volcano [19]. To combine the best of both worlds, Volcano now runs on an interconnected group, e.g., a hypercube or mesh architecture, of shared-memory parallel machines. Within each shared-memory machine, shared-memory mechanisms are used for synchronization and communication, while message-passing is used between machines. We can now 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.

Figure 5 shows a general hierarchical architecture. The important point is the combination of local busses within shared-memory parallel machines and a global interconnection network between machines. The diagram is only a very general outline of such an architecture; many details are deliberately left out and unspecified. The network could be

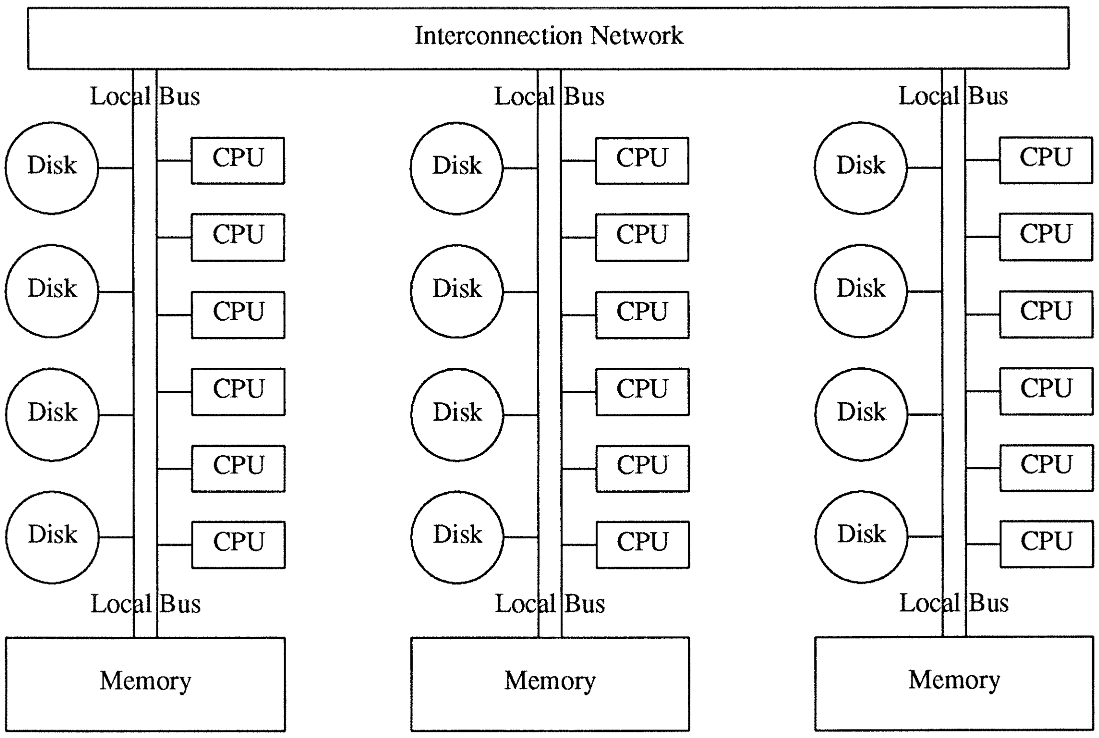

Figure 5. A Hierarchical-Memory Architecture.

implemented using a bus such as an ethernet, a ring, a hypercube, a mesh, or a set of point-to-point connections. The local busses may or may not be split into code and data or by address range to obtain less contention and higher bus bandwidth and hence higher scalability limits for the use of shared memory. Design and placement of caches, disk controllers, terminal connections, and local- and wide-area network connections are also left open. Tape drives or other backup devices would probably be connected to local busses.

Modularity is a very important consideration for such an architecture, i.e., the ability to add, remove, and replace individual units. For example, it should be possible to replace all CPU boards with upgraded models without having to replace memories or disks. Considering that new components will change communication demands, it is also important that the allocation of boards to local busses can be changed. For example, faster CPUs might require more local bus bandwidth, and it should be easy to reconfigure a machine with 4×16 CPUs into one with 8×8 CPUs.

Beyond the effect of faster communication and synchronization, this architecture can also have a significant effect on control overhead, load balancing, and response time problems due to these problems. Investigations in the Bubba project at MCC demonstrated that large degrees of parallelism may reduce performance unless startup overhead (e.g., synchronization and communication) and load imbalance can be kept relatively low [1, 6]. Consider placing 100 CPUs either in 100 nodes or in 10 nodes of 10 CPUs each. It is much faster to distribute query plans to all CPUs and much easier to achieve reasonably balanced loads in the second case than in the first case. Load balancing within a shared-memory machine can be accomplished by exchanging data (relatively cheaply) or by allocating resources such as CPUs, disk drives, and memory to reflect an uneven load and to attempt to achieve similar processing times among parallel processes.

This architecture may also be exploited for reliability and availability similarly to distributed-memory machines. Tandem's mirroring architecture could be recreated by pairing shared-memory machines and their storage devices. Long-distance logging, task migration, and recovery could also be designed and implemented. High availability also requires that components can be replaced while the rest of the machine is still operating.

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 tradeoffs 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. In other words, the parallelism operator is the only operator that needs to "understand" the underlying architecture, while all "work" operators can be implemented without any concern for parallelism, data distribution, flow control, etc. As current shared-memory and distributed-memory machines are subsumed by this architecture, the model of parallel query processing in Volcano subsumes those of Bubba [2, 3, 31], Gamma [9, 10], and the Teradata database machine [33, 34, 40] (distributed memory) as well as those of XPRS [38-] (shared memory). The versatility of the Volcano software to run on shared-memory, distributed-memory, and hierarchical architectures makes the Volcano project unique among experimental parallel database platforms.

Using the Volcano engine as an experimental testbed, we are now exploring tradeoffs in extensible parallel query processing, e.g., vertical parallelism (pipelining) versus horizontal parallelism (partitioning), left-deep versus right-deep versus bushy plans, shared-everything (single bus) versus shared-disks (e.g. VAX cluster) versus shared-nothing (distributed memory) versus hierarchical-memory architectures, data- versus demand-driven dataflow in database systems, self-scheduling operators versus central scheduling, extensibility versus performance, granularity of data exchange between processes, etc. None of these issues can be trivially answered; for example, the granularity of data exchange was found to have some influence on system performance even in a shared-memory system due to operating system semaphore and scheduling overhead [19]; however, reducing such overhead and chosing the granules too large can create unnecessary waiting, in particular at the beginning and the end of a data stream. In a distributed-memory environment, we have started to explore this issue but don't have any real data or insights yet.

The design and implementation of an experimental testbed for parallel request processing has been completed. We are now exploring some of these tradeoffs. Future research will analyze current trends in computer architecture, operating systems, and database systems, and determine where matches, overlaps, and inconsistencies enable, further, hamper, or prevent the development of next-generation high-performance database management systems.

4. Summary

Research into extensible, parallel query processing is one of the central challenges in building high-performance next-generation, object-oriented database management systems. Performance in query and request processing can be achieved through effective optimization, suitable physical database design, efficient processing algorithms, and parallel execution.

The work on the Volcano optimizer generator focuses on strategies and mechanisms to derive important facts and sub-plans for a query or request in a data model-independent yet efficient way. After an appropriate number (or all) possible query evaluation plans have been explored, the optimal plan is one of the facts derived. Future work will include deriving dynamic query execution plans, a knowledge base of logical expressions and suitable plans to speed other optimizations and facilitate global query optimization, and the foundations of an expert system for physical database design that interacts with the query optimizer.

The work on parallel query processing in the Volcano extensible database system has already provided mechanisms that allow parallelizing existing and new operators on various machines and architectures, including hierarchical architectures based on a closely tied group of shared-memory parallel nodes. Current work includes research into performance tradeoffs in extensible, parallel database systems, and uses this research to investigate current trends in computer architecture, operating systems, and database systems and to identify their concepts that are complementary to or inconsistent with the needs of an extensible DBMS.

The Volcano research is unique as it combines architecture-independent parallelism with extensibility and data model-independence. Data model-independence is pursued far enough in both modules, the optimizer generator and the execution engine to allow integration and optimization of non-database operations. Our test cases will be the integration of numerical operations in scientific database systems and integrated data and compute servers [15, 41], and the use of the Volcano concepts and software as one main component of the request processing engine in an object-oriented database management system [7, 14, 18].

Acknowledgements

Bill McKenna has been implementing the Volcano optimizer generator and gave valuable comments on the section on query optimization. Diane Davison extended Volcano's original shared-memory exchange to distributed and hierarchical memory machines, and ported it from a network of Sun workstations to an Intel iPSC/2 hypercube. Henry Bremers designed and evaluated the skew-resistant one-to-many match scheme. — This manuscript is based on research partially funded by the National Science Foundation with grants IRI-8996270, IRI-8912618, and IRI-9006348, the Oregon Advanced Computing Institute (OACIS), ADP, Intel Supercomputer Systems Division, and Sequent Computer Systems.

References

1. W. Alexander and G. Copeland, ''Process and Dataflow Control in Distributed Data-Intensive Systems'', *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 90.
2. H. Boral, ''Parallelism in Bubba'', *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Austin, TX., December 1988, 68.
3. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith and P. Valduriez, ''Prototyping Bubba, A Highly Parallel Database System'', *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 4.
4. K. Bratbergsengen, ''Hashing Methods and Relational Algebra Operations'', *Proceedings of the Conference on Very Large Data Bases*, Singapore, August 1984, 323.
5. 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 (editor), Morgan Kaufman, San Mateo, CA., 1990.
6. G. Copeland, W. Alexander, E. Boughter and T. Keller, ''Data Placement in Bubba'', *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 99.
7. S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt and B. Vance, ''Query Optimization in Revelation, an Overview'', *IEEE Database Engineering 14*, 2 (June 1991).
8. D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, ''Implementation Techniques for Main Memory Database Systems'', *Proceedings of the ACM SIGMOD Conference*, Boston, MA., June 1984, 1.
9. D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar and M. Muralikrishna, ''GAMMA - A High Performance Dataflow Database Machine'', *Proceedings of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 228.
10. D. J. DeWitt, S. Ghandeharadizeh, D. Schneider, A. Bricker, H. I. Hsiao and R. Rasmussen, ''The Gamma Database Machine Project'', *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 44.
11. G. Graefe and D. J. DeWitt, ''The EXODUS Optimizer Generator'', *Proceedings of the ACM SIGMOD Conference*, San Francisco, CA., May 1987, 160.
12. G. Graefe, ''Rule-Based Query Optimization in Extensible Database Systems'', *Ph.D. Thesis, University of Wisconsin–Madison*, August 1987.
13. G. Graefe, ''Software Modularization with the EXODUS Optimizer Generator'', *IEEE Database Engineering 10*, 4 (December 1987).
14. G. Graefe and D. Maier, ''Query Optimization in Object-Oriented Database Systems: A Prospectus'', in *Advances in Object-Oriented Database Systems*, vol. 334 , K. R. Dittrich (editor), Springer-Verlag, September 1988, 358.
15. G. Graefe, ''DataCube: An Integrated Data and Compute Server Based on a Cube-Connected Dataflow Database Machine'', *Oregon Graduate Center, Computer Science Technical Report*, Beaverton, OR., July 1988.
16. G. Graefe and K. Ward, ''Dynamic Query Evaluation Plans'', *Proceedings of the ACM SIGMOD Conference*, Portland, OR, May-June 1989, 358.
17. G. Graefe, ''Relational Division: Four Algorithms and Their Performance'', *Proceedings of the IEEE Conference on Data Engineering*, Los Angelos, CA, February 1989, 94.
18. G. Graefe, D. Maier, S. Daniels and T. Keller, ''A Software Architecture for Efficient Query Processing in Object-Oriented Database Systems with Encapsulated Behavior'', *unpublished manuscript*, April 1990.
19. G. Graefe and S. S. Thakkar, ''Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor'', *CU Boulder Comp. Sci. Tech. Rep. 470*, April 1990.
20. G. Graefe and L. D. Shapiro, ''Full-Time Data Compression: An ADT for Database Performance'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 503*, December 1990.
21. G. Graefe, ''Encapsulation of Parallelism in the Volcano Query Processing System'', *Proceedings of the ACM SIGMOD Conference*, Atlantic City, NJ., May 1990, 102.
22. G. Graefe and D. L. Davison, ''Architecture-Independent Parallel Query Evaluation in Volcano'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 500*, December 1990.
23. G. Graefe, ''Parallel External Sorting in Volcano'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 459*, February 1990.
24. G. Graefe and L. D. Shapiro, ''Data Compression and Database Performance'', *Proc. ACM/IEEE-CS Symp. on Applied Computing*, Kansas City, MO, April 1991.
25. G. Graefe, A. Linville and L. D. Shapiro, ''Sort versus Hash Revisited'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 534*, July 1991.
26. G. Graefe, ''Volcano, An Extensible and Parallel Dataflow Query Processing System'', *accepted for publication in IEEE Transactions on Knowledge and Data Engineering*, . A more detailed version is available as CU Boulder

Computer Science Technical Report 481, July 1990.

27. L. Haas, W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey and E. Shekita, "Starburst Mid-Flight: As the Dust Clears", *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 143.
28. W. Hasan and H. Pirahesh, "Query Rewrite Optimization in Starburst", *Computer Science Research Report*, San Jose, CA., August 1988.
29. T. Keller and G. Graefe, "The One-to-One Match Operator of the Volcano Query Processing System", *Oregon Graduate Center, Computer Science Technical Report*, Beaverton, OR., June 1989.
30. T. Keller, G. Graefe and D. Maier, "Efficient Assembly of Complex Objects", *Proceedings of the ACM SIGMOD Conference*, Denver, CO, May 1991, 148.
31. S. Khoshafian and P. Valduriez, "Parallel Execution Strategies for Declustered Databases", *Proceedings of the 5th International Workshop on Database Machines*, Karuizawa, Japan, October 1987.
32. R. P. Kooi, "The Optimization of Queries in Relational Databases", *Ph.D. Thesis, Case Western Reserve University*, September 1980.
33. P. M. Neches, "Hardware Support for Advanced Data Management Systems", *IEEE Computer 17*, 11 (November 1984), 29.
34. P. M. Neches, "The Ynet: An Interconnect Structure for a Highly Concurrent Data Base Computer System", *Proc. 2nd Symp. on the Frontiers of Massively Parallel Computation*, Fairfax, October 1988.
35. K. Ono and G. M. Lohman, "Extensible Enumeration of Feasible Joins for Relational Query Optimization", *IBM Research Report RJ 6625 (63936)* (December 1988).
36. K. Ono and G. M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization", *Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, 1990, 314.
37. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database Management System", *Proceedings of the ACM SIGMOD Conference*, Boston, MA., May-June 1979, 23.
38. M. Stonebraker, P. Aoki and M. Seltzer, "Parallelism in XPRS", *UCB/Electronics Research Lab. Memorandum M89/16*, Berkeley, February 1989.
39. M. Stonebraker, R. Katz, D. Patterson and J. Ousterhout, "The Design of XPRS", *Proceedings of the Conference on Very Large Databases*, Long Beach, CA., August 1988, 318.
40. Teradata, *DBC/1012 Data Base Computer, Concepts and Facilities*, Teradata Corporation, Los Angeles, CA., 1983.
41. R. Wolniewicz and G. Graefe, "Automatic Optimization and Parallelization in Scientific Databases", *in preparation*, 1991.