# The Volcano Optimizer Generator

Goetz Graefe, William McKenna

CU-CS-563-91 December 1991

University of Colorado at Boulder

DEPARTMENT OF COMPUTER SCIENCE

# The Volcano Optimizer Generator

Goetz Graefe, William McKenna
University of Colorado at Boulder
Computer Science Technical Report 563
December 1991

## Abstract

Novel database applications demand not only high functionality but also high performance. To combine these two requirements, the Volcano project provides efficient, extensible tools for query and request processing in novel application domains, particularly in object-oriented and scientific database systems. One of these tools is a new optimizer generator. Data model, logical algebra, physical algebra, and 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 as it provides direct and effective support for non-trivial cost models and for physical properties like sort order and partitioning, but at the same time it is much more efficient. Compared with other rule-based optimization systems, it provides more extensibility and data model independence. The Volcano optimizer generator and its search engine demonstrate the feasibility of a query optimization system that is highly extensible but also very effective and efficient.

## 1. Introduction

Numerous database systems are currently being developed, most of them from scratch rather than by using an extensible toolkit [27], because extensible database technology currently does not combine the required levels of generality and performance. The basic idea of extensible database systems is to divide DBMS functions into well-defined components and for each of them either to generate code from specifications or to provide a toolkit or "grab bag" of alternative modules. The former approach has been used for parsing and query optimization, while the latter approach seems particularly suitable for the storage and access mechanisms. Examples of code generators include parser generators like YACC [15] and optimizer generators like the EXODUS optimizer generator [4]; extensible toolkits include Genesis [1] and the attachment mechanisms in Starburst [13].

While extensibility is an important requirement for database research, performance must not be sacrificed for two reasons. First, data volumes stored in database systems continue to grow, in many application areas far beyond the capabilities of most existing database systems [26]. Second, in order to overcome acceptance problems in novel database application domains, database systems must achieve at least the same performance as the file systems currently in use. In other words, the additional software layers for database management must be counterbalanced by database performance techniques normally not used in these application areas. We believe that optimization and parallelization are prime candidates, and that 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 [11], the REVELATION OODBMS project [2, 7], optimization and parallelization in scientific databases [31], and dynamic query plans [8], we felt the need for an extensible query optimization system. Our earlier experience with the EXODUS optimizer generator had been inconclusive as it had proved the feasibility and validity of the optimizer generator paradigm, but it had been difficult to construct efficient, production-quality optimizers. Therefore, we set out to design a new optimizer generator, requiring several important improvements over the EXODUS prototype. First, the new system must be more efficient, both in optimization time and memory consumption for the search.

Second, it must provide effective, efficient, and extensible support for physical properties like sort order, location and partitioning, compression status, and "assembledness" in object-oriented systems [16]. Third, it must permit extensive use of heuristics and data model semantics to guide the search and to prune futile parts of the search space. Fourth, it must learn optimization heuristics, both from observations during the search and from observations of actual query performance. Finally, it must support flexible cost models that could be used for generating dynamic plans for incompletely specified queries.

In this paper, we describe the Volcano optimizer generator which fulfills or shortly will fulfill all the requirements above. In Section 2, we describe related work on rule-based extensible query optimization. Section 3 introduces the main concepts of the Volcano optimizer generator and details how the optimizer implementor can tailor a new optimizer. Section 4 discusses the optimizer search strategy in detail. Functionality, extensibility, and search efficiency of the EXODUS and Volcano optimizer generators is compared in Section 5. We offer our conclusions from this research in Section 6.

## 2. Related Work

The EXODUS optimizer generator was a first attempt at extensible query optimization. It was successful in the sense that it defined a general approach to the problem based on the generator paradigm (data model specification as input data), separation of logical and physical algebra, extensive use of rules (transformation rules, implementation rules), and its focus on software modularization [4-6]. Considering the complexity of typical query optimization software and the importance of well-defined components to conquer complexities of software design and maintenance, the latter point might be one of 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, allowing very fast pattern matching. However, the EXODUS generator's search engine was far from optimal. First, the modifications required for unforeseen algebras and their peculiarities made it quite a patchwork of code [6]. Second, the organization of the "MESH" data structure (that 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 query optimization subsystem of the Starburst extensible-relational database management system consists of two rule-based subsystems with nested scopes [3, 12, 13, 20]. The first one, called query rewrite or query graph model (QGM), covers all operators except select, project, and join. Optimization of query rewrite operators, e.g., nested SQL queries, union, outer join, grouping, and aggregation, is based entirely on heuristics and is not cost sensitive. Select-project-join (SPJ) query components are covered by the second optimizer, also called the join enumerator, which performs rule-based, grammar-like, cost-sensitive expansion of SPJ queries from relational calculus into access plans. The join enumerator performs exhaustive search within certain structural boundaries. For example, it is possible to restrict the search space to left-deep trees (no composite inner), to include all bushy trees, or to set a parameter that explores some but not all bushy trees. However, the optimizer design is focussed on step-wise expansion of join expressions based on grammar-like rules, and it is not obvious how the existing rule set would interact with additional operators and expansion rules.

Kemper and Moerkotte designed a rule-based query optimizer for the Generic Object Model [17]. 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 [18]. 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.

In summary, we believe that the extensibility of the EXODUS optimizer generator and its modular software engineering approach are important for truly extensible systems. On the other hand, the carefully designed search strategy of Starburst's join enumerator is probably much more efficient than the EXODUS optimizer generator's very general search engine. Therefore, we tried to combine these two key advantages, generality and efficiency, in our new optimizer generator and its search engine.

## 3. The Outside View of the Volcano Optimizer Generator

In this section, we provide a general overview of the Volcano optimizer generator. There are three fundamental design decisions embodied in the system. First, query processing, both optimization and execution, are presumed to be based on algebraic techniques. Second, rules are used to specify the data model and its properties. Third, rules are transformed by an optimizer generator into source code in a standard programming language (C) to be compiled and linked with the other DBMS modules.

While query processing in relational systems has always been based on relational algebra, it is becoming increasingly clear that query processing in extensible and object-oriented systems will also be based on algebraic techniques. Several object-oriented algebras have recently been proposed, e.g. [25, 28-30]. Their common thread is that algebra operators consume one or more bulk types (e.g., set, bag, array, 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 will be one of the tasks of query optimization. Therefore, the Volcano optimizer generator uses two algebras, called the logical and the physical algebra, and generates optimizers that map an expression of the logical into an expression of the physical algebra using 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. For an extensible query optimization system, the properties of the data model and its execution engine must be imported knowledge (input data), suggesting a separation of rule specification and rule execution as realized in the generator and its associated search engine. Furthermore, in order to permit extensions of an existing optimizer, all input to the generator must be modular. 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 maintenance. In our design, rules are translated independently from one another and are combined only by the search engine.

The third 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. Earlier experiments demonstrated that query optimization is very processing-intensive; therefore, 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 is never done on the fly, making the strongest argument for an interpreter pointless. Nonetheless, we have been using an interpreted Lisp prototype for preliminary algorithm studies because Lisp offers both powerful pattern matching and algorithmic control. Since these studies were completed about a year ago, we have been developing and working on the optimizer generator.

## 3.1. The Optimizer Generator Paradigm

Figure 1 shows the general generator paradigm. While building the DBMS software, a model specification is translated into optimizer source code which is then compiled and linked with the other software. Some of this software is written by the optimizer implementor, e.g., cost functions. When the DBMS is used 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.

The correctness of the transformation rules as well as their soundness and completeness[1] cannot be verified by the optimizer generator since these rules are the only source of knowledge about the data model, its operations, and its equivalence criteria. However, we plan on complementing the Volcano optimizer generator with an "advisor" that scans the rule set and suggests modifications to the optimizer implementor, e.g., rules that seem redundant, missing implementation algorithms for logical operators, etc. We feel that such a separate advisor is an additional advantage of the generator paradigm.

After a data model description has been translated into source code for the optimizer, the generated code is compiled and linked with the search engine (code) that is associated with and part of the Volcano optimization software. The search engine is compiled each time a new optimizer is built; while this is not strictly necessary, it was convenient to design the system in this way because a number of constants that are derived from the model description, e.g., the maximal operator arity, can be compiled into the optimizer. We have considered modifying the search engine such that it can be compiled only once; while this would make the turn-around time during optimizer implementation and debugging faster, it would require dynamic space allocation for several data structures, therefore slowing the generated optimizers to some extent.

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

Figure 1. The Generator Paradigm.

[1] Sound — only correct expressions and plans, i.e., equivalent to the original user query, can be derived. Complete — all correct expressions and plans can be derived.

## 3.2. 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 can define when implementing a new database query optimizer. We discuss parts of the operation of generated optimizers here, but leave it to the section on search to draw all the pieces together. In order to ensure a common language, Table 1 summarizes the most important terms used throughout this paper and introduced in turn in this section.

The user queries to be optimized are specified using an 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. 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 algebra expression of algorithms.

Optimization consists of mapping a logical algebra expression into the optimal equivalent physical algebra expression. In other words, the 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. Notice that these are rules

| | |
|---|---|
| Query | Logical algebra expression. |
| Logical algebra, operator | High-level operators that specify data transformations without specifying the algorithm to be used; for example, relational algebra. |
| Plan | Physical algebra expression. |
| Physical algebra, algorithm | Algorithms for data transformations, composable into complex query evaluation plans. |
| Transformation rules | Rules that govern rewriting logical algebra expressions; e.g., join commutativity. |
| Implementation rules | Rules that specify the relationships between logical operators and physical operators; e.g., (logical operator) join and (physical algorithm) hybrid hash join. |
| Condition code | Determines, after a rule pattern match has succeeded, whether or not a transformation rule, algorithm, or enforcer is applicable to a logical algebra expression. |
| Property | Description of results of (sub-) queries and (sub-) plans; divided into logical, system, and physical properties. |
| Property vector | Indicates which physical properties must be enforced during optimization of a logical algebra expression, e.g., sortedness. |
| Enforcer | Part of the physical algebra that enforces physical properties like sort order, location in a network, or decompression. |
| Applicability function | Decides whether an algorithm or enforcer can deliver a logical expression conforming to a physical property vector, and determines what physical properties its inputs must satisfy. |
| Cost, Cost function | An abstract data type that captures an algorithm's or plan's cost; typically a number or a record. A cost function estimates the cost of a algorithm or enforcer. |
| Property function | One for each operator and algorithm, they determine logical, system, and physical properties. |

Table 1. Optimizer Generator Concepts.

to allow for complex mappings. For example, a join followed by a projection (without duplicate removal) should be implemented in a single procedure. Beyond simple pattern matching of operators and algorithms, additional conditions may be specified with both kinds of rules using condition code attached to a rule to be invoked after a pattern match succeeded.

The results of expressions are described using properties. Logical properties can be derived from the logical algebra expression alone and include the schema, expected size, etc., while physical properties depend on algorithms, e.g., sortedness, partitioning, etc. Logical properties are attached to equivalence classes — sets of equivalent logical expressions and plans — while physical properties are attached to specific plans and algorithm choices. Since we expect some logical properties to be used in any database query optimizer, we defined them separately as system properties. We hope to use system properties later for learning search heuristics automatically, for example, the fact that hash join is only worthwhile if the left (build) input is smaller than the right (probe) input. Currently, only item count (i.e., cardinality) and average item size (equivalent to record size) are system properties, but we may decide to revisit this design decision.

The set of physical properties is summarized for each intermediate result in a property vector. The property vector 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 number of the fields in this vector, their types, and their semantics 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 "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 to determine whether or not the algorithm or enforcer can deliver the logical expression with physical properties that satisfy the 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, 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 optimizing the input to the sort, 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 additions 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 like sort order can only be determined after an execution plan has been chosen. As a simple consistency check, 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 addition and comparison, (6) an ADT "logical properties," (7) an ADT "physical property vector," (8) comparisons for physical properties (equality and cover), (9) an applicability function for each algorithm and enforcer, (10) a cost function for each algorithm and enforcer, (11) a property function for each logical operator, and (12) a property function for each 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. Moreover, 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 still be significantly less than designing and implementing a new optimizer from scratch.

The resulting optimizer is extensible in several important dimensions identified as necessary for query optimization in object-oriented systems, namely (1) the set of logical operators, (2) the set of equivalence rules, (3) the set of query execution algorithms, (4) the mapping from data model operations to algorithms, (5) the cost measure and cost estimation, and (6) the selectivity estimation model and statistics. Thus, we hope that the Volcano optimizer generator will prove to be a useful tool for query processing research in next-generation, extensible and object-oriented database systems.

## 3.3. Example Generator Input

Figure 2 shows example generator input. Operators, algorithms, and enforcers are declared with their arity, i.e., the number of algebra expressions required as their input. Transformation rules are defined by a pair of logical algebra expressions and a transfer symbol. Operators can either be explicitly named or can be variables, as in the example. This permits specification of generic rules, e.g., commutativity and associativity, a feature that was missing in the EXODUS optimizer generator. The arguments, e.g., join predicates, appear in the expressions to permit correct transfer of arguments during optimization, e.g., of the two predicates appearing in a join associativity rule. Condition code can refer to the macro REJECT, which is supplied automatically by the optimizer generator. Application code is invoked after a rule has been applied, in this case to reverse the roles of left and right join attributes. Implementation rules specify mappings of operators to algorithms, and can be more complex than shown in Figure 2, e.g., to map the combination of two operators to one algorithm.

```
-- Define logical operators and their input arities.
%operator JOIN 2
%operator INTERSECTION 2

-- Define algorithms and their input arities.
%algorithm HASH_JOIN 2
%algorithm FILE_SCAN 0

-- Define enforcers and their input arities.
%enforcer SORT 1

-- Transformation rule: Commutativity rule for join and intersection.
-- '->!' indicates the rule should not be applied to expressions
-- which were generated by this rule.
%trans_rule (?op1 ?op_arg1 (?1 ?2)) ->! (?op1 ?op_arg2 (?2 ?1))
%cond_code
{{
  /* Check for correct operators; difference is not commutative. */
  if (?op1 != JOIN  &&  ?op1 != INTERSECTION)
    REJECT;
}}
%appl_code
{{
  /* Reverse the argument if the rule is applied. */
  if (?op1 == JOIN)
    reverse_argument (?op_arg2, ?op_arg1) ;
}}

-- Implementation rule: JOIN can be implemented by HASH_JOIN.
%impl_rule (JOIN ?op_arg1 (?1 ?2)) -> (HASH_JOIN ?al_arg1 (?1 ?2))
```

Figure 2. Example Generator Input.

## 4. The Search Engine

We consider query optimization to be a planning problem similar to many other planning problems in artificial intelligence (AI). However, there is no standard formulation or solution to the query optimization problem in AI because query optimization and AI researchers have traditionally not interacted very much; if at all, mostly to consider database techniques for knowledge bases rather than AI techniques for database problems. A second reason is that there is no standard formulation of "the" query optimization problem among database researchers because data models and query languages differ as well as query execution engines, their facilities and costs. Furthermore, the separation of logical algebra (like relational algebra with operators like join) and executable algebra (with operators like nested loops, hash join, merge join, and sort) makes this planning problem unique.

### 4.1. The Search Algorithm

Since our experience with the EXODUS optimizer generator indicated that it is easy to waste a lot of search effort, we designed the search algorithm for the Volcano optimizer generator to be very goal-oriented and driven by needs rather than by possibilities. In some sense, while the EXODUS search engine could be described as using forward chaining (in the sense this term is used in AI), the Volcano search algorithm uses backward chaining since it explores only those subqueries and plans that truly participate in a larger expression.

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 expression 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 3 shows a simplified diagram of an equivalence class; a network of equivalence classes is created during optimization and contains all optimization results.

Figure 4 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 ORDER BY clause of SQL), and a cost limit, typically infinity for the original user query until a first plan is found.

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., hash join for unsorted output and merge join for join output

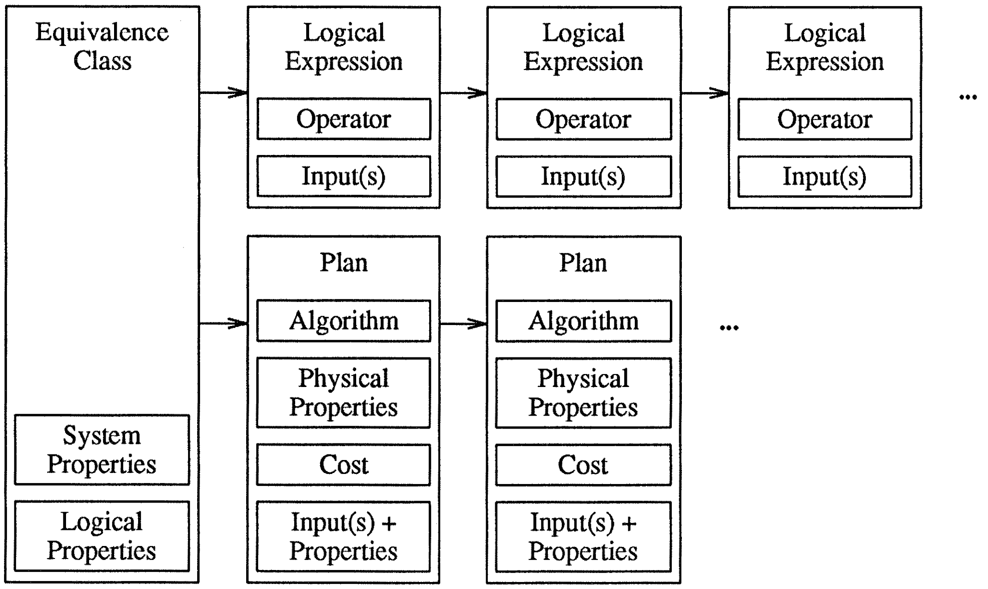

Figure 3. Data Structure of an Equivalence Class.

```
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 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 all inputs I while TotalCost < Limit
                determine required phys. prop. for I
                find cost by calling FindBestPlan
                add cost to TotalCost
        else /* move uses an 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 4. Outline of the Search Algorithm.

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.

Both matching and duplicate detection can be time-consuming activities, although the experience with the EXODUS optimizer generator indicated that expression matching can be done very efficiently after compilation and resolution of operator names into integers. Nonetheless, since we want to be able to build complex optimizers with very large rule sets, we decided to use a hashing scheme (hashed on the top-most operator in an expression) for fast rule matching and duplicate detection.

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 can 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 [13]. The difference between Starburst's two-level and Volcano's approach is that this separation is mandatory in Starburst while Volcano leaves 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 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.

Sometimes a new equivalence class is created during a transformation. Consider the associativity rule in Figure 5. 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.

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. The applicability function for the algorithm determines the physical property vectors for the algorithms inputs, and their costs are determined by invoking FindBestPlan for the inputs. 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.

For some binary operators, the actual physical properties of the inputs are not as important as the consistency of physical properties of all inputs. For example, for a sort-based implementation of intersection, i.e., an algorithm

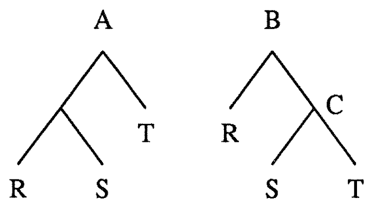

Figure 5. Associativity Rule.

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 relations 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 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. In many respects, enforcers are dealt with exactly like algorithms, which is not surprising considering that both are operators of the physical algebra. 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 5 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 a 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.

## 4.2. Current Work

The current operational version of the Volcano optimizer generator uses exhaustive search, i.e., search restricted only by the rule set and the condition code. Note, however, that this already permits implementing the restrictions found in the Starburst join enumerator/optimizer, e.g., limiting the number of relations that may comprise a "composite inner" [21]. We are currently coding the mechanisms required to exploit promise in the search as described above.

Using promise can limit the search effort in two ways. It prunes the search space indirectly and directly. First, it can be used to direct the search and to establish upper bounds quickly. Thus, it limits the search effort indirectly via branch-and-bound. Notice that all branch-and-bound pruning is safe, i.e., the pruned alternatives cannot be part of the optimal plan. Second, promise can also be used to avoid exploring alternatives which are unlikely to participate in the optimal plan. As the alternatives with highest promise were executed first, we explore alternatives with low promise last or not at all. We expect that such pruning can result in significant savings in optimization effort, but only so at the risk of missing an optimal plan. If promise is used for such direct pruning, the promise functions must be very carefully designed. Furthermore, the cutoff value for promise below which alternatives are pruned (ignored) without any exploration must be carefully tuned, probably differently for ad-hoc and repetitive queries.

Another way to exploit knowledge of promise is to divide the query optimization effort into multiple phases. One algorithm based on left-deep vs. bushy trees was explored experimentally in [6]; another strategy called the "pilot pass approach" based on parameter relaxation was conceptually designed by Rosenthal et al. [23]. We plan on experimenting with both ideas. After a quick optimization of an entire query expression without established upper cost bound but with very tight pruning parameters and a limited rule set, another optimization step might explore the entire query again with more tolerant pruning parameters but a firm upper bound. In fact, it might be possible to save overall optimization effort by gradually relaxing pruning parameters but tightening upper bounds in multiple phases.

Since query optimization can be a very time-consuming task, as shown many times for "simple" relational join optimization [19, 22], 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 [5], 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 we hope that our results will benefit relational, extensible, and object-oriented systems alike.

## 5. Comparison with the EXODUS Optimizer Generator

The search algorithm used in the Volcano optimizer generator differs significantly from the one in the EXODUS optimizer generator in a number of important aspects. We first summarize their differences in functionality and then present a preliminary performance comparison.

## 5.1. 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 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, 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 even more MESH nodes.

Third, 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 them, together with the logical expression, drive the optimization process as is done in Volcano was entirely absent in EXODUS.

The concept of physical property is very powerful and extensible. The most obvious 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; interestingly, this physical property would have two enforcers, sort- and hash-based. Location and partitioning in parallel and distributed systems can be enforced with a network and parallelism operator like Volcano's *exchange* operator [9, 10]. For query optimization in object-oriented system, we plan on defining "assembledness" of complex objects in memory as a physical property and using the assembly operator described in [16] as the enforcer for this property. Finally, considering the inherent error in selectivity estimation for complex queries [14], plan "robustness" might be a useful physical property that can be enforced by the *choose-plan* operator introduced in [8], which permits including several equivalent sub-plans in a single plan and delaying an optimization decision until query execution time.

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 similar to the cost model in System R [24], or even 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, which allows optimizing queries and plans for any run-time situation with respect to memory contention. Other variables that might change between optimization and execution include the number of available processors and program variables still unbound at compilation and optimization time. Of course, functions are not totally ordered, and the cost comparison function provided by the optimizer implementor might not be able determine which of two functions is "less" than the other. In some situations, it will be possible, e.g., if one plan dominates another one independently of memory availability. If neither function consistently dominates the other, we propose using a *choose-plan* operator.

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 Figure 5. 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.

## 5.2. Search Efficiency and Effectiveness

In this section, we experimentally compare the efficiency of the mechanisms built into the EXODUS and Volcano search engines. For the experiments, we specified the data model descriptions as similarly as possible. In particular, we specified the same operators (scan, 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).

As a first comparison between the two search engines, we performed exhaustive optimizations of relational select-join queries. Figure 6 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 and refer to the scale on the left. Dashed lines indicate estimated plan execution times and refer to the scales on the right. Measurements from the EXODUS optimizer generator are marked with □'s, Volcano measurements are marked with O's. Note that both scales for the y-axis are logarithmic.

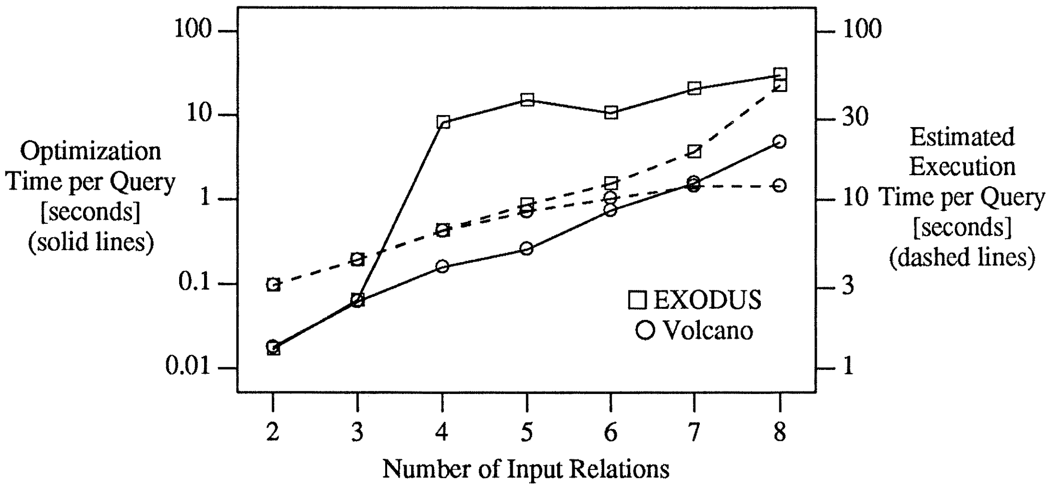

Figure 6. Exhaustive Optimization Performance.

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 [4]. The measurements in Figure 6 represent only those queries for which the EXODUS optimizer generator completed the optimization. The Volcano optimizer generator performed exhaustive search for all queries without problem.

For the queries represented in Figure 6, the plan quality as shown by the estimated execution cost is equal for moderately complex queries (up to 4 input relations). However, for more complex queries, it is significantly higher for EXODUS-optimized plans because the EXODUS-generated optimizer and its search engine ran out of memory (i.e., MESH had grown to 20 MB) and returned the best plan known at that time. The Volcano-generated optimizer used less than 1 MB for all queries.

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 at this point, reanalyzing becomes a substantial part of the query optimization effort in EXODUS. 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 possible, equivalent logical algebra expressions [22]. For more complex queries, the EXODUS' and Volcano's optimization times differ by about an order of magnitude.

## 6. Summary and Conclusions

Novel database applications demand not only high functionality but also high performance. To combine these two requirements, the Volcano project provides efficient, extensible tools for query and request processing in novel application domains, particularly in 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 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.

One of the techniques used to achieve this goal is a new optimizer generator, designed and implemented to further explore extensibility, search, heuristics, and learning in rule-based query optimization, and for research into extensible query processing in the Volcano and REVELATION projects, scientific databases, and dynamic query evaluation plans. Its design goals include extensibility, effectiveness, 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. From a preliminary performance comparison with the EXODUS optimizer generator, we concluded that optimizers built with the Volcano optimizer generator will be far more efficient than 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.

## Acknowledgements

Jim Martin has made many valuable contributions to this research, and he, Roger King, Rick Cole, and Richard Wolniewicz suggested improvements to earlier drafts of this paper. — This research has been partially supported by NSF with awards IRI-8805200, IRI-8912618, and IRI-9116547, DARPA with contract DAAB07-91-C-Q518, Texas Instruments, Intel Supercomputer Systems Division, Sequent Computer Systems, ADP, and the Oregon Advanced Computing Institute (OACIS).

## References

1. D. S. Batory, T. Y. Leung and T. E. Wise, ''Implementation Concepts for an Extensible Data Model and Data Language'', *ACM Transaction on Database Systems 13*, 3 (September 1988), 231.
2. S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt and B. Vance, ''Query Optimization in Revelation, an Overview'', *IEEE Database Eng. 14*, 2 (June 1991).
3. J. C. Freytag, ''A Rule-Based View of Query Optimization'', *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 173.
4. G. Graefe and D. J. DeWitt, ''The EXODUS Optimizer Generator'', *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 160.
5. G. Graefe, ''Software Modularization with the EXODUS Optimizer Generator'', *IEEE Database Eng. 10*, 4 (December 1987).
6. G. Graefe, ''Rule-Based Query Optimization in Extensible Database Systems'', *Ph.D. Thesis, University of Wisconsin–Madison*, August 1987.
7. 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.
8. G. Graefe and K. Ward, ''Dynamic Query Evaluation Plans'', *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 358.
9. G. Graefe, ''Encapsulation of Parallelism in the Volcano Query Processing System'', *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 102.
10. G. Graefe and D. L. Davison, ''Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Processing'', *submitted for publication*, 1991. Also CU Boulder Computer Science Technical Report 559.
11. G. Graefe, ''Volcano, An Extensible and Parallel Dataflow Query Processing System'', *to appear in IEEE Trans. on Knowledge and Data Eng.*, 1992. A more detailed version is available as CU Boulder Computer Science Technical Report 481, July 1990.
12. L. Haas, J. Freytag, G. Lohman and H. Pirahesh, ''Extensible Query Processing in Starburst'', *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 377.
13. 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 Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 143.
14. Y. E. Ioannidis and S. Christodoulakis, ''On the Propagation of Errors in the Size of Join Results'', *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 268.
15. S. C. Johnson, ''YACC: Yet Another Compiler Compiler'', *Computing Science Technical Report 32* (1975), Bell Laboratories.
16. T. Keller, G. Graefe and D. Maier, ''Efficient Assembly of Complex Objects'', *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 148.
17. 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, 1990, 290.
18. A. Kemper and G. Moerkotte, ''Access Support in Object Bases'', *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 364.
19. R. P. Kooi, ''The Optimization of Queries in Relational Databases'', *Ph.D. Thesis, Case Western Reserve University*, September 1980.

20. G. M. Lohman, ''Grammar-Like Functional Rules for Representing Query Optimization Alternatives'', *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 18.
21. K. Ono and G. M. Lohman, ''Extensible Enumeration of Feasible Joins for Relational Query Optimization'', *IBM Research Report RJ 6625 (63936)* (December 1988).
22. 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, 1990, 314.
23. A. Rosenthal, U. Dayal and D. Reiner, ''Fast Query Optimization Over a Large Strategy Space: The Pilot Pass Approach'', *unpublished manuscript*, 1986.
24. 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 Systems, Morgan-Kaufman, San Mateo, CA, 1988.
25. G. M. Shaw and S. B. Zdonik, ''A Query Algebra for Object-Oriented Databases'', *Proc. IEEE Conf. on Data Eng.*, Los Angelos, CA, February 1990, 154.
26. A. Silberschatz, M. Stonebraker and J. Ullman, ''Database Systems: Achievements and Opportunities'', *Communications of the ACM, Special Section on Next-Generation Database Systems 34*, 10 (October 1991), 110.
27. M. Stonebraker, ''Introduction to the Special Issue on Database Prototype Systems'', *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 1.
28. D. D. Straube, ''Queries and Query Processing in Object-Oriented Database Systems'', *University of Alberta, Department of Computer Science Technical Report 90-33*, Edmonton, Alberta, Canada, December 1990.
29. 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.
30. 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.
31. R. Wolniewicz and G. Graefe, ''Automatic Optimization and Parallelization in Scientific Databases'', *in preparation*, 1992.