The Cascades Framework for Query Optimization

Goetz Graefe

Abstract

This paper describes a new extensible query optimization framework that resolves many of the shortcomings of the EXODUS and Volcano optimizer generators. In addition to extensibility, dynamic programming, and memorization based on and extended from the EXODUS and Volcano prototypes, this new optimizer provides (i) manipulation of operator arguments using rules or functions, (ii) operators that are both logical and physical for predicates etc., (iii) schema-specific rules for materialized views, (iv) rules to insert "enforcers" or "glue operators," (v) rule-specific guidance, permitting grouping of rules, (vi) basic facilities that will later permit parallel search, partially ordered cost measures, and dynamic plans, (vii) extensive tracing support, and (viii) a clean interface and implementation making full use of the abstraction mechanisms of C++. We describe and justify our design choices for each of these issues. The optimizer system described here is operational and will serve as the foundation for new query optimizers in Tandem's NonStop SQL product and in Microsoft's SQL Server product.

1 Introduction

Following our experiences with the EXODUS Optimizer Generator [GrD87], we built a new optimizer generator as part of the Volcano project [GrM93]. The main contributions of the EXODUS work were the optimizer generator architecture based on code generation from declarative rules, logical and physical algebra's, the division of a query optimizer into modular components, and interface definitions for support functions to be provided by the database implementor (DBI), whereas the Volcano work combined improved extensibility with an efficient search engine based on dynamic programming and memorization. By using the Volcano Optimizer Generator in two applications, a object-oriented database systems [BMG93] and a scientific database system prototype [WoG93], we identified a number of flaws in its design. Overcoming these flaws is the goal of a completely new extensible optimizer developed in the Cascades project, a new project applying many of the lessons learned from the Volcano project on extensible query optimization, parallel query execution, and physical database design. Compared to the Volcano design and implementation, the new Cascades optimizer has the following advantages. In their entirety, they represent a substantial improvement over our own earlier work as well as other related work in functionality, ease-of-use, and robustness.

- Abstract interface classes defining the DBI-optimizer interface and permitting DBI-defined subclass hierarchies
- Rules as objects
- Facilities for schema- and even query-specific rules
- Simple rules requiring minimal DBI support
- Rules with substitutes consisting of a complex expression

- Rules that map an input pattern to a DBI-supplied function
- Rules to place property enforcers such as sort operations
- Operators that may be both logical and physical, e.g., predicates
- Patterns that match an entire subtree, e.g., a predicate
- Optimization tasks as data structures
- Incremental enumeration of equivalent logical expressions
- Guided or exhaustive search
- Ordering of moves by promise
- Rule-specific guidance
- Incremental improvement of estimated logical properties

The points in the list above and their effects will be discussed in this paper. While the system is operational, we have not performed any performance studies and the system is not fully tuned yet. Detailed analysis and focused improvement of the Cascades optimizer's efficiency is left for further work.

2 Optimization Algorithm and Tasks

The optimization algorithm is broken into several parts, which we call ”tasks.” While each task could easily be implemented as a procedure, we chose to realize tasks as objects that, among other methods, have a ”perform” method defined for them. Task objects offer significantly more flexibility than procedure invocations, in particular with respect to search algorithm and search control. A task object exists for each task that has yet to be done; all such task objects are collected in a task structure. The task structure is currently realized as a last-in-first-out stack; however, other structures can easily be envisioned. In particular, task objects can be reordered very easily at any point, enabling very flexible mechanisms for heuristic guidance. Moreover, we plan on representing the task structure by a graph that captures dependencies or the topological ordering among tasks and permit efficient parallel search (using shared memory). However, in order to obtain a working system fast, the current implementation is restricted to a LIFO stack, and scheduling a task is very similar to invoking a function, with the exception that any work to be done after a sub-task completes must be scheduled as a separate task.

Figure 1 shows the tasks that make up the optimizer's search algorithm. Arrows indicate which type of task schedules (invokes) which other type; dashed arrows indicate where invocations pertain to inputs, i.e., subqueries or subplans. Brief pseudo-code for the tasks is also given in the appendix. The ”optimize()” procedure first copies the original query into the internal ”memo” structure and then triggers the entire optimization process with a task to optimize the class corresponding to the root node of the original query tree, which in turn triggers optimization of smaller and smaller subtrees.

A task to optimize a group or an expression represents what was called an ”optimization goal” in the Volcano optimizer generator: it combines a group or expression with a cost limit and with required and excluded physical properties. Performing such a task results either in a plan or a failure. Optimizing a group means finding the best plan for any expression in the group and therefore applies rules to all expressions, whereas optimizing an expression starts with a single expression. The former is realized by invoking the latter for each expression. The latter results in transitive rule applications and therefore, if the rule set is complete, finds the best plan within the starting expression's group. The distinction between the two task types is made purely for pragmatic reasons. On the one hand, there must be a task to find the best plan for any expression in a group in order to initiate optimization

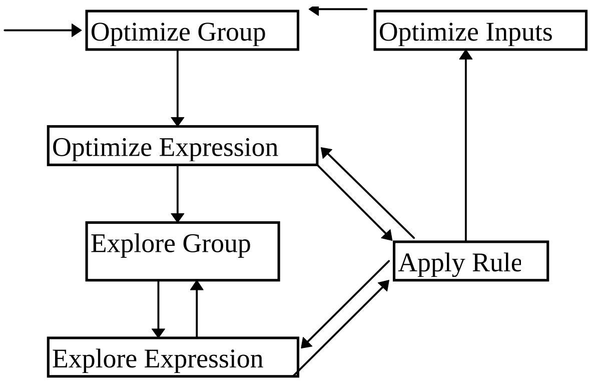

Figure 1. Optimization Tasks

of an entire query tree or a subtree after an implementation rule has been applied; on the other hand, there must be a task to optimize a single (new) expression after applying a transformation rule.

The task to optimize a group also implements dynamic programming and memorization. Before initiating optimization of all a group's expressions, it checks whether the same optimization goal has been pursued already; if so, it simply returns the plan found in the earlier search. Reusing plans derived earlier is the crucial aspect of dynamic programming and memorization. Exploring a group or an expression is an entirely new concept that has no equivalent in the Volcano optimizer generator. In the Volcano search strategy, a first phase applied all transformation rules to create all possible logical expressions for a query and all its subtrees. The second phase, which performed the actual optimization, navigated within that network of equivalence classes and expressions, applied implementation rules to obtain plans, and determined the best plan.

In the Cascades optimizer, this separation into two phases is abolished, because it is not useful to derive all logically equivalent forms of all expressions, e.g., of a predicate. A group is explored using transformation rules only on demand, and it is explored only to create all members of the group that match a given pattern. Thus, exploring a group or an expression (the distinction between these two mirrors the distinction between optimizing a group or an expression) means deriving all logical expressions that match a given pattern. The pattern, which is part of the task definition, is a subtree of the rule's antecedent or ”before”-pattern.

As do optimization tasks, exploration tasks also avoid duplicate work. Before exploring a group's expressions, the task to explore a group checks whether the same pattern has already been explored for the given group. If so, the task terminates immediately without spawning other tasks. Thus, the overall effort to expand logical expressions is also reduced by dynamic programming, i.e., retaining and reusing results of earlier search effort. The decision whether or not a pattern has already been explored is made using a ”pattern memory” initialized and administered by the DBI.

In order to make this discussion more concrete, consider a join associativity rule. In Volcano, all equivalence classes are completely expanded to contain all equivalent logical expressions before the actual optimization phase begins. Thus, during the optimization phase, when a join operator matches the top join operator in the rule, all join expressions for the rule's lower join are readily available so the rule can immediately applied with all possible bindings. In Cascades, these expressions are not immediately available and must be derived before the rule is applied. The exploration tasks provide this functionality; they are invoked not during a pre-optimization phase as in Volcano but on demand for a specific group and a specific pattern.

One might ask which of the Volcano technique and the Cascades technique is more efficient and more effective. The Volcano technique generates all equivalent logical expressions exhaustively in the first phase. Even if the actual optimization phase uses a greedy search algorithm, this first phase in Volcano must still be exhaustive. In the Cascades technique, this represents the worst case. If there is no guidance indicating which rule might lead to expressions matching the given pattern, exhaustive enumeration of all equivalent logical expressions cannot be avoided. On the other hand, if there is some guidance, some of that effort can be avoided, and the Cascades search strategy seems superior. On the other hand, the same group might have to be explored multiple times for different patterns ; if so, redundant rule applications and derivations might occur. In order to avoid that, each expression in the ”memo” structure includes a bit map that indicates which transformation rules have already been applied to it and thus should not be re-applied. Thus, we believe that the Cascades search strategy is more efficient because it explores groups only for truly useful patterns. In the worst case, i.e., without any guidance, the efficiency of the Cascades search will equal that of the Volcano search strategy.

On the other hand, if such guidance is incorrect, incorrect pruning of the search space may occur and the Cascades optimizer’s effectiveness might suffer. Thus, it is very important that such guidance be correct. We plan on using two techniques for guidance, which are not implemented yet. First, by inspecting the entire rule set, in particular the top operators of each rule’s antecedent (”before”-pattern) and consequent (”after”-pattern, substitute), we can identify which operators can be mapped to which other operators in a single rule application. By taking the transitive closure of this reachability relationship, we can exclude some rules from consideration. Note that this transitive closure can be computed when the optimizer is generated from the rule set, i.e., only once. Second, we plan on implementing mechanisms for guidance by the DBI.

Applying a rule creates a new expression; notice that the new expression can be complex (consisting of multiple operators, as in a join associativity rule) and may be either a transformation rule (creating a new logical expression) or an implementation rule (creating a new physical expression or plan). In fact, since an operator can be both logical and physical, one rule may be both a transformation and an implementation rule. Correct rule application for such rules is guaranteed, although we expect such operators and rules to be exceptions rather than the norm.

Performing an ”apply rule” task is fairly complex. It may roughly be broken into four components. First, all bindings for the rule’s pattern are derived and iterated over one by one. Second, for each binding, the rule is used to create a new expression. Note that for function rules, there may be multiple new expressions for each binding. Third, the new expressions are integrated in the ”memo” structure. Within this process, exact replicas of expressions that already exist in ”memo” are identified and removed from further consideration. Fourth, each expression that is not a duplicate of an earlier one is optimized or explored with the same goal and context that triggered the current rule application. Let us discuss these four components in turn.

Since each rule’s antecedent (”before”-pattern) may be complex, the Cascades optimizer employs a complex procedure to identify all possible bindings for a rule. This procedure is recursive, with each recursive invocation for each node in the pattern. Most of its complexity serves to obtain all possible bindings for a rule’s pattern. In fact, the procedure is realized as an iterator that produces the next feasible binding with each invocation. The state of this iteration is captured in the ”BINDING” class with one instance of that class for each node in the pattern. Once a binding is found, it is translated into a tree consisting of ”EXPR” nodes (note that this class is part of the DBI interface, whereas the optimizer’s internal data structures are not). This copy step represents some effort, but it isolates the optimizer from the DBI methods that may be invoked for this tree. For each binding, the rule’s condition function is invoked and qualifying bindings are then translated into the rule’s consequent (”after”-pattern, substitute). For some rules, this is very easy and entirely left to the optimizer. For other rules, the DBI specified a function to create the substitute, and this function is invoked repeatedly to create as many substitute as possible. In other words, this function may be an iterator producing multiple substitutes in consecutive invocations. Thus, the effort of extracting a binding from the ”memo” is leveraged for multiple transformations if possible.

Each substitute expression is then integrated into the ”memo” structure. This process includes search for and detection of duplicates, i.e., expression that have been derived earlier in the optimization. This process is very similar to duplicate expression detection in both the EXODUS and Volcano optimizer generators. It is a recursive process that starts at the leaves of the substitute, which may be either query or plan tree leaves (i.e., scans) or leaf operators that denote the scope of a rewrite operation (as described as part of the DBI interface), and works upwards in the substitute towards the substitute’s root; this direction is required for correct duplicate detection. The search for duplicates is very fast as it employs a hash table using an operator and the groups of its inputs as keys.

Finally, if a substitute's root is a new expression, follow-on tasks may be initiated. If the substitute was created as part of an exploration, a task is created to explore the substitute for the same pattern. If the substitute was created as part of an optimization, the follow-on tasks depend on whether the rule was a transformation or an implementation rule, i.e., whether the substitute's root operator is a logical or a physical operator. Note, again, that an operator can be both logical and physical; thus, a rule can be both a transformation or an implementation rule. In that case, both types of follow-on tasks are created. For a logical root operator, an optimization task is created to optimize the substitute, keeping the same optimization goal. For a physical root operator, a new task is scheduled to optimize the operator's inputs and to calculated processing costs. The ”optimize inputs” task is different from all other tasks. While all other tasks schedule their follow-on tasks and then vanish, this sixth task type become active multiple times. In other words, it schedules a follow-on task, waits for its completion, resumes and schedules the next follow-on task, etc. The follow-on tasks are all of the same type, which is optimizing input groups for a suitable optimization goal. Thus, like the Volcano search strategy, the Cascades search engine guarantees that only those subtrees and interesting properties are optimized that could indeed participate in a query evaluation plan. Each time after an input has been optimized, the optimize inputs task obtains the best execution cost derived, and derives a new cost limit for optimizing the next input. Thus, pruning is as tight as possible.

3 Data Abstraction and the User Interface

Developing the Cascades optimizer system required pursuing three different activities in rapid alternation. First, designing the interface between database implementor and optimizer had to focus on minimal, functional, and clean abstractions. Second, implementing a prototype optimizer as our own DBI was an exercise in exploiting the interface as effectively as possible. Third, design and implementation of an efficient search strategy was based on lessons learned during the EXODUS and Volcano projects, combined with the requirements set forth by a workshop of academic and industrial query optimization researchers and by the first user group of this software. Each of these three activities had different goals and required a different mind-set; in our internal discussions, we constantly alternated among these perspectives in order to design and develop a truly extensible and useful tool. In this section, we describe the data structuring decisions made for the interface between database implementor and the optimizer.

Users of the EXODUS and Volcano optimizer generator generators made it very clear that the interface of these system could bear improvement. Feedback from users of the Volcano optimizer generator matches our own analysis [BMG93]; therefore, we focused on (i) clean abstractions for support functions in order to enable an optimizer generator to create them from specification, (ii) rule mechanisms that permit the DBI to choose rules or functions to manipulate operator arguments (such as predicates), and (iii) more concise and complete interface specifications, both in the code and in the written documentation. Following these guidelines, we designed the following interface.

Each of the classes that make up the interface between the Cascades optimizer and the DBI is designed to become the root of a subclass hierarchy. Thus, creation of new objects of one of these classes is associated with another class. For example, creation a new ”guidance” structure is associated with a ”rule” object. The rule object can be of some DBI-defined subclass of the interface class ”RULE,” and the newly created guidance structure can be of any DBI-defined subclass of the interface class ”GUIDANCE.” The optimizer relies only on the method defined in this interface; the DBI is free to add additional methods when defining subclasses.

3.1 Operators and Their Arguments

Central to any database query optimizer are the sets of operators supported in the query language and in the query evaluation engine. Notice that these two sets are different; we call them logical and physical operators [Gra93]. While previous extensible operators required that these two sets be disjunct, we have abandoned this requirement. The ”class OP-ARG” in the Cascades optimizer interface includes both logical and physical operators. For each operator, one method called ”is-logical” indicates whether or not an operator is a logical operator, while a second method called ”is-physical” indicates whether or not an operator is a physical operator. In fact, it is possible that an operator is neither logical or physical; such an operator might be useful if the optimization is organized as an expansion grammar including ”non-terminals” like the Starburst optimizer [Loh88]. On the other hand, a DBI who wishes to do so can easily retain a strict separation of logical and physical operators, e.g., by defining subclasses with suitable definitions for the methods ”is-logical” and ”is-physical” and by defining all operators as subclasses of these two classes.

The definition of operators includes their arguments. Thus, no separate mechanisms are required or provided for ”argument transfer” as in EXODUS and Volcano. Notice, however, that there are two crucial facilities that permits and encourage modeling predicates etc., which had been modeled as operator arguments in all our prototypes constructed in the EXODUS and Volcano frameworks, as primary operators in the logical and physical algebra’s. First, an operator can be both logical and physical, which is natural for single-record predicates, called ”sargable” in System R [SAC79]. Second, specific predicate transformations, e.g., splitting from a complex predicate those components that can be pushed through a join, which are most easily and efficiently implemented in a DBI function rather than as rules to be interpreted by the optimizer’s search engine, can easily be realized in rules that invoke a DBI-supplied to map an expression to substitute expressions (one or more). Thus, after the EXODUS and the Volcano work has been repeatedly criticized that predicate manipulation has been very cumbersome, the Cascades optimizer offers much improved facilities.

The optimizer’s design does not include assumptions about the logical and physical algebra’s to be optimized; therefore, no query or plan operators are built into the optimizer. For use in rules, however, there are two special operators, called ”LEAF-OP” and ”TREE-OP.” The leaf operator can be used as leaf in any rule; during matching, it matches any subtree. Before a rule is applied, an expression is extracted from the search memory that matches the rule’s pattern; where the rule’s pattern has leaves, the extracted expression also has leaf operators that refer (via an array index) to equivalence classes in the search memory. The tree operator is like the leaf operator except that the extracted expression contains an entire expression, independent of its size or complexity, down to the leaf operators in the logical algebra. This operator is particularly useful in connection with function rules, which are described below.

Beyond the methods ”is-logical” and ”is-physical,” all operators must provide a method ”opt- cutoff”. Given a set of moves during an optimization task, this method determines how many of those will be pursued, obviously the most promising ones. By default, all possible moves will be pursued, because exhaustive search guarantees that the optimal plan will be found. There is also a small set of methods that must be provided only for those operators that have been declared logical. For pattern matching and for finding duplicate expressions, methods for matching and hashing are required. Methods for finding and improving logical properties are used to determine an original set of properties (e.g., the schema) and then to improve it when alternative expressions have been found (e.g., more bounds on selectivity or the output size). Finally, for exploration tasks, an operator may be called upon to initialize a pattern memory and to decide how many moves to pursue during an exploration task.

Similarly, there are some methods for physical operators. Obviously, there is a method to determine an operator’s (physical) output properties, i.e., properties of the representation. Moreover, there are three methods that compute and inspect costs. The first of these calculates the local cost of an algorithm, without any regard to the costs of its inputs. The second one combines the costs and physical properties of an algorithm’s inputs into the cost of an entire subplan. The third of these methods verifies, between optimizing two inputs of an algorithm, that the cost limit has not been exceeded yet, and computes a new cost limit to be used when optimizing the next input. Finally, just as the last method maps an expression's cost limit to a cost limit for one of its inputs, there is a method that maps the optimization goal for an expression to an optimization goal for one of its inputs, i.e., a cost limit and required and excluded physical properties, called ”input-reqd-prop.” Let us discuss properties and their methods next.

3.2 Logical and Physical Properties, Costs

The interface to the interface for anticipated execution costs, the ”class COST,” is very simple, since instances of costs are created and returned by methods associated with other classes, e.g., operators. Beyond destruction and printing, the only method for costs is a comparison method. Similarly, the only method for the encapsulation of logical properties, the ”class SYNTH-LOG- PROP,” is a hash function that permits faster retrieval of duplicate expressions Since even this function does not apply to physical expressions, the encapsulation for physical properties, the ”class SYNTH-PHYS-PROP,” has no methods at all. The class for required physical properties, the ”class REQD-PHYS-PROP,” has only one method associated with it, which determines whether a synthesized physical property instance covers the required physical properties. If one set of properties is more specific than another, e.g., one indicates a result sorted on attributes ”A, B, C” and the one requires sort order on ”A, B” only, the comparison method returns the value ”MORE.” The default implementation of this method returns the value ”UNDEFINED.”

3.3 Expression Trees

In order to communicate expressions between the DBI and the optimizer, e.g., as queries, as plans, or in rules, another abstract data type is part of the interface, call the ”class EXPR.” Each instance of this class is a node in a tree, consisting of an operator and to pointers to input nodes. Obviously, the number of children in any expression node must be equal to the arity function of the node's operator. Methods on an expression node, beyond constructor, destructor, and printing, include methods to extract the operator or one of the inputs as well as a matching method, which recursively traverses two expression trees and invokes the matching method for each node's operator.

3.4 Search Guidance

In addition to pattern, cost limits, and required and excluded physical properties, rule application can also controlled by heuristics represented by instances of the ”class GUIDANCE.” Its purpose is to transfer optimization heuristics from one rule application to the next. Notice that costs and properties pertain to the expressions being manipulated and to the intermediate result those expressions will product when a query plan is executed; the guidance class captures knowledge about the search process and heuristics for future search activities. For example, some rules such as commutativity rules are to applied only once; for those, a simple guidance structure and a rule class are provided as part of the DBI interface, called ”ONCE-GUIDANCE” and ”ONCE-RULE.”

Some researchers have advocated to divide a query optimizer's rule set into ”modules” that can be invoked one at a time, e.g., Mitchell et al. [MDZ93]. Guidance structures can easily facilitate this design: a guidance structure indicates which module is to be chosen, and each rule checks this indication in its promise (or condition) function and then creates suitable indications when creating guidance structures for its newly created expressions and their inputs.

3.5 Pattern Memory

In addition to the search guidance, exploration effort can be restricted by use of the pattern memory. The purpose of the pattern memory is to prevent that the same group is explored unnecessarily, e.g., twice for the same pattern. There is one instance of a pattern memory associated with each group. Before a group is explored for a pattern, the pattern memory is permitted to add the pattern to itself and is asked to determine whether or not exploration should take place. In the most simple search, in which exploration for any pattern is performed by exhaustive application of transformation rules, the pattern memory needs to contain only a Boolean, i.e., a memory whether or not the group has been explored previously. More sophisticated pattern memories would store each pattern.

Obviously, the pattern memory interacts with the exploration promise function. For the most simple promise function that always admits exhaustive search, the simple pattern memory above is suitable. It is left to the DBI to design pattern memory and promise functions most suitable to the algebra to be optimized.

Beyond checking whether a given pattern already exists in the memory, and saving it to detect a second exploration with the same pattern, the most complex method for pattern memories is to merge two pattern memories into one. This method is required when two groups of equivalent expressions are detected to be actually one, i.e., when a transformed expression already occurs in a different group in the search memory.

3.6 Rules

Next to operators, the other important class of objects in the Cascades optimizer are rules. Notice that rules are objects; thus, new ones can be created at run-time, they can be printed, etc. While other rule-based optimizers, in particular the EXODUS and Volcano optimizer generators, divide logical and physical operators as well as (logical) transformation and (physical) implementation rules into disjoint sets, the Cascades optimizer does not distinguish between those rules, other than by invoking the is-logical and is-physical methods on newly created expressions. All rules are instances of the ”class RULE,” which provides for rule name, an antecedent (the ”before” pattern), and a consequent (the substitute). Pattern and substitute are represented as expression trees, which were discussed above.

In their simplest case, rules do not contain more than that; whenever the pattern is found or can be created with exploration tasks, the substitute expression is included in the search memory. Both a rule’s pattern and substitute can be arbitrarily complex. In the EXODUS and Volcano optimizer generators, an implementation rule’s substitute could not consist of more than a single implementation operator; in the Cascades design, this restriction has been removed. The remaining restriction is that all but the substitute’s top operator must be logical operators. For example, it is possible to transform a (logical) join operator into a (physical) nested loops operator with a (logical) selection on its inner input, thus, detaching the selection predicate from the join algorithm and pushing it into the inner input tree.

For more sophisticated rules, two types of condition functions are supported. All of them consider not only the rule but also the current optimization goal, i.e., cost limit and required and excluded physical properties. First, before exploration starts, ”promise” functions informs the optimizer how useful the rule might be. There is one promise function for optimization tasks and one for exploration tasks. For unguided exhaustive search, all promise functions should return the value 1.0. A value of 0 or less will prevent the optimizer from further work for the current rule and expression. The default promise function returns 0 if a specific physical property is required, 2 if the substitute is an implementation algorithm, and 1 otherwise. If the cutoff methods associated with the operators choose exhaustive search (see above), the return value of the promise function will not change the quality of the final query evaluation plan, although it may affect the order in which plans are found, pruning effectiveness, and therefore the time an optimization takes.

Since the promise functions are invoked before exploration of subgroups, i.e., before entire expression trees corresponding to a rule’s pattern have been explored and extracted from the search memory, a ”condition” function checks whether a rule is truly applicable after exploration is complete and a complete set of operators corresponding to the pattern in the rule is available. Whereas the promise functions return a real value that expresses grades of promise, the condition function returns a Boolean to indicate whether or not the rule is applicable.

In addition to promise and condition functions, a small set of methods is associated with rules. Of course, there are constructor, destructor, and print methods, as well as method to extract the pattern, the substitute, the rule’s name, and its arity (the pattern’s number of leaf operators). The ”rule-type” method indicates whether a rule is a simple rule (as described so far) or a function rule (to be described shortly). The ”top-match” method determines whether or not an operator in the search memory matches the top operator in the rule’s pattern; this method is the only built-in check before an promise function is invoked. The method ”opt-cases” indicates how often a physical expression is to be optimized with different physical properties. In all but a few cases, this will be one; one of the few exception is a merge-join algorithm with two equality clauses (say ”R.A == S.A and R.B == S.B”) that should be optimized for two sort orders (sorted on ”A, B” and on ”B, A”). By default, this method returns 1. The remaining methods all create new guidance structures to be used when optimizing a newly created expression and its inputs. There are two methods each for optimization and for exploration, and two each for the new expression and for its inputs, called ”opt-guidance,” ”expl-guidance,” ”input-opt-guidance,” and ”input-expl-guidance.” By default, all of them return ”NULL,” i.e., no specific guidance.

If a rule’s substitute consists of only a leaf operator, the rule is a reduction rule. If a reduction rule is applicable, two groups in the search memory will be merged. On the other hand, if a rule’s pattern consists of only a leaf operator, the rule is an expansion rule that is always applicable. The Cascades optimizer must rely on the DBI to design appropriate promise and condition functions to avoid useless transformations. Nonetheless, there is an important class of situations in which expansion rules are useful, namely the insertion of physical operators that enforce or guarantee desired physical properties. Such rules may also be called enforcer rules. Consider the inputs to a merge-join’s inputs, which must be sorted. An enforcer rule may insert a sort operation, the rule’s promise and condition functions must permit this rule only of sort order is required, and the sort operator’s ”input-reqd-prop” method must set excluded properties to avoid consideration of plans that produce their output in the desired sort order as input to the sort operator.

In some situations, it is easier to write a function that directly transforms an expression than to design and control a rule set for the same transformation. For example, dividing a complex join predicate into clauses that apply to left, right, and both inputs is a deterministic process best implemented by a single function. For those cases, the Cascades optimizer supports a second class of rules, called the ”class FUNCTION-RULE.” Once an expression is extracted that corresponds to the rule’s pattern, an iterator method is invoked repeatedly to create all substitutes for the expression. Note that the extracted expression can be arbitrarily deep and complex if the tree operator (see above) is employed in the rule’s pattern. Thus, tree operators and function rules permit the DBI to write just about any transformation. In the extreme case, a set of function rules could perform all query transformations, although that would defeat some of the Cascades framework’s purpose.

4 Future Work

Of course, there is a lot of work that should be done to make the Cascades optimizer more useful and complete. First, the optimizer has not yet gone through a thorough evaluation and tuning phase. Second, building additional optimizers based on this framework will undoubtedly show many weaknesses not yet apparent. Third, one or more generators that produce Cascades specifications from higher-level data model and algebra descriptions would be very useful. Fourth, we already know of a number of desirable improvements for the search strategy and its implementation.

The Cascades optimizer was designed to be reasonably fast, although extensibility was a more important design goal. Among the artifacts of separating the optimizer framework and the DBI’s specification of operators, cost functions, etc. are extensive use of virtual methods, a very large number of references between structures, and very frequent object allocation and deallocation. While unavoidable, there probably is room for improvement, in particular if one is willing to give up the strong separation that permits modifications of DBI code without recompiling Cascades code. Before this ”de-modularization” step is taken, however, a strong argument should be made based on a measurement study that this would indeed improve an optimizer’s performance.

5 Summary and Conclusions

Beyond a better and more robust implementation than found in the EXODUS and Volcano optimizer generators, the Cascades optimizer offers a number of advantages, without giving up modularity, extensibility, dynamic programming, and memorization explored in those earlier prototypes. First, predicates and other item operations can conveniently modeled as part of the query and plan algebra. Operators that are both logical and physical; thus, it is easy to specify operators that may appear both in the optimizer input (the query) an din its output (the plan). Function rules and the tree operator permit direct manipulation of even complex trees of item operations using DBI-supplied functions. Second, enforcers such as sorting are normal operators in all ways; in particular, they are inserted into a plan based on explicit rules. In Volcano, they were special operators that did not appear in any rule. Third, both exploration (enumeration of equivalent logical expressions) and optimization (mapping a logical to a physical expression) can be guided and controlled by the DBI. Together with the more robust implementation as required for industrial deployment, we believe that the Cascades optimizer represents a substantial improvement over earlier extensible database query optimizers.

6 Acknowledgments

The query processing group at Tandem has been very helpful in forcing me to address the hard problems unresolved in the EXODUS and Volcano optimizer generators and in finding effective and usable solutions. David Maier has been a great sounding board for ideas during the design and development of the Cascades optimizer.

7 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.

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

[Gra93] G. Graefe, Query Evaluation Techniques for Large Databases, ACM Computing Surveys 25, 2 (June 1993), 73-170.

[GrM93] G. Graefe and W. J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, Proc. IEEE Int'l. Conf. on Data Eng., Vienna, Austria, April 1993, 209.

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

[MDZ93] G. Mitchell, U. Dayal, and S. B. Zdonik, Control of an Extensible Query Optimizer: A Planning-Based Approach, Proc. Int'l. Conf. on Very Large Data Bases, Dublin, Ireland, August 1993, 517.

[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.

[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.