The Volcano Optimizer Generator: Extensibility and Efficient Search

Goetz Graefe, Portland State University
William J. McKenna, University of Colorado at Boulder

From Proc. IEEE Conf. on Data Eng., Vienna, April 1993, p. 209.

Abstract

Emerging database application domains demand not only new functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. One of these tools is a new optimizer generator. Data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code. Compared with our earlier EXODUS optimizer generator prototype, the search engine is more extensible and powerful; it provides effective support for non-trivial cost models and for physical properties such as sort order. At the same time, it is much more efficient as it combines dynamic programming, which until now had been used only for relational select-project-join optimization, with goal-directed search and branch-and-bound pruning. Compared with other rule-based optimization systems, it provides complete data model independence and more natural extensibility.

Index Terms

Extensible database systems, query optimization, algebra, search, transformation rules, implementation rules, rewriting systems, dynamic programming, modularization, search efficiency.

1. Introduction

While extensibility is an important goal and requirement for many current database research projects and system prototypes, performance must not be sacrificed for two reasons. First, data volumes stored in database systems continue to grow, in many application domains far beyond the capabilities of most existing database systems. Second, in order to overcome acceptance problems in emerging database application areas such as scientific computation, database systems must achieve at least the same performance as the file systems currently in use. Additional software layers for database management must be counterbalanced by database performance advantages normally not used in these application areas. Optimization and parallelization are prime candidates to provide these performance advantages, and tools and techniques for optimization and parallelization are crucial for the wider use of extensible database technology.

For a number of research projects, namely the Volcano extensible, parallel query processor [GCD94], the REVELATION OODBMS project [MDK94] and optimization and parallelization in scientific databases [WoG93] as well as to assist research efforts by other researchers, we have built a new extensible query optimization system. Our earlier experience with the EXODUS optimizer generator had been inconclusive; while it had proven the feasibility and validity of the optimizer generator paradigm, it was difficult to construct efficient, production-quality optimizers. Therefore, we designed a new optimizer generator, requiring several important improvements over the EXODUS prototype. First, this new optimizer generator had to be usable both in the Volcano project with the existing query execution software as well as in other projects as a stand-alone tool. Second, the new system had to be more efficient, both in optimization time and in memory consumption for the search. Third, it had to provide effective, efficient, and extensible support for physical properties such as sort order and compression status. Fourth, it had to permit use of heuristics and data model semantics to guide the search and to prune futile parts of the search space. Finally, it had to support flexible cost models that permit generating dynamic plans for incompletely specified queries.

In this paper, we describe the Volcano Optimizer Generator, which will soon fulfill all the requirements above. Section 2 introduces the main concepts of the Volcano optimizer generator and enumerates facilities for tailoring a new optimizer. Section 3 discusses the optimizer search strategy in detail. Functionality, extensibility, and search efficiency of the EXODUS and Volcano optimizer generators are compared in Section 4. In Section 5, we describe and compare other research into extensible query optimization. We offer our conclusions from this research in Section 6.

2. The Outside View of the Volcano Optimizer Generator

In this section, we describe the Volcano optimizer generator as seen by the person who is implementing a database system and its query optimizer. The focus is the wide array of facilities given to the optimizer implementor, i.e., modularity and extensibility of the Volcano optimizer generator design. After considering the design principles of the Volcano optimizer generator, we discuss generator input and operation. Section 3 discusses the search strategy used by optimizers generated with the Volcano optimizer generator.

Figure 1 shows the optimizer generator paradigm. When the DBMS software is being built, a model specification is translated into optimizer source code, which is then compiled and linked with the other DBMS software such as the query execution engine. Some of this software is written by the optimizer implementor, e.g., cost functions. After a data model description has been translated into source code for the optimizer, the generated code is compiled and linked

$$\begin{tikzcd} & \text{Model Specification} \arrow[d, "\text{Optimizer Generator}"] & \\ & \text{Optimizer Source Code} \arrow[d, "\text{Compiler and Linker}"] & \\ \text{Query} \arrow[rr, "\text{Optimizer}"] & {} & \text{Plan} \end{tikzcd}$$

Figure 1. The Generator Paradigm.

with the search engine that is part of the Volcano optimization software. When the DBMS is operational and a query is entered, the query is passed to the optimizer, which generates an optimized plan for it. We call the person who specifies the data model and implements the DBMS software the "optimizer implementor." The person who poses queries to be optimized and executed by the database system is called the DBMS user.

2.1. Design Principles

There are five fundamental design decisions embodied in the system, which contribute to the extensibility and search efficiency of optimizers designed and implemented with the Volcano optimizer generator. We explain and justify these decisions in turn.

First, while query processing in relational systems has always been based on the relational algebra, it is becoming increasingly clear that query processing in extensible and object-oriented systems will also be based on algebraic techniques, i.e., by defining algebra operators, algebraic equivalence laws, and suitable implementation algorithms. Several object-oriented algebras have recently been proposed, e.g. [ShZ90, StO91, VaD91] among many others. Their common thread is that algebra operators consume one or more bulk types (e.g., a set, bag, array, time series, or list) and produce another one suitable as input into the next operator. The execution engines for these systems are also based on algebra operators, i.e., algorithms consuming and producing bulk types. However, the set of operators and the set of algorithms are different, and selecting the most efficient algorithms is one of the central tasks of query optimization. Therefore, the Volcano optimizer generator uses two algebras, called the logical and the physical algebras, and generates optimizers that map an expression of the logical algebra (a query) into an expression of the physical algebra (a query evaluation plan consisting of algorithms). To do so, it uses transformations within the logical algebra and cost-based mapping of logical operators to algorithms.

Second, rules have been identified as a general concept to specify knowledge about patterns in a concise and modular fashion, and knowledge of algebraic laws as required for equivalence transformations in query optimization can easily be expressed using patterns and rules. Thus, most extensible query optimization systems use rules, including the Volcano optimizer generator. Furthermore, the focus on independent rules ensures modularity. In our design, rules are translated independently from one another and are combined only by the search engine when optimizing a query. Considering that query optimization is one of the conceptually most complex components of any database system, modularization is an advantage in itself both for initial construction of an optimizer and for its maintenance.

Third, the choices that the query optimizer can make to map a query into an optimal equivalent query evaluation plan are represented as algebraic equivalences in the Volcano optimizer generator's input. Other systems use multiple intermediate levels when transforming a query into a plan. For example, the cost-based optimizer component of the extensible relational Starburst database system uses an "expansion grammar" with multiple levels of "non-terminals" such as commutative binary join, non-commutative binary join, etc. [Loh88]. We felt that multiple intermediate levels and the need to re-design them for a new or extended algebra confuse issues of equivalence, i.e., defining the choices open to the optimizer, and of search method, i.e., the order in which the optimizer considers possible query evaluation plans. Just as navigational query languages are less user-friendly than non-navigational ones, an extensible query optimization system that requires control information from the database implementor is less convenient than one that does not. Therefore, optimizer choices are represented in the Volcano optimizer generator's input file as algebraic equivalences, and the optimizer generator's search engine applies them in a suitable manner. However, for database implementors who wish to exert control over the search, e.g., who wish to specify search and pruning heuristics, there will be optional facilities to do so.

The fourth fundamental design decision concerns rule interpretation vs. compilation. In general, interpretation can be made more flexible (in particular the rule set can be augmented at run-time), while compiled rule sets typically execute faster. Since query optimization is very CPU-intensive, we decided on rule compilation similar to the EXODUS optimizer generator. Moreover, we believe that extending a query processing system and its optimizer is so complex and time-consuming that it can never be done quickly, making the strongest argument for an interpreter pointless. In order to gain additional flexibility with compiled rule sets, it may be useful to parameterize the rules and their conditions, e.g., to control the thoroughness of the search, and to observe and exploit repeated sequences of rule applications. In general, the issue of flexibility in the search engine and the choice between interpretation vs. compilation are orthogonal.

Finally, the search engine used by optimizers generated with the Volcano optimizer generator is based on dynamic programming. We will discuss the use of dynamic programming in Section 3.

2.2. Optimizer Generator Input and Optimizer Operation

Since one major design goal of the Volcano optimizer generator was to minimize the assumptions about the data model to be implemented, the optimizer generator only provides a framework into which an optimizer implementor can integrate data model specific operations and functions. In this section, we discuss the components that the optimizer implementor defines when implementing a new database query optimizer. The actual user queries and execution plans are input and output of the generated optimizer, as shown in Figure 1. All other components discussed in this section are specified by the optimizer implementor before optimizer generation in the form of equivalence rules and support functions, compiled and linked during optimizer generation, and then used by the generated optimizer when optimizing queries. We discuss parts of the operation of generated optimizers here, but leave it to the section on search to draw all the pieces together.

The user queries to be optimized by a generated optimizer are specified as an algebra expression (tree) of *logical operators*. The translation from a user interface into a logical algebra expression must be performed by the parser and is not discussed here. The set of logical operators is declared in the model specification and compiled into the optimizer during generation. Operators can have zero or more inputs; the number of inputs is not restricted. The output of the optimizer is a plan, which is an expression over the algebra of algorithms. The *set of algorithms*, their capabilities and their costs represents the data formats and physical storage structures used by the database system for permanent and temporary data.

Optimization consists of mapping a logical algebra expression into the optimal equivalent physical algebra expression. In other words, a generated optimizer reorders operators and selects implementation algorithms. The algebraic rules of expression equivalence, e.g., commutativity or associativity, are specified using *transformation rules*. The possible mappings of operators to algorithms are specified using *implementation rules*. It is important that the rule language allow for complex mappings. For example, a join followed by a projection (without duplicate removal) should be implemented in a single procedure; therefore, it is possible to map multiple logical operators to a single physical operator. Beyond simple pattern matching of operators and algorithms, additional conditions may be specified with both kinds of rules. This is done by attaching condition code to a rule, which will be invoke after a pattern match has succeeded.

The results of expressions are described using properties, similar to the concepts of properties in the EXODUS optimizer generator and the Starburst optimizer. *Logical properties* can be derived from the logical algebra expression and include schema, expected size, etc., while *physical properties* depend on algorithms, e.g., sort order, partitioning, etc. When optimizing a many-sorted algebra, the logical properties also include the type (or sort) of an intermediate result, which can be inspected by a rule's condition code to ensure that rules are only applied to expressions of the correct type. Logical properties are attached to equivalence classes – sets of equivalent logical expressions and plans – whereas physical properties are attached to specific plans and algorithm choices.

The set of physical properties is summarized for each intermediate result in a *physical property vector*, which is defined by the optimizer implementor and treated as an abstract data type by the Volcano optimizer generator and its search engine. In other words, the types and semantics of physical properties can be designed by the optimizer implementor.

There are some operators in the physical algebra that do not correspond to any operator in the logical algebra, for example sorting and decompression. The purpose of these operators is not to perform any logical data manipulation but to enforce physical properties in their outputs that are required for subsequent query processing algorithms. We call these operators *enforcers*; they are comparable to the "glue" operators in Starburst. It is possible for an enforcer to ensure two properties, or to enforce one but destroy another.

Each optimization goal (and subgoal) is a pair of a logical expression and a physical property vector. In order to decide whether or not an algorithm or enforcer can be used to execute the root node of a logical expression, a generated optimizer matches the implementation rule, executes the condition code associated with the rule, and then invokes an *applicability function* that determines whether or not the algorithm or enforcer can deliver the logical expression with physical properties that satisfy the physical property vector. The applicability functions also determine the physical property vectors that the algorithm's inputs must satisfy. For example, when optimizing a join expression whose result should be sorted on the join attribute, hybrid hash join does not qualify while merge-join qualifies with the requirement that its inputs be sorted. The sort enforcer also passes the test, and the requirements for its input do not include sort order. When the input to the sort is optimized, hybrid hash join qualifies. There is also a provision to ensure that algorithms do not qualify redundantly, e.g., merge-join must not be considered as input to the sort in this example.

After the optimizer decides to explore using an algorithm or enforcer, it invokes the algorithm's *cost function* to estimate its cost. Cost is an *abstract data type* for the optimizer generator; therefore, the optimizer implementor can choose cost to be a number (e.g., estimated elapsed time), a record (e.g., estimated CPU time and I/O count), or any other type. Cost arithmetic and comparisons are performed by invoking functions associated with the abstract data type "cost."

For each logical and physical algebra expression, logical and physical properties are derived using *property functions*. There must be one property function for each logical operator, algorithm, and enforcer. The *logical properties* are determined based on the logical expression, before any optimization is performed, by the property functions associated with the logical operators. For example, the schema of an intermediate result can be determined independently of which one of many equivalent algebra expressions creates it. The logical property functions also encapsulate selectivity estimation. On the other hand, *physical properties* such as sort order can only be determined after an execution plan has been chosen. As one of many consistency checks, generated optimizers verify that the physical properties of a chosen plan really do satisfy the physical property vector given as part of the optimization goal.

To summarize this section, the optimizer implementor provides (1) a set of logical operators, (2) algebraic transformation rules, possibly with condition code, (3) a set of algorithms and enforcers, (4) implementation rules, possibly with condition code, (5) an ADT "cost" with functions for basic arithmetic and comparison, (6) an ADT "logical properties," (7) an ADT "physical property vector" including comparisons functions (equality and cover), (8) an applicability function for each algorithm and enforcer, (9) a cost function for each algorithm and enforcer, (10) a property function for each operator, algorithm, and enforcer. This might seem to be a lot of code; however, all this functionality is required to construct a database query optimizer with or without an optimizer generator. Considering that query optimizers are typically one of the most intricate modules of a database management systems and that the optimizer generator prescribes a clean modularization for these necessary optimizer components, the effort of building a new database query optimizer using the Volcano optimizer generator should be significantly less than designing and implementing a new optimizer from scratch. This is particularly true since the optimizer implementor using the Volcano optimizer generator does not need to design and implement a new search algorithm.

3. The Search Engine

Since the general paradigm of database query optimization is to create alternative (equivalent) query evaluation plans and then to choose among the many possible plans, the search engine and its algorithm are central components of any query optimizer. Instead of forcing each database and optimizer implementor to implement an entirely new search engine and algorithm, the

Volcano optimizer generator provides a search engine to be used in all created optimizers. This search engine is linked automatically with the pattern matching and rule application code generated from the data model description.

Since our experience with the EXODUS optimizer generator indicated that it is easy to waste a lot of search effort in extensible query optimization, we designed the search algorithm for the Volcano optimizer generator to use dynamic programming and to be very goal-oriented, i.e., driven by needs rather than by possibilities.

Dynamic programming has been used before in database query optimization, in particular in the System R optimizer [SAC79] and in Starburst's cost-based optimizer [LFL88, Loh88], but only for relational select-project-join queries. The search strategy designed with the Volcano optimizer generator extends dynamic programming from relational join optimization to general algebraic query and request optimization and combines it with a top-down, goal-oriented control strategy for algebras in which the number of possible plans exceeds practical limits of pre-computation. Our dynamic programming approach derives equivalent expressions and plans only for those partial queries that are considered as parts of larger subqueries (and the entire query), not all equivalent expressions and plans that are feasible or seem interesting by their sort order [SAC79]. Thus, the exploration and optimization of subqueries and their alternative plans is tightly directed and very goal-oriented. In a way, while the search engines of the EXODUS optimizer generator as well as of the System R and Starburst relational systems use forward chaining (in the sense in which this term is used in AI), the Volcano search algorithm uses backward chaining, because it explores only those subqueries and plans that truly participate in a larger expression. We call our search algorithms *directed dynamic programming*.

Dynamic programming is used in optimizers created with the Volcano optimizer generator by retaining a large set of partial optimization results and using these earlier results in later optimization decisions. Currently, this set of partial optimization results is reinitialized for each query being optimized. In other words, earlier partial optimization results are used during the optimization of only a single query. We are considering research into longer-lived partial results in the future.

Algebraic transformation systems always include the possibility of deriving the same expression in several different ways. In order to prevent redundant optimization effort by detecting redundant (i.e., multiple equivalent) derivations of the same logical expressions and plans during optimization, expression and plans are captured in a hash table of expressions and equivalence classes. An equivalence class represents two collections, one of equivalent logical and one of physical expressions (plans). The logical algebra expressions are used for efficient and complete exploration of the search space, and plans are used for a fast choice of a suitable input plan that satisfies physical property requirements. For each combination of physical properties for which an equivalence class has already been optimized, e.g., unsorted, sorted on A, and sorted on B, the best plan found is kept.

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

```
FindBestPlan (LogExpr, PhysProp, Limit)
if the pair LogExpr and PhysProp is in the look-up table
   if the cost in the look-up 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 uses a transformation
      apply the transformation creating NewLogExpr
      call FindBestPlan (NewLogExpr, PhysProp, Limit)
   else if the move uses an algorithm
      TotalCost := cost of the algorithm
      for each input I while TotalCost ≤ Limit
         determine required physical properties PP for I
         Cost = FindBestPlan (I, PP, Limit − TotalCost)
         add Cost to TotalCost
   else /* move uses an enforcer */
      TotalCost := cost of the enforcer
      modify PhysProp for enforced property
      call FindBestPlan for LogExpr with new PhysProp
/* maintain the look-up table of explored facts */
if LogExpr is not in the look-up table
   insert LogExpr into the look-up table
insert PhysProp and best plan found into look-up table
return best Plan and Cost
```

Figure 2. Outline of the Search Algorithm.

example, sort order as in the ORDER BY clause of SQL), and a cost limit. This limit is typically infinity for a user query, but the user interface may permit users to set their own limits to "catch" unreasonable queries, e.g., ones using a Cartesian product due to a missing join predicate.

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 the 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 sets of 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., hybrid hash join for unsorted output and 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 hybrid hash join even if the final output is to be sorted.

After all possible moves have been generated and assessed, the most promising moves are pursued. Currently, with only exhaustive search implemented, all moves are pursued. In the future, a subset of the moves will be selected, determined and ordered by another function provided by the optimizer implementor. Pursuing all moves or only a selected few is a major heuristic placed into the hands of the optimizer implementor. In the extreme case, 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 query rewrite level. The difference between Starburst's two-level and Volcano's approach is that this separation is mandatory in Starburst while Volcano will leave it as a choice to be made by the optimizer implementor.

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 (the user query or some part of it) and a physical property vector, no other 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 speed 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.

Often a new equivalence class is created during a transformation. Consider the associativity rule in Figure 3. The expressions rooted at A and B are equivalent and therefore belong into the same class. However, expression C is not equivalent to any expression in the left expression and requires a new equivalence class. In this case, a new equivalence class is created and optimized as required for cost analysis and optimization of expression B.

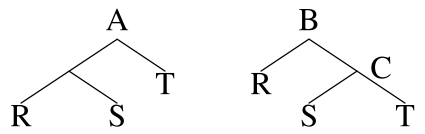

Figure 3. Associativity Rule.

If a move to be pursued is the exploration of a normal query processing algorithm such as merge-join, its cost is calculated by the algorithm's cost function. The algorithm's applicability function determines the physical property vectors for the algorithms inputs, and their costs and optimal plans are found by invoking FindBestPlan for the inputs.

For some binary operators, the actual physical properties of the inputs are not as important as the consistency of physical properties among the inputs. For example, for a sort-based implementation of intersection, i.e., an algorithm very similar to merge-join, any sort order of the two inputs will suffice as long as the two inputs are sorted in the same way. Similarly, for a parallel join, any partitioning of join inputs across multiple processing nodes is acceptable if both inputs are partitioned using compatible partitioning rules. For these cases, the search engine permits the optimizer implementor to specify a number of physical property vectors to be tried. For example, for the intersection of two inputs R and S with attributes A, B, and C where R is sorted on (A,B,C) and S is sorted on (B,A,C), both these sort orders can be specified by the optimizer implementor and will be optimized by the generated optimizer, while other possible sort orders, e.g., (C,B,A), will be ignored.

If the move to be pursued is the use of an enforcer such as sort, its cost is estimated by a cost function provided by the optimizer implementor and the original logical expression is optimized using FindBestPlan with a suitably modified (i.e., relaxed) physical property vector. In many respects, enforcers are dealt with exactly like algorithms, which is not surprising considering that both are operators of the physical algebra. During optimization with the modified 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 the sort requirement, hybrid 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, which includes both plans optimal for given physical properties as well as failures that can save future optimization effort for a logical expression and a physical property vector with the same or even lower cost limits.

In summary, the search algorithm employed by optimizers created with the Volcano optimizer generator uses dynamic programming by storing all optimal subplans as well as optimization failures until a query is completely optimized. Without any a-priori assumptions about the algebras themselves, it is designed to map an expressions over the logical algebra into the optimal equivalent expressions over the physical algebra. Since it is very goal-oriented through the use of physical properties and derives only those expressions and plans that truly participate in promising larger plans, the algorithm is more efficient than previous approaches to using dynamic programming in database query optimization.

4. Comparison with the EXODUS Optimizer Generator

Since the EXODUS optimizer generator was our first attempt to design and implement an extensible query optimization system or tool, this section compares the EXODUS and Volcano optimizer generators in some detail. The EXODUS optimizer generator was successful to the extent that it defined a general approach to the problem based on query algebras, the generator paradigm (data model specification as input data), separation of logical and physical algebras, separation of logical and physical properties, extensive use of algebraic rules (transformation rules and implementation rules), and its focus on software modularization [Gra87, GrD87]. Considering the complexity of typical query optimization software and the importance of well-defined modules to conquer the complexities of software design and maintenance, the latter two points might well be the most important contributions of the EXODUS optimizer generator research.

The generator concept was very successful because the input data (data model specification) could be turned into machine code; in particular, all strings were translated into integers, which ensured very fast pattern matching. However, the EXODUS optimizer generator's search engine was far from optimal. First, the modifications required for unforeseen algebras and their peculiarities made it a bad patchwork of code. Second, the organization of the "MESH" data structure (which held all logical and physical algebra expressions explored so far) was extremely cumbersome, both in its time and space complexities. Third, the almost random transformations of expressions in MESH resulted in significant overhead in "reanalyzing" existing plans. In fact, for larger queries, most of the time was spent reanalyzing existing plans.

The Volcano optimizer generator has solved these three problems, and includes new functionality not found in the EXODUS optimizer generator. We first summarize their differences in functionality and then present a performance comparison for relational queries.

4.1. Functionality and Extensibility

There are several important differences in the functionality and extensibility of the EXODUS and Volcano optimizer generators. 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 such as join and a physical algorithm such as hybrid hash join. To retain equivalent plans using merge-join and hybrid 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, physical properties were handled rather haphazardly in EXODUS. If the algorithm with the lowest cost happened to deliver results with useful physical properties, this was recorded in MESH and used in subsequent optimization decisions. Otherwise, the cost of enforcers (to use 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 these properties, together with the logical expression, drive the optimization process was entirely absent in

EXODUS and has contributed significantly to the efficiency of the Volcano optimizer generator search engine.

The concept of physical property is very powerful and extensible. The most obvious and well-known candidate for a physical property in database query processing is the sort order of intermediate results. Other properties can be defined by the optimizer implementor at will. Depending on the semantics of the data model, uniqueness might be a physical property with two enforcers, sort- and hash-based. Location and partitioning in parallel and distributed systems can be enforced with a network and parallelism operator such as Volcano's *exchange* operator [GCD94]. For query optimization in object-oriented systems, we plan on defining "assembledness" of complex objects in memory as a physical property and using the assembly operator described in [KGM91] as the enforcer for this property.

Third, the Volcano algorithm is driven top-down; subexpressions are optimized only if warranted. In the extreme case, if the only move pursued is a transformation, a logical expression is transformed on the logical algebra level without optimizing its subexpressions and without performing algorithm selection and cost analysis for the subexpressions. In EXODUS, a transformation is 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 expected 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 an extremely large number of MESH nodes.

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 for which all calculations and comparisons are performed by invoking functions provided by the optimizer implementor. It can be a simple number, e.g., estimated elapsed seconds, a structure, e.g., a record consisting of CPU time and I/O count for a cost model similar to the one in System R [SAC79], or even a function, e.g., of the amount of available main memory.

Finally, we believe that the Volcano optimizer generator is more extensible than the EXODUS prototype, 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 outlined in the previous section. We are still modifying (extending and refining) the search strategy, and plan on modifying it further in subsequent years and on using the Volcano optimizer generator for further research.

4.2. Search Efficiency and Effectiveness

In this section, we experimentally compare efficiency and effectiveness of the mechanisms built into the EXODUS and Volcano search engines. The example used for this comparison is a rather small "data model" consisting of relational select and join operators only; as we will see, however, even this small data model and query language suffices to demonstrate that the search strategy of the Volcano optimizer generator is superior to the one designed for the earlier EXODUS prototype. The effects exposed here would be even stronger for richer and more complex data models, (logical) query algebras, and (physical) execution algebras.

For the experiments, we specified the data model descriptions as similarly as possible for the EXODUS and Volcano optimizer generators. In particular, we specified the same operators (get, select, join) and algorithms (file scan, filter for selections, sort, merge-join, hybrid hash join), the same transformation and implementation rules, and the same property and cost functions. Sorting was modeled as an enforcer in Volcano while it was implicit in the cost function for merge-join in EXODUS. The transformation rules permitted generating all plans including bushy ones (composite inner inputs). The test relations contained 1,200 to 7,200 records of 100 bytes. The cost functions included both I/O and CPU costs. Hash join was presumed to proceed without partition files, while sorting costs were calculated based on a single-level merge.

As a first comparison between the two search engines, we performed exhaustive optimizations of relational select-join queries. Figure 4 shows the average optimization effort and, to show the quality of the optimizer output, the estimated execution time of produced plans for queries with 1 to 7 binary joins, i.e., 2 to 8 input relations, and as many selections as input relations. Solid lines indicate optimization times on a Sun SparcStation-1 delivering about 12 MIPS. Dashed lines indicate estimated plan execution times. Note that the y-axis are logarithmic. Measurements from the EXODUS optimizer generator are marked with □'s, Volcano measurements are marked with ○'s.

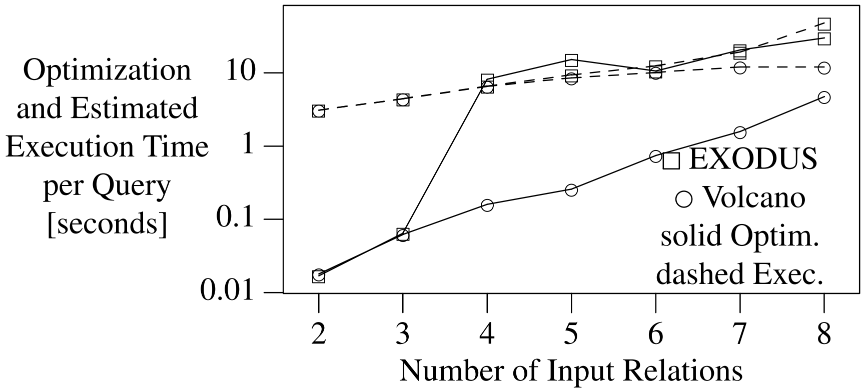

Figure 4. Exhaustive Optimization Performance.

For each complexity level, we generated and optimized 50 queries. For some of the more complex queries, the EXODUS optimizer generator aborted due to lack of memory or was aborted because it ran much longer than the Volcano optimizer generator. Furthermore, we observed in repeated experiments that the EXODUS optimizer generator measurements were quite volatile. Similar problems were observed in EXODUS experiments reported in [GrD87]. The Volcano-generated optimizer performed exhaustive search for all queries with less than 1 MB of work space. The data points in Figure 4 represent only those queries for which the EXODUS optimizer generator completed the optimization.

The search times reflect Volcano's more efficient search strategy, visible in the large distance between the two solid lines. For the EXODUS-generated optimizer, the search effort increases dramatically from 3 to 4 input relations because reanalyzing becomes a substantial part of the query optimization effort in EXODUS at this point. The increase of Volcano's optimization costs is about exponential, shown in an almost straight line, which mirrors exactly the increase in the number of equivalent logical algebra expressions [OnL90]. For more complex queries, the EXODUS' and Volcano's optimization times differ by about an order of magnitude.

The plan quality (shown by the estimated execution cost; dashed lines in Figure 4) is equal for moderately complex queries (up to 4 input relations). For more complex queries, however, the cost is significantly higher for EXODUS-optimized plans, because the EXODUS-generated optimizer and its search engine do not systematically explore and exploit physical properties and interesting orderings.

In summary, the Volcano optimizer generator is not only more extensible, it is also much more efficient and effective than the earlier EXODUS prototype. In the next section, we compare our work with other related work.

5. Other Related Work

The query optimizer of the Starburst extensible-relational database management system consists of two rule-based subsystems with nested scopes. The two subsystems are connected by a common data structure, which represents an entire query and is called query graph model (QGM). The first subsystem, called query rewrite, merges nested subqueries and bundles selection and join predicates for optimization in a second, cost-based optimizer. Optimization during the query rewrite phase, i.e., nested SQL queries, union, outer join, grouping, and aggregation, is based entirely on heuristics and is not cost-sensitive. Select-project-join query components are covered by the second optimizer[1], also called the cost-based optimizer, which performs rule-based expansion of select-project-join queries from relational calculus into access plans and compares the resulting plans by estimated execution costs [LFL88, Loh88]. The cost-based optimizer performs exhaustive search within certain structural boundaries. For example, it

[1] Actually, the cost-based optimizer covers all operators. However, its optimization and algorithm choices are very limited for all but the select-project-join blocks in a query.

is possible to restrict the search space to left-deep trees (no composite inner), to include all bushy trees, or to set a parameter for exploration of some but not all bushy trees. For moderately complex join queries, the exhaustive search of Starburst's cost-based optimizer is very fast because of its use of dynamic programming. Moreover, the cost-based optimizer considers physical properties such as sort order and creates efficient access plans that include "glue" operators to enforce physical properties.

As we see it, there are two fundamental problems in Starburst's approach to extensible query optimization. First, the design of the cost-based optimizer is focused on step-wise expansion of join expressions based on grammar-like rules. The "grammar" depends on a hierarchy of intermediate levels (similar to non-terminals in a parsing grammar), e.g., commutative join and non-commutative join, and the sets of rules and intermediate levels are tailored specifically to relational join optimization. The problem is that it is not obvious how the existing rule set would interact with additional operators and expansion rules. For example, which level of the hierarchy is the right place for a multi-input join algorithm? What new intermediate levels (non-terminals) must be defined for the expansion grammar? In order to integrate a new operator into Starburst's cost-based optimizer, the database implementor must design a number of new intermediate levels and their new grammar rules. These rules may interact with existing ones, making any extension of Starburst's cost-based optimizer a complex and tedious task. Volcano's algebraic approach seems much more natural and easier to understand. Most recent work in object-oriented query optimization and some work on database programming languages has focused on algebras and algebraic transformations, e.g. [LiD92, ShZ90, StO91, VaD91] among many others.

Second, in order to avoid the problems associated with adding new operators to the cost-based optimizer, new operators are integrated at the query rewrite level. However, query optimization on the query rewrite level is heuristic; in other words, it does not include cost estimation. While heuristics are sufficient for some transformations, e.g., rewriting nested SQL queries into join expressions, they are not sufficient for the relational operators already in Starburst's query rewrite level and certainly not for an extensible query optimization system in which future algebra operators and their properties are yet unknown. As an example for insufficient optimization capabilities for existing Starburst operators, consider that optimizing the union or intersection of N sets is very similar to optimizing a join of N relations; however, while join optimization uses exhaustive search of tree shapes and join orderings as well as selectivity and cost estimation, union and intersection are optimized using query rewrite heuristics and commutativity only. We believe that a single-level approach, in which all algebraic equivalences and transformations are specified in a single language and performed by a single optimizer component, is much more conducive for future research and exploration of database query algebras and their optimization. Note that the Volcano optimizer generator will permit heuristic transformations by suitable ranking and selection of "moves"; however, it leaves the choice to the database implementor when and how to use heuristics vs. cost-sensitive optimization rather than making this choice a priori as in the Starburst design.

Sciore and Sieg criticized earlier rule-based query optimizers and concluded that modularity is a major requirement for extensible query optimization systems, e.g., in the rule set and in the control structures for rule application [ScS90]. The different tasks of query optimization, such as rule application and selectivity estimation, should be encapsulated in separate and cooperating "experts." Mitchell et al. recently proposed a very similar approach for query optimization in object-oriented database systems [MZD92]. While promising as a conceptual approach, we feel that this separation can be sustained for some aspects of query optimization (and have tried to do so in the abstract data types for cost etc. in the Volcano optimizer generator), but we have found it extremely hard to maintain encapsulation of all desirably separate concerns in an actual implementation.

Kemper and Moerkotte designed a rule-based query optimizer for the Generic Object Model [KeM90a]. The rules operate almost entirely on path expressions (e.g., employee.department.floor) by extending and cutting them to permit effective use of access support relations [KeM90b]. While the use of rules makes the optimizer extensible, it is not clear to what extent these techniques can be used for different data models and for different execution engines.

6. Summary and Conclusions

Emerging database application domains demand not only high functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. 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 query and request languages are and will continue to be based on sets, other bulk types, predicates, and operators. Therefore, 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 collections, including heterogeneous collections and many-sorted algebras. 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 of parallel algorithms for database query processing. Thus, our fundamental assumption for query processing in extensible database systems are compatible with high-performance parallel processing.

One of the tools provided by the Volcano research is a new optimizer generator, designed and implemented to further explore extensibility, search algorithms, effectiveness (i.e., the quality of produced plans), heuristics, and time and space efficiency in the search engine. Extensibility was achieved by generating optimizer source code from data model specifications and by encapsulating costs as well as logical and physical properties into abstract data types. Effectiveness was achieved by permitting exhaustive search, which will be pruned only at the discretion of the optimizer implementor. Efficiency was achieved by combining dynamic programming with directed search based on physical properties, branch-and-bound pruning, and heuristic guidance into a new search algorithm that we have called *directed dynamic* *programming*. A preliminary performance comparison with the EXODUS optimizer generator demonstrated that optimizers built with the Volcano optimizer generator are much more efficient than those built with the EXODUS prototype. We hope that the new Volcano optimizer generator will permit our own research group as well as others to develop more rapidly new database query optimizers for novel data models, query algebras, and database management systems. The Volcano optimizer generator has been used to develop optimizers for computations over scientific databases [WoG93] and for Texas Instruments's Open OODB project [BMG93, WBT92], which introduces a new "materialize" or scope operator that captures the semantics of path expressions in a logical algebra expression. Both of these optimizers have recently become operational. Moreover, the Volcano optimizer generator is currently being evaluated by several academic and industrial researchers in three continents.

In addition to combining an efficient implementation of exhaustive search based on dynamic programming (as also found in the cost-based component of the Starburst's relational optimizer) with the generality of the EXODUS optimizer generator and the more natural single-level algebraic transformation approach, the Volcano optimizer generator has a number of new features that enhance its value as a software development and research tool beyond all earlier extensible query optimization efforts.

First, the choice when and how to use heuristic transformations vs. cost-sensitive optimization is not prescribed or "wired in." In EXODUS, cost analysis was always performed after a transformation; in Starburst, one level can only perform heuristic optimization while the other level performs cost-sensitive exhaustive search. Thus, the Volcano optimizer generator has removed the restrictions on the search strategy imposed by the earlier extensible query optimizer designs.

Second, optimizers generated with the Volcano optimizer generator use physical properties very efficiently to direct the search. Rather than optimizing an expression first and then adding "glue" operators and their cost to a plan (the Starburst approach), the Volcano optimizer generator's search algorithm immediately considers which physical properties are to be enforced and can be enforced by which enforcer algorithms, and subtracts the cost of the enforcer algorithms from the bound that is used for branch-and-bound pruning. Thus, the Volcano optimizer generator promises to be even more efficient in its search and pruning than the relational Starburst optimizer.

Third, for binary (ternary, etc.) operations that can benefit from multiple, alternative combinations of physical properties, the subexpressions can be optimized multiple times. For example, any sort order can be exploited by an intersection algorithm based on merge-join as long as the two inputs are sorted in the same way. Although the same consideration applies to location and partitioning in parallel and distributed relational query processing, no earlier query optimizer has provided this feature.

Fourth, the internal structure for equivalence classes is sufficiently modular and extensible to support alternative search strategies, far beyond the parameterization of rule condition codes, which can be found to a roughly similar extent in Starburst and EXODUS. We are exploring several directions with respect to the search strategy, namely preoptimized subplans, learning of transformation sequences, an alternative, even more parameterized search algorithm that can be "switched" to different existing algorithms, and parallel search (on shared-memory machines).

Finally, the consistent separation of logical and physical algebras makes specification as well as modifications at either level particularly easy for the database and optimizer implementor and makes the search engine very efficient. For example, the introduction of a new, non-trivial algorithm such as a N-ary join (rather than binary joins) requires one or two implementation rules in Volcano, whereas the design of Starburst's cost-based optimizer requires reconsideration of almost the entire rule set. While the separation of logical and physical algebras was already present in the EXODUS rule language, the Volcano design also exploits this separation in the search engine, which makes extending the code supplied by the optimizer implementor (which sometimes must inspect the internal data structures, e.g., in rule condition code) significantly easier to write, understand, and modify. In summary, the Volcano optimizer generator is a much more extensible and useful research tool than both the Starburst optimizer and the EXODUS optimizer generator.

Acknowledgements

Jim Martin, David Maier, and Guy Lohman have made valuable contributions to this research. Jim Martin, Guy Lohman, Barb Peters, Rick Cole, Diane Davison, and Richard Wolniewicz suggested numerous improvements to drafts of this paper. We thank José Blakeley and his colleagues at Texas Instruments for using the Volcano Optimizer Generator in the Open OODB project. – This research was performed at the University of Colorado at Boulder with partial support by NSF with awards IRI-8805200, IRI-8912618, and IRI-9116547, ARPA with contract DAAB07-91-C-Q518, and Texas Instruments.

References

[BMG93] J. A. Blakeley, W. J. McKenna, and G. Graefe, Experiences Building the Open OODB Query Optimizer, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 287.

[Gra87] G. Graefe, Software Modularization with the EXODUS Optimizer Generator, *IEEE Database Eng. Bull. 10*, 4 (December 1987), .

[GrD87] G. Graefe and D. J. DeWitt, The EXODUS Optimizer Generator, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 160.

[GCD94] G. Graefe, R. L. Cole, D. L. Davison, W. J. McKenna, and R. H. Wolniewicz, *Extensible Query Optimization and Parallel Execution in Volcano*, Morgan-Kaufman, San Mateo, CA, 1994.

[KGM91] T. Keller, G. Graefe, and D. Maier, Efficient Assembly of Complex Objects, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 148.

[KeM90a] A. Kemper and G. Moerkotte, Advanced Query Processing in Object Bases Using Access Support Relations, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 290.

[KeM90b] A. Kemper and G. Moerkotte, Access Support in Object Bases, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 364.

[LFL88] M. Lee, J. C. Freytag, and G. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 218.

[LiD92] D. F. Lieuwen and D. J. DeWitt, A Transformation-Based Approach to Optimizing Loops in Database Programming Languages, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 91.

[Loh88] G. M. Lohman, Grammar-Like Functional Rules for Representing Query Optimization Alternatives, *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 18.

[MDK94] D. Maier, S. Daniels, T. Keller, B. Vance, G. Graefe, and W. McKenna, Challenges for Query Processing in Object-Oriented Databases, in *Query Processing for Advanced Database Applications*,

J. C. Freytag, G. Vossen and D. Maier (ed.), Morgan-Kaufman, San Mateo, CA, 1994, 337.

[MZD92] G. Mitchell, S. B. Zdonik, and U. Dayal, An Architecture for Query Processing in Persistent Object Stores, *Proc. Hawaii Conf. on System Sciences, Vol. 2*, January 1992, 787.

[OnL90] K. Ono and G. M. Lohman, Measuring the Complexity of Join Enumeration in Query Optimization, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 314.

[ScS90] E. Sciore and J. Sieg, A Modular Query Optimizer Generator, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 146.

[SAC79] 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, *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 23. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

[ShZ90] G. M. Shaw and S. B. Zdonik, A Query Algebra for Object-Oriented Databases, *Proc. IEEE Int'l. Conf. on Data Eng.*, Los Angeles, CA, February 1990, 154.

[StO91] D. D. Straube and M. T. Ozsu, Execution Plan Generation for an Object-Oriented Data Model, *Proc. Conf. on Deductive and Object-oriented Databases*, Munich, Germany, December 1991.

[VaD91] S. L. Vandenberg and D. J. DeWitt, Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 158.

[WBT92] D. Wells, J. A. Blakeley, and C. W. Thompson, Architecture of an Open Object-Oriented Database Management System, *IEEE Computer 25*, 10 (October 1992), 74.

[WoG93] R. H. Wolniewicz and G. Graefe, Algebraic Optimization of Computations over Scientific Databases, *Proc. Int'l. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993, 13.

Table of Contents

List of Figures