Paper Number 323

OPT++ : An Object–Oriented Implementation for Extensible Database Query Optimization[*]

Navin Kabra David J. DeWitt

Computer Sciences Department
University of Wisconsin, Madison
{navin, dewitt}@cs.wisc.edu

Abstract

In this paper we describe the design and implementation of OPT++, a tool for Extensible Database Query Optimization that uses an object-oriented design to simplify the task of implementing, extending, and modifying an optimizer. Building an optimizer using OPT++ makes it easy to extend the query algebra (to add new query algebra operators and physical implementation algorithms to the system), easy to change the search space explored, and also easy to change the search strategy used. Furthermore, OPT++ comes equipped with a number of search strategies that are available for use by an Optimizer–Implementor. OPT++ considerably simplifies both, the task of implementing an optimizer for a new database system, and the task of experimenting with various optimization techniques and strategies to decide what techniques are best suited for that database system. We present the results of performance studies which validate our design and show that, in spite of its flexibility, OPT++ can be used to build efficient optimizers.

1 Introduction

Although constructing a high-performance database engine has become almost straightforward, building query optimizers remains a "black art". Writing an optimizer, debugging it, and evaluating different optimization strategies remains a difficult and time-consuming task. Consequently, the state of commercial optimizers is frequently not very good, in spite of the fact that query optimization has been a subject of research for more than 15 years. Furthermore, existing commercial optimizers are often so brittle from years of patching that further improvement ranges from difficult to impossible. While quite a bit has been published about extensible query optimizers in the research literature, the actual success of this work is limited. Thus, good tools are still needed to streamline the process of implementing and evolving query optimizers.

Extensible query optimization frameworks that have been proposed in the research literature have a number of drawbacks. Optimizers that allow easy addition of new query algebra operators/algorithms

[*]This research was supported by the Advanced Research Project Agency, ARPA order number 144-CS84, monitored by the Oregon Graduate Institute of Science and Technology and the Army Research Laboratory.

often have a fixed search strategy that cannot be changed. On the other hand, optimizers that offer extensibility of the search strategy are not very extensible with respect to the query algebra. Furthermore, there are often no studies of the efficiency of the resulting optimizers.

The remainder of this paper describes our attempt to develop an alternative framework for constructing query optimizers. First, it should be easy to add new operators as well as new execution algorithms for existing operators. Second, the framework should allow the Optimizer–Implementor to experiment with various heuristics that can limit the search space explored by the optimizer. The Optimizer–Implementor should be able to try different search strategies, and if necessary, to mix multiple strategies in a single optimizer. Finally, this flexibility should be achieved without compromising the efficiency of the optimizer – *i.e.*, an optimizer built in this extensible framework should not be significantly worse in its space or time requirements than an equivalent "custom-made" optimizer.

In order to address the issues of extensibility and maintainability, OPT++ exploits the object-oriented features of C++. It defines a few key abstract classes with virtual methods. These class definitions do not assume any knowledge about the query algebra or the database execution engine. The search strategy is implemented **entirely** in terms of these abstract classes. The search strategy invokes the virtual methods of these abstract classes to perform the search and the cost-based pruning of the search space.

An optimizer for a specific database system can be written by deriving new classes from these abstract classes. Information about the specific query algebra and execution engine for which the optimizer is built, and the search space of execution plans to be explored, are encoded in the virtual methods of these derived classes. The C++ inheritance mechanism ensures that the search strategy of the optimizer does not have to be changed when this is done.

Furthermore, the search strategy itself is a class with virtual methods that can be over-ridden. Thus, new classes can be derived from this class to implement different search strategies. OPT++ comes equipped with a number of search strategies that can be directly used by the Optimizer–Implementor. In addition, the Optimizer–Implementor can implement new search strategies by deriving new classes from the provided search strategy classes and rewriting the virtual methods.

An optimizer built using OPT++ consists of three components: the "Search Strategy" component determines what strategy is used to explore the search space (*e.g.*, dynamic-programming, randomized, *etc.*), the "Search Space" component determines what that search space is (*e.g.*, space of left-deep join trees, space of bushy join trees, *etc.*), and the "Algebra" component determines the actual logical and physical algebra for which the optimizer is written. OPT++ strives for separation of these components and, to a large extent, provides an architecture in which each of these components can be changed with minimum impact on the other components.

The remainder of this paper is organized as follows. Related work is presented in Section 2. Section 3 describes the design of our optimizer. Section 4 discusses our experiences to date using our optimizer framework, illustrating the ease of use as well as the efficiency of OPT++. In section 5 we present our conclusions.

2 Related Work

Extensible query optimizers proposed in the literature fall mainly into two categories: those that offer a fixed search strategy and allow the easy addition of new algorithms and operators, and those that allow the search strategy itself to be extensible. In OPT++ we have tried to achieve both these goals by coming up with a design in which the search strategy itself is extensible, and, for any search strategy implemented using this design, the addition of new algorithms and operators to the system is easy.

Most optimizers that allow extensibility of the query algebra propose some form of a rule-based system that uses rewrite rules to describe transformations that can be performed to optimize a query expression [Fre87, Gra87, PHH92, FG91]. These systems usually offer a more-or-less fixed search strategy that is difficult to modify or extend.

Freytag [Fre87] describes an architecture in which the translation of a query into an executable plan is completely based on rules. He describes a System-R style optimizer that can be built using various sets of rules. One set of rules is used to convert the query into an algebraic tree. Other sets of rules are used to generate access paths, join orderings, and join methods in that order.

The optimizer developed as a part of the Starburst project [LFL88, HP88] uses a two step process to optimize queries. The first phase uses a set of production rules to transform the query heuristically into an equivalent new query that (hopefully) offers both faster execution than the old query and is better suited for cost-based optimization. In the second phase, query processing alternatives are specified using grammar–like production rules. Each "non-terminal" in the grammar can have multiple production rules (suggesting execution alternatives) and conditions of applicability. These rules are used to construct an optimal execution plan in a bottom up fashion similar to the System-R optimizer. Cost estimates are used for choosing between alternatives.

This approach has several limitations. The rewrite phase (first phase) uses equivalence transformations to rewrite the query heuristically. While such heuristic transformations work in a number of cases, the heuristics sometimes make incorrect decisions because they are not based on cost estimates. The second phase (the cost-based optimizer) is built using grammar–like rules that are used to build bigger and bigger plans. While this approach is well suited for access method and join enumeration, it is not clear how this can be used to optimize queries containing non-relational operators and complicated transformations.

The optimizers generated by the Exodus Optimizer Generator [GD87] and the Volcano Optimizer Generator [GM93] use algebraic equivalence rules to transform an operator tree for a query into other, equivalent operator trees, and use implementation rules to determine what algorithms are used to implement the various operators. The algebraic transformation rules are used to generate all possible operator trees that are equivalent to the input query. The implementation rules are used to generate access plans corresponding to the operator trees.

Like the Volcano Optimizer Generator and the Starburst optimizer, OPT++ incorporates extensible specification of logical algebra operators, execution algorithms, logical and physical properties, and selectivity and cost estimation functions. Interesting physical properties, input constraints for execution algorithms and enforcers ("glue" operators) are also supported. OPT++ can be used to emulate both, the Starburst as well as the Exodus/Volcano based optimizers. The search strategies that are used in those optimizer generators are both built into OPT++. The constructs allowed in OPT++ can be used to implement the transformation rules and implementation rules of Volcano and the rewrite rules and production rules of Starburst. In addition, OPT++ has several advantages. First, those optimizers offer a more or less fixed search strategy while OPT++ offers a choice of different strategies. In fact, the search strategy in OPT++ is extensible and can be modified to fit the optimization problem, if necessary. Also, OPT++ offers the ability to mix the constructs from the two optimizers. Specifically, Volcano-like algebraic transformation rules can be mixed with System-R style building up of operator trees if necessary and feasible. Our experience with the implementation of an optimizer using OPT++ shows that this flexibility is achieved without sacrificing performance.

Various architectures have been proposed to allow extensible control over the search strategy of an optimizer. The region-based optimizer architecture of Mitchell et al. [MDZ93], the modular optimizer architecture by Sciore and Sieg [SJ90], and the blackboard architecture of Kemper, Moerkotte and Peithner [KMP93], are all based on the concept of dividing an optimizer into regions that carry out different parts of the optimization. A query then has to pass through these various regions to be optimized. They differ in the methods used to pass control between the various regions. In [SJ90] control passes from one region to another in a fixed sequence. In [MDZ93] there is a hierarchy of regions in which the parent region dynamically controls the sequence in which the query passes through the various regions while being optimized. In the blackboard approach [KMP93], knowledge sources are responsible for moving the queries between regions.

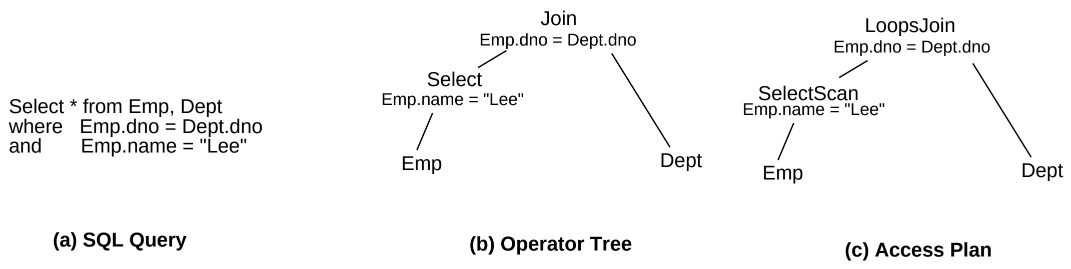
All these architectures describe very general frameworks for extensible query optimization that support multiple optimizer control strategies and allow addition of new control strategies. By making very specific assumptions about the kinds of manipulations that are allowed on the operator trees and access plans, OPT++ is able to put a lot of the functionality of an optimizer into the part of the code that does not depend upon the specific query algebra. This makes it much easier to write an optimizer from scratch in OPT++ than in any of the systems discussed above. In spite of these assumptions, a number of different search strategies can be implemented in OPT++ quite easily. Finally, while addition of algorithms and operators for any search strategy remains easy in OPT++, it is not clear how easy it is in these other systems.

Lanzelotte and Valduriez [LV91] also describe an object-oriented design for an extensible query optimizer. The design of the search strategy code in OPT++ is inspired by this work. However, OPT++ differs in its modeling of the query algebra and the search space. In particular, OPT++ has a clear separation between the logical algebra (operator trees) and the physical algebra (access plans). We believe this separation is necessary for efficiency of the optimizer as well as for clarity and extensibility. Although [LV91] discusses extensibility of the search strategy in detail, it is not clear how extensible their design is in terms of adding new operators and algorithms and modifying the search space explored, or how such changes would interact with one another or with the search strategy.

3 OPT++ System Design

3.1 Basic Concepts

We assume that a query can be logically represented as an *operator tree*. An *operator tree* is a tree in which each node represents a logical query algebra operator being applied to its inputs. For example,



```
Select * from Emp, Dept
where   Emp.dno = Dept.dno
and     Emp.name = "Lee"
```

Figure 1: Query Representations

Figure 1(a) shows an SQL query and Figure 1(b) shows that query represented as a tree of relational operators. A given query can be represented by one or more operator trees that are equivalent.

One or more physical execution algorithms can be used in a database for implementing a given query algebra operator. For instance, the join operator can be implemented using nested-loops or sort-merge algorithms. Replacing the operators in an operator tree by the algorithms used to implement them gives rise to a "tree of algorithms" known as an *access plan* or an execution plan [SAC+79]. Figure 1(c) shows one possible access plan corresponding to the operator tree in Figure 1(b). Each operator tree, in general, will have a number of corresponding access plans.

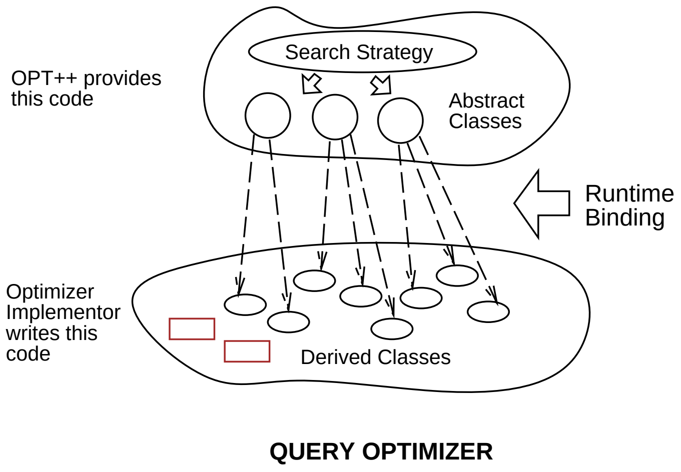


Figure 2: Basic System Design

During the course of query optimization, a query optimizer must generate various operator trees that represent the input query (or parts of it), generate various access plans corresponding to each operator tree, and compute/estimate various properties of the operator trees and access plans (for example, cardinality of the output relation, estimated execution cost, *etc.*). In the rest of this section, we describe how this is implemented in OPT++ in a query-algebra-independent manner.

As mentioned earlier, a key feature of OPT++ is that a few abstract classes and their virtual methods are defined *a priori* and the search strategy is written **entirely** in terms of these classes. Figure 2 gives an overview of the OPT++ architecture.

We first describe the abstract classes that OPT++ uses to represent operator trees and access plans and compute their properties. We then describe the abstract classes that it uses to generate and manipulate different operator trees and their corresponding access plans.

3.2 Representing Operator Trees and Access Plans

In this section, we describe the `Operator` and `Algorithm` abstract classes. These classes are used to represent operator trees and access plans, and for computing their properties.

For each abstract class, we describe what the abstract class represents, and the virtual methods on it. We describe how the search strategy uses that abstract class. To illustrate, we give examples of actual classes that an Optimizer–Implementor might derive from these abstract classes to implement a simple relational query optimizer.

3.2.1 The `Operator` Class

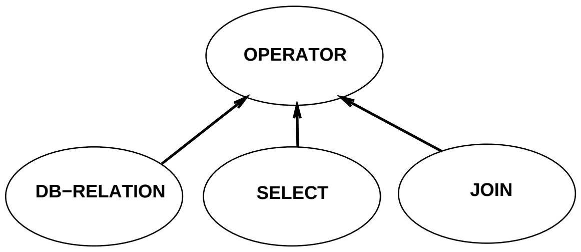


Figure 3: Operator Class Hierarchy for a Relational Optimizer

The abstract `Operator` class represents operators in the query algebra. From the `Operator` class the Optimizer–Implementor is expected to derive one class for each operator in the actual query algebra. An instance of one of these derived operator classes represents the application of the corresponding query language operator. As an example, the classes that an Optimizer–Implementor might derive from the `Operator` class to implement a simple SQL optimizer are shown in Figure 3[1]. The `Select` and `Join` classes represent the relational select and the relational join operators respectively. The `DBRelation` operator is explained in the next paragraph. In this SQL optimizer, an instance of the `Select` operator will represent an application of the select operator to one input relation, and an instance of the `Join` operator will represent an application of the join operator to two input relations.

The inputs of an operator can either be database entities (for example, relations for a relational database) that already exist in the database, or they can be the result of the application of other operators. An operator tree can thus be represented as a tree of instances of the operator class (more accurately, an instance of a class derived from the abstract `Operator` class). Each operator instance represents the corresponding query language operator being applied to the output produced by the child-nodes of that operator in the operator tree. Dummy operators serve as leaf nodes of this operator tree. These dummy operators represent database entities that already exist in the database. For example, the relations in the from clause of an SQL query are represented by the dummy `DBRelation` operator in all our examples.

[1]In all our figures, classes are represented by ovals and an arrow between classes indicates inheritance.

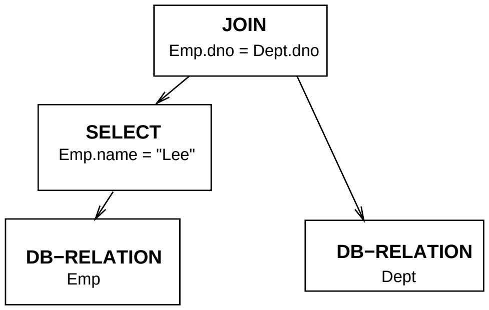


Figure 4: An Example Operator Tree

Figure 4 shows an example of an operator tree[2]. (This operator tree corresponds to the query shown in Figure 1.) The two instances of the `DBRelation` class represent the two relations in the from clause of the query – `Emp` and `Dept`. The instance of the `Select` class represents a selection predicate being applied to the `Emp` relation. The instance of the `Join` class represents the `Dept` relation being joined to the result of the selection.

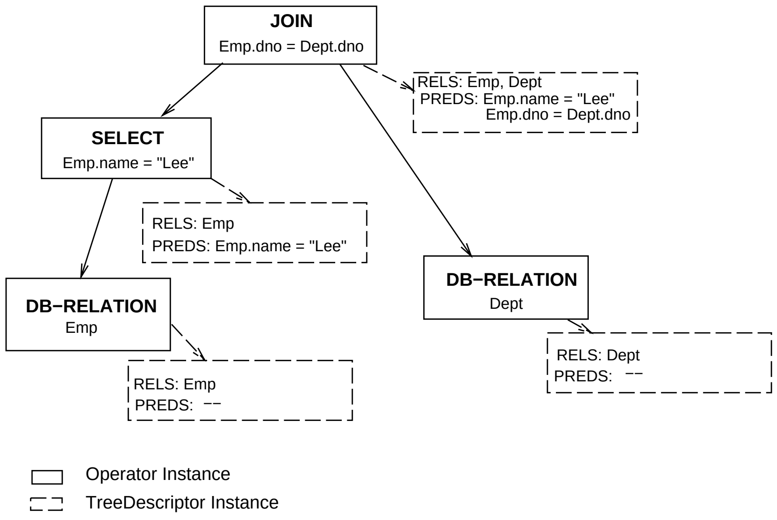


Figure 5: An Example Operator Tree with its Tree Descriptors

During the course of optimization, the optimizer needs to compute and keep track of the properties of the resultant output of an operator tree. For example, a simple relational optimizer needs to estimate properties such as the cardinality, or the size of the relation resulting from the execution of an operator tree. Since such information depends upon the query algebra, OPT++ has to rely on the Optimizer–Implementor to provide these properties. To do this, the Optimizer–Implementor is expected to define a `TreeDescriptor` class that stores information about an operator tree. The information stored could be logical algebraic properties (*e.g.*, set of relations already joined in, predicates applied), estimated properties (*e.g.*, number of tuples in output) or any other information of interest to the Optimizer–

[2]To distinguish classes from class instances, we have used ovals to represent classes and boxes to represent instances in our figures. Thus class hierarchies will be drawn using ovals, while operator trees and access plans will be drawn using boxes.

Implementor.

Every operator instance contains a pointer to an instance of the `TreeDescriptor` class, that stores information about the operator tree rooted at that operator instance. Figure 5 reproduces the operator tree of Figure 4 showing the `TreeDescriptor` instances associated with each operator instance. In this example, each `TreeDescriptor` instance stores all the relations joined in and all the predicates applied by the whole operator tree rooted at the corresponding operator instance.

With the `TreeDescriptor` class the Optimizer–Implementor has to provide an `IsEquivalent` method that determines whether two `TreeDescriptor` instances are equivalent. Two `TreeDescriptor` instances should be equivalent if the corresponding operator trees are algebraically equivalent. The `TreeDescrip-tor` also should have an `IsCompleteQuery` method that determines whether the corresponding operator tree represents the whole query or just a sub-computation.

The `Operator` class has a virtual method called `DeriveTreeDescriptor`. This is invoked on an operator instance to construct the `TreeDescriptor` object for the operator tree rooted at that operator instance, given the `TreeDescriptor` instances of its inputs.

The `Operator` class has another virtual method called `CanBeApplied` that determines whether that operator can be legally applied to given inputs according to the rules of the query algebra.

Given an operator tree, the search strategy can compute the `TreeDescriptor` for it by invoking the `DeriveTreeDescriptor` method on each of the operator instances in the tree. Note that the search strategy just invokes the methods on the abstract `Operator` class and does not need to have any information about the actual class of each instance. Through runtime binding, the proper `DeriveTreeDescrip-tor` method is invoked and the correct `TreeDescriptor` computed. Thus the search strategy (which is implemented in terms of the abstract `Operator` class) can compute the correct `TreeDescriptors` for an operator tree even though it has no knowledge of the actual operators in the query algebra. Then by invoking the `IsCompleteQuery` method on the resultant `TreeDescriptor` instance it can determine whether that operator tree represents the complete input query. Similarly it can determine whether two operator trees are equivalent using the `IsEquivalent` method. It can invoke the `CanBeApplied` method on the operator instances in an operator tree to determine whether that operator tree is legal.

3.2.2 The `Algorithm` Class

Representation of access plans is very similar to that of operator trees. The `Algorithm` abstract class is used to represent physical execution algorithms used to implement operators in the database system. The Optimizer–Implementor is expected to derive one class from the `Algorithm` class to represent each of the actual algorithms in the system.

Figure 6 shows the algorithm classes that were derived from the abstract `Algorithm` class for our simple SQL optimizer. The `HeapFile` and `Index` algorithms are dummy algorithms for the `DBRela-tion` operator as explained earlier. The `SelectScan` algorithm used to implement the `Select` operator represents a sequential scan of a `HeapFile` that outputs tuples satisfying a select-predicate. The `Ind-exSelect` uses a B-Tree `Index` to implement the same operation. `NestedLoopsJoin` and `MergeJoin` are algorithms to implement the `Join` operator. The `Sort` algorithm is not associated with any operator, but is used to enforce a sort-order among the tuples of a relation.

An access plan can thus be represented as a tree of instances of algorithm classes. As a special

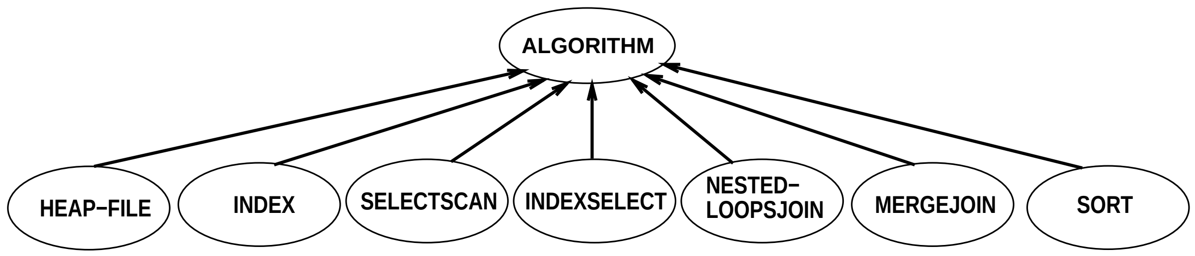


Figure 6: Algorithm Class Hierarchy for a Relational Optimizer

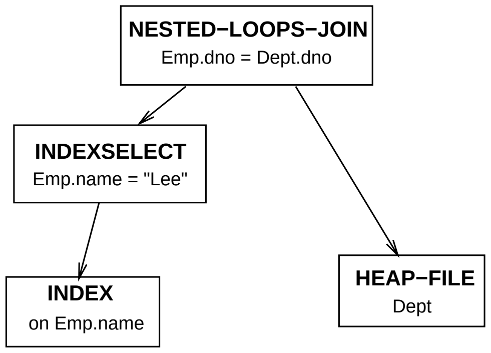


Figure 7: An Example Access Plan

case, we note that leaf nodes of access plans are represented by dummy "algorithms" representing access paths that exist on the database entities. For example, a relation may be accessed either as a sequential (heap) file or via an index. We use the `HeapFile` and `Index` dummy algorithm classes to represent these cases in our examples. Note that these algorithm classes are associated with the dummy `DBRelation` operator class defined in the previous section.

Figure 7 shows an example access plan. (This is an access plan corresponding to the operator tree in Figure 4.) An `Index` on `Emp.name` is used by the `IndexSelect` algorithm to perform the selection on '`Emp.name = "Lee"`'. The `NestedLoopsJoin` algorithm takes the result of the `IndexSelect` and joins it with the `Dept` relation using the `HeapFile` access method (implying a sequential scan).

Similar to the `TreeDescriptor` class in the case of operator trees, OPT++ requires a `PlanDescriptor` class to store physical properties of an access plan. For example, for a relational optimizer the `PlanDescriptor` class might store the sort-order of the result. Figure 8 reproduces the access plan of Figure 7 showing the `PlanDescriptor` instances associated with each algorithm instance.

The Optimizer–Implementor should provide an `IsEquivalent` method for the `PlanDescriptor` class to determine whether the physical properties of two access plans are the same. This class should also provide an `IsInteresting` method that specifies whether the result of the corresponding access plan has any *interesting* physical properties[3].

The abstract `Algorithm` class has a `DerivePlanDescriptor` virtual method. This method is invoked on an algorithm instance to construct the `PlanDescriptor` instance for the access plan rooted at the

[3]A physical property (such as sort-order) is interesting if it might help some later operation to be carried out cheaply. For example, a sort-order is interesting if it will be useful in a sort-merge join later on [SAC+79].

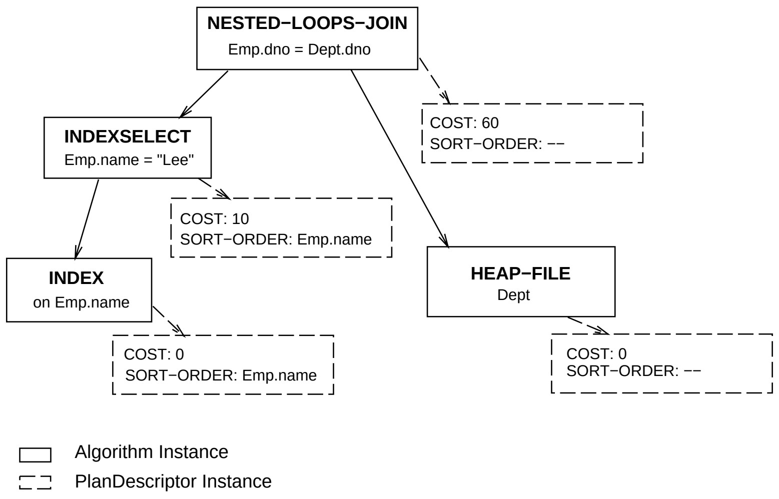


Figure 8: An Example Access Plan with its Plan Descriptors

algorithm instance, given the `PlanDescriptor` instances of its inputs.

The `Algorithm` class also has a virtual method called `Cost` that computes the estimated cost of executing the algorithm with the given inputs. This cost is used by the search strategy for pruning sub-optimal plans.

In addition, the `Algorithm` class has an `InputConstraint` virtual method. This method indicates what physical properties an input should have for it to be usable by that algorithm. For example, the merge-join operator requires that its inputs be sorted on the join attributes. As described in a later section, the search strategy will try to use this information to automatically enforce those physical properties.

A database system might have special execution algorithms that do not correspond to any operator in the logical algebra, for example sorting and decompression. The purpose of these algorithms is not to perform any logical data manipulation but to enforce physical properties in their outputs that are required for subsequent query processing algorithms. These are referred to as *enforcers* in the Volcano Optimizer Generator [GM93], and are comparable to the *glue operators* in Starburst [LFL88]. Classes corresponding to such *enforcers* should also be derived from the `Algorithm` class. For example, in a relational query optimizer, the `Sort` algorithm is an enforcer that can be used to ensure that the inputs of the `MergeJoin` algorithm are sorted on the join attribute.

Given an access plan, the search strategy can use the virtual methods of the abstract `Algorithm` class to determine properties of the access plan, estimate its cost, and determine equivalence of different access plans. All of this is achieved by invoking these methods on the abstract `Algorithm` class without any knowledge of the actual algorithms in the database system.

3.3 Generating Operator Trees and Access Plans

In the previous section we saw how operator trees and access plans are represented in OPT++. If the search strategy is given an operator tree or an access plan, we saw how it can compute its properties and compare it with other trees or plans by using the virtual methods of the `Operator` and `Algorithm` abstract classes.

In this section, we describe how the various operator trees and access plans are generated by the search strategy during the course of optimization.

3.3.1 The `TreeToTreeGenerator` Class

Classes derived from the `TreeToTreeGenerator` abstract class are used to generate various operator trees. These classes have a virtual method called `Apply` that takes an existing operator tree and creates one or more new operator trees.

Let us consider the System-R style [SAC$^+$79] search strategy to illustrate the concept behind the `TreeToTreeGenerator` class. Such an optimizer starts with single relations and then builds bigger and bigger operator trees from them by first applying selections and then applying joins to them. At each step, the search strategy picks an existing operator tree and then *expands* it to get a bigger operator tree by applying a new select operation or a join operation at the top of the tree.

The process of *expanding* an existing operator tree by applying one more operator to it and generating new trees can be accomplished by using `TreeToTreeGenerator` classes.

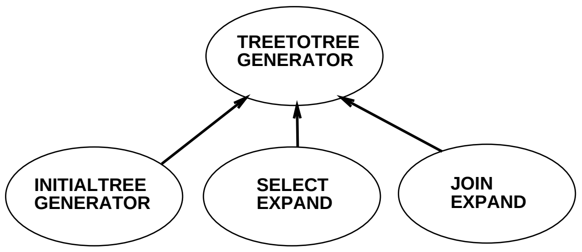


Figure 9: Example `TreeToTreeGenerator` Class Hierarchy

Specifically, to implement a relational System-R style optimizer, the Optimizer–Implementor can derive from the `TreeToTreeGenerator` abstract class a `SelectExpand` class to generate applications of the `Select` operator and a `JoinExpand` class to generate applications of the `Join` operator. See Figure 9. The `SelectExpand::Apply` method is expected to take an operator instance (representing an operator tree) and create one or more new instances of the `Select` operator representing application of some selection to the input operator tree. Similarly the `JoinExpand::Apply` method should create various `Join` instances representing different ways of applying a join to the given input.

Figure 10(b) illustrates the `JoinExpand::Apply` method being invoked during the optimization of the query in Figure 10(a). The figure shows an instance of the `Select` operator that represents the predicate '`Emp.name = "Lee"`' being applied to the `Emp` relation. The `JoinExpand::Apply` method is invoked in order to expand the operator tree rooted at that `Select` operator instance. Since the result of the select can be joined either with the `Dept` relation or with the `Job` relation, two instances of the `Join` operator are created as shown in the figure.

The `TreeToTreeGenerator` class also has a virtual method called `CanBeApplied` that determines whether that `TreeToTreeGenerator` can be applied to a given operator instance.

One class derived from the `TreeToTreeGenerator` class is designated by the Optimizer–Implementor as the `InitialTreeGenerator`. The `Apply` method of this class is used by the search strategy to start

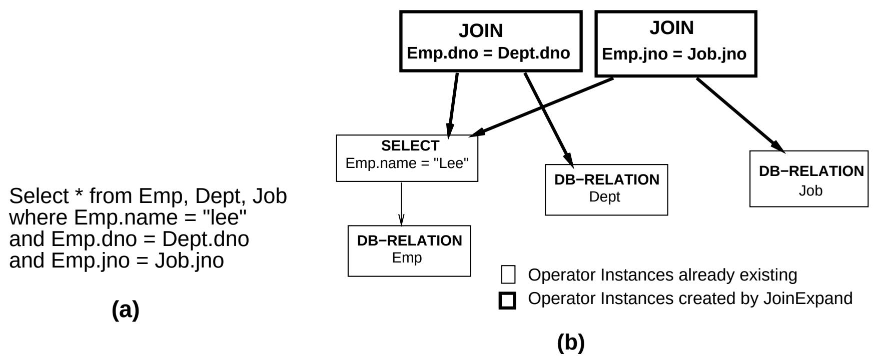


Figure 10: Application of `JoinExpand::Apply`

the optimization process. For the relational optimizer the `InitialTreeGenerator` creates one `DBRelation` instance for each relation in the from clause. After that, the search strategy picks some operator instance (representing an operator tree) and generates new operator trees from it by invoking the `Apply` method of various `TreeToTreeGenerator` classes on it. The `CanBeApplied` method is used to determine whether the `TreeToTreeGenerator` should be applied to that operator instance. This process can be repeated to generate various operator trees corresponding to the input query.

Note that the search strategy does not need to know any details about the `TreeToTreeGenerator` classes in the system. All it needs is a list containing a pointer to one instance of each of the `TreeToTreeGenerator` classes. By invoking the virtual methods of the `TreeToTreeGenerator` abstract class on these instances, the search strategy can generate all operator trees required for optimization.

3.3.2 The `TreeToPlanGenerator` Class

An access plan can be generated from an operator tree by replacing each operator instance in the operator tree by an instance of an algorithm class that can be used to implement that operator. Classes derived from the `TreeToPlanGenerator` abstract class are used to generate algorithm instances corresponding to an operator instance.

The `TreeToPlanGenerator` abstract class has a virtual method called `Apply` that takes an operator instance as an input parameter and creates one or more new algorithm instances representing different ways of using physical execution algorithms to execute the operation represented by that operator instance.

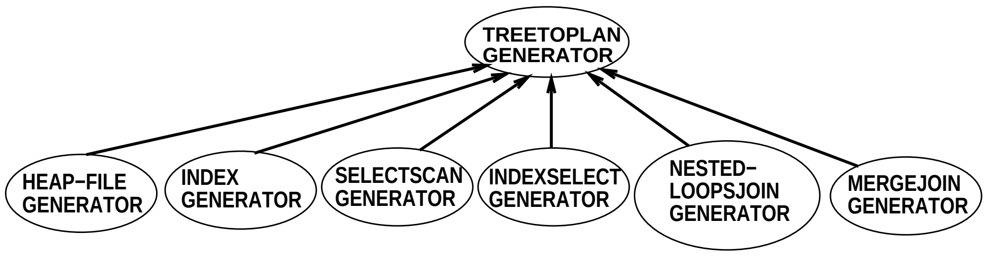


Figure 11: Example `TreeToPlanGenerator` Class Hierarchy

For example, consider a relational optimizer. From the `TreeToPlanGenerator` class the Optimizer–Implementor might derive one class corresponding to each algorithm in the system. Each of these classes takes an operator instance and creates one or more algorithm instances indicating how the corresponding algorithm can be used to implement that operation. Figure 11 shows the classes derived from the `TreeToPlanGenerator` class.

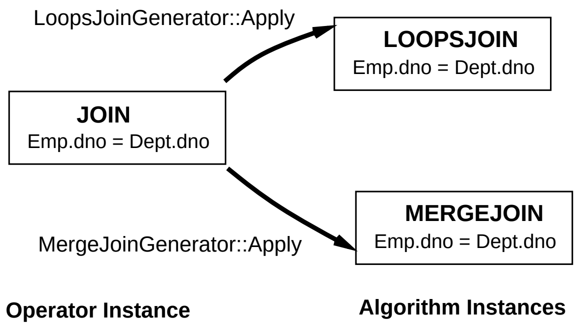


Figure 12: Examples of `TreeToPlanGenerator::Apply`

Figure 12 shows some examples of `TreeToPlanGenerator::Apply` being applied to a join operator instance. As can be seen, the `NestedLoopsJoinGenerator::Apply` results in an instance of the `NestedLoopsJoin` class being created while the `MergeJoinGenerator::Apply` results in an instance of the `MergeJoin` class being created.

Given an operator tree, the search strategy can invoke the `Apply` method of various `TreeToPlanGenerator` classes on each of the operator instances in the tree to generate various access plans corresponding to the operator tree.

The `TreeToPlanGenerator` class has a `CanBeApplied` virtual method that determines whether that `TreeToPlanGenerator` can be applied to the given operator instance.

Note that the search strategy does not need to know any details about the actual `TreeToPlanGenerator` classes in the system. All it needs is a list containing a pointer to one instance of each of the actual `TreeToPlanGenerator` classes. By using this list and invoking virtual methods on the instances in this list, the search strategy is able to enumerate all the access plans for any operator tree.

3.3.3 The `PlanToPlanGenerator` Class

The `PlanToPlanGenerator` class is used to further modify an access plan after it has been generated. The `PlanToPlanGenerator::Apply` virtual method takes an algorithm instance (representing an access plan) and creates one or more new algorithm instances each representing some other access plan.

An important use of this class is to automatically insert into an access plan instances of enforcers that can change the physical properties of the output of some access plan. This might be required in order to satisfy input constraints of some algorithm. For instance, in a relational optimizer, the `SortEnforcer` class can be derived from the `PlanToPlanGenerator` class to enforce various sort-orders on results of access plans.

Figure 14 illustrates the use of the `SortEnforcer::Apply` virtual method. This method is invoked with the `IndexSelect` instance as an input parameter; it creates a new instance of the `Sort` algorithm

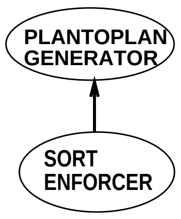


Figure 13: `PlanToPlanGenerator` Class Hierarchy

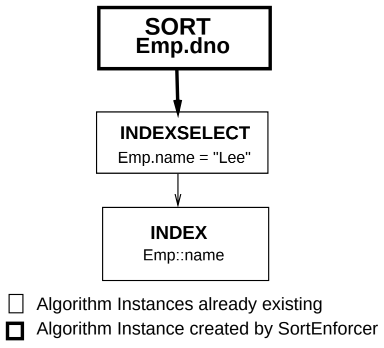

SORT Emp.dno
INDEXSELECT Emp.name = "Lee"
INDEX Emp::name
Algorithm Instances already existing
Algorithm Instance created by SortEnforcer

Figure 14: Use of `SortEnforcer::Apply` to enforce a sort-order

(enforcer) as shown in the figure.

The `PlanToPlanGenerator` class also has a `CanBeApplied` virtual method that determines whether the `PlanToPlanGenerator` can be applied to the given input.

During the course of optimization, when the search strategy is building various access plans using the `TreeToPlanGenerator` classes, it invokes the `InputConstraint` method whenever a new algorithm instance is created. If it turns out that the inputs of that algorithm instance do not satisfy the input constraints, it attempts to rectify the situation by applying an appropriate `PlanToPlanGenerator`. The search strategy uses the `CanBeApplied` virtual method of the `PlanToPlanGenerator` classes to determine which generators can be used to enforce the given properties, and invokes the `Apply` method to create new access plans that satisfy the corresponding input constraints. Thus the enforcers will get applied automatically without the Optimizer–Implementor having to worry about them.

3.4 The Search Strategies

So far, we have seen the `Operator`, `Algorithm`, and various tree and plan `Generator` classes. As described in the previous sections, any search strategy that is implemented entirely in terms of these abstract classes and their virtual methods becomes independent of the query algebra in the sense that the actual operators, algorithms and generators in the system can be modified without modifying the search strategy code.

A number of search strategies have been implemented in OPT++ in this query-algebra-independent manner. The implementation of the various search strategies is loosely modeled on the object-oriented scheme described in [LV91]. OPT++ defines a `SearchStrategy` abstract class with virtual methods, and each of the search strategies in OPT++ is actually implemented as a class derived from the `Search-`

`Strategy` abstract class. Any of these search strategies can be used for optimization by the Optimizer–Implementor by declaring an object of the corresponding class and invoking the `Optimize` virtual method on that object. Another consequence of this design is that Optimizer–Implementor can modify the behavior of any search strategy by deriving a new class from it and redefining some of the virtual methods. Due to space constraints, we do not describe this in detail. Refer to [LV91] to get an idea of how this works. In this section we concentrate on describing how the various search strategies are implemented in terms of the `Operator`, `Algorithm`, and `Generator` abstract classes, and in the next section we describe how the Optimizer–Implementor can easily switch from one search strategy to another.

We describe below the various search strategies that are currently implemented in OPT++. The "Bottom-up" search strategy is similar to the one used by the System-R optimizer [SAC$^+$79]. The "Transformative" search strategy is based upon the search engine of the Volcano Optimizer Generator [GM93]. Finally, three randomized search strategies, Iterated Improvement [SG88], Simulated Annealing [IW87], and Two Phase Optimization [IK90], have been implemented.

3.4.1 The Bottom-up Search Strategy

This search strategy can be used to implement optimizers that use bottom-up dynamic-programming similar to the System-R optimizer [SAC$^+$79].

The `InitialTreeGenerator` is invoked to initialize the collection of operator trees. To generate bigger trees, the search strategy picks an existing operator tree and *expands* it. To expand an operator tree, it determines what `TreeToTreeGenerators` can be applied to the operator instance at its root by invoking the `CanBeApplied` method of the various `TreeToTreeGenerators`. Then the `Apply` method of each of the applicable `TreeToTreeGenerators` is invoked to get new operator trees.

For each new operator tree, all the corresponding access plans are generated. This is done by applying various `TreeToPlanGenerators` to the operator instances in the tree to get the corresponding algorithm instances.

Cost-based pruning of access plans is done in a manner similar to the techniques used by the System-R optimizer. Whenever a new access plan is created, The virtual methods of the `Algorithm` class are used to determine the *cost* of that access plan, to determine whether it has any *interesting* physical properties, and to locate all other access plans that are equivalent to it. From this set of equivalent access plans, only the cheapest plan and those plans that have *interesting* physical properties are retained. All others are deleted[4].

Optimization is complete when none of the operator trees can be further *expanded.* At this point the cheapest access plan that represents the complete input query is returned as the optimal plan. The `IsCompleteQuery` method is used to determine whether or not an access plan represents the complete input query.

[4]To "delete" an access plan, only the algorithm instance at the root of that access plan is actually deleted. The other algorithm instances in the access plans are not deleted because they maybe shared by other access plans.

3.4.2 The Transformative Search Strategy

In Section 3.3.1 we have only given examples of `TreeToTreeGenerators` that *expand* a given tree by applying a new operator to it. However, we can also have `TreeToTreeGenerator` classes that transform an operator tree into another, algebraically-equivalent operator tree. In other words, a class derived from the `TreeToTreeGenerator` class can represent an algebraic transformation rule (such as those used by the Volcano Optimizer generator). The `CanBeApplied` method determines whether the transformation rule is applicable to a given operator tree, and the `Apply` method creates the new tree that results from the transformation.

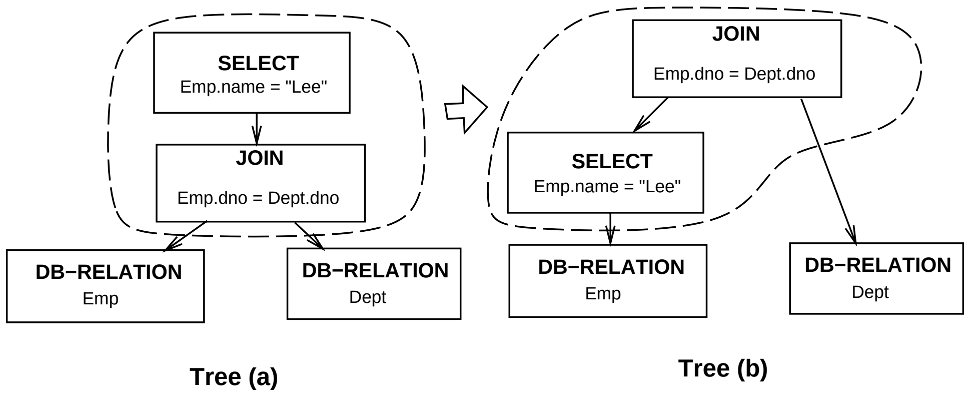


Figure 15: A Rule-based Transformation

Figure 15 shows an example of a transformative `TreeToTreeGenerator` being applied. Assume that a class called `SelectPushDown` is derived from the the `TreeToTreeGenerator` class. This class represents the following transformation rule: "If a join is immediately followed by a select, and if the select predicate only references attributes from the left input of the join, then the select can be pushed below the join into its left input tree." Figure 15 shows the result of `SelectPushDown::Apply` being invoked on an operator tree. It is applied to Tree (a) and the new operator tree resulting from the transformation is shown in Tree (b). This new tree is generated by creating the two new operator instances shown in the oval in Tree (b). The new `Select` operator instance represents the selection predicate being applied to `Emp` relation. The new `Join` operator instance represents the result of that select being joined with the `Dept` relation. When these two new operator instances are created, we have a new operator tree that is equivalent to the old one.

The search strategy invokes the `InitialTreeGenerator` to get one operator tree corresponding to the input query. It then repeatedly applies `TreeToTreeGenerators` (transformation rules) to the existing operator trees to generate equivalent operator trees. As before, the `CanBeApplied` method is used to determine whether a `TreeToTreeGenerator` can be applied to an operator tree, and the `Apply` method is used to generate the new tree.

The procedure for generation of access plans corresponding to an operator tree, and for their pruning is similar to that used in the bottom-up search strategy. Note that our `TreeToPlanGenerator` classes are analogous to the implementation rules of the Volcano Optimizer Generator [GM93].

Optimization is complete when none of the existing operator trees can be further transformed.

3.4.3 Randomized Search Strategies

In this section, we briefly describe the implementation of the randomized search strategies in OPT++. As with the Transformative strategy, these algorithms assume that the classes derived from the `TreeToTree-Generator` class represent algebraic transformation rules. Here we briefly describe the implementation of the Simulated Annealing Algorithm. The implementation of the other algorithms is very similar, and is omitted for brevity.

The Simulated Annealing algorithm has a a variable called *temperature* that is initialized before optimization is begun. The `InitialTreeGenerator` is then used to generate one complete operator tree. The `TreeToPlanGenerator` classes are used to create an access plan corresponding to that operator tree. After this, at each step a random operator instance in the operator tree is picked for processing. Then a random `TreeToTreeGenerator` or a random `TreeToPlanGenerator` is chosen and applied to that operator instance. This gives rise to a new access plan. The cost of the new plan is estimated. The search strategy *accepts* or *rejects* the new plan with a probability that depends upon the difference between the costs of the old plan and the new plan, and upon the *temperature.* If the new plan is rejected, the new plan is deleted and the old plan remains the *current* plan. If the new plan is accepted, the old plan is deleted, and the new plan becomes the *current* plan.

The *temperature* is decreased after each step, and the process is repeated. Optimization continues until the temperature becomes zero and there is no improvement in the cost for some number of steps. At this point, the current plan is output as the optimal plan.

3.5 Extensibility in OPT++

This section summarizes what is involved in implementing a new optimizer, or extending or modifying an existing optimizer built using OPT++. Section 4 has some examples of such extensions as applied to a real optimizer.

3.5.1 Implementing a new Optimizer

Figure 16 shows the overall system architecture of an optimizer implemented using OPT++.

The Search Strategy Component : This represents the code that is provided with OPT++, and includes the implementations of the various search strategies. This part of the code is completely independent of the actual query algebra and the database system, and therefore does not have to be modified to implement a particular optimizer. Thus a large part of the code required for an optimizer is already provided with OPT++ and does not have to be written by the Optimizer–Implementor.

The Algebra Component : This contains the classes derived by the Optimizer–Implementor from the `Operator` and the `Algorithm` classes, and also the implementation of the `TreeDescriptor` and `PlanDescriptor` classes. This part of the code depends only upon the query algebra and the physical implementation algorithms available in the database system. Specifically, this code does not have to be changed when the optimizer is modified to use a different search strategy (*e.g.*, switching from a transformative strategy to simulated annealing) or when the search space explored is changed (*e.g.*, switching from left-deep join tree enumeration, to bushy join tree enumeration).

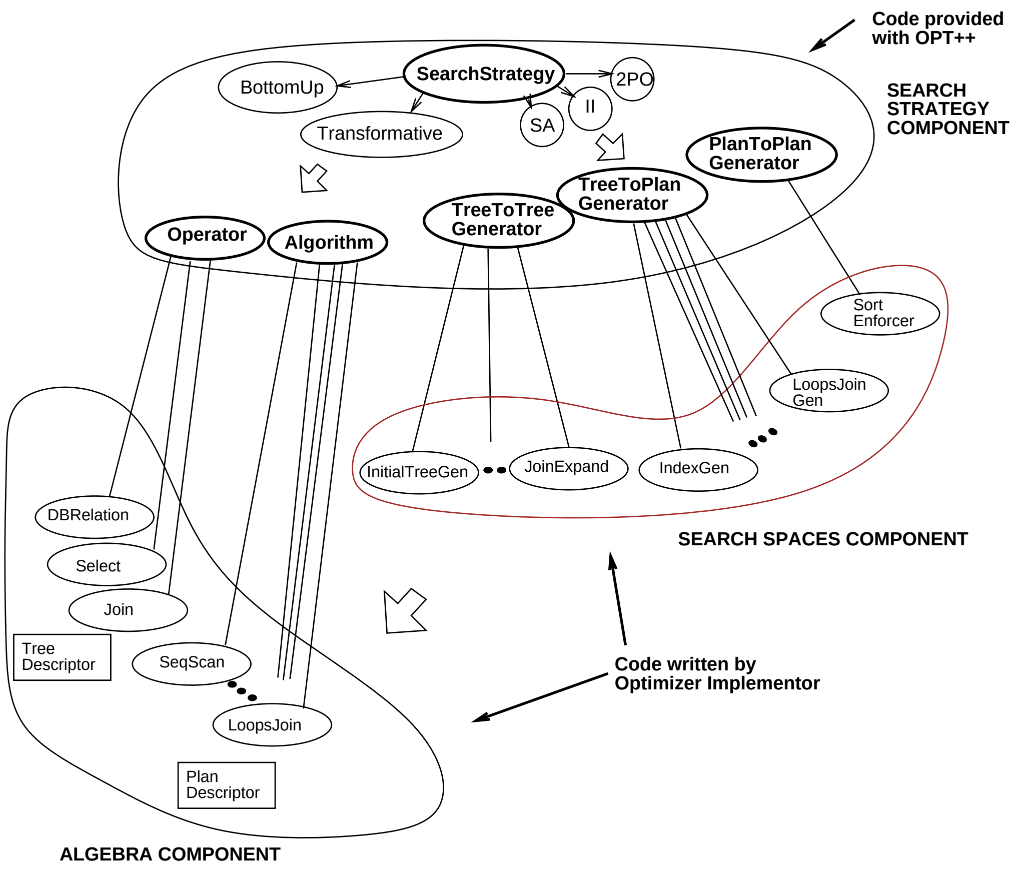


Figure 16: Implementing an Optimizer in OPT++

The Search Space Component : This contains the classes derived by the Optimizer–Implementor from the `TreeToTreeGenerator`, `TreeToPlanGenerator`, and the `PlanToPlanGenerator` classes. These classes are used to decide what operator trees and access plans are generated, and hence play a large part in controlling the search space that is explored by the search strategy. For example, implementing a `JoinExpand` class that only generates joins in which the inner relation is a base relation restricts the search space to the space of left-deep join trees. On the other hand, implementing a `BushyJoin-Expand` class that considers composite inners will generate all bushy trees. In fact, the various join enumeration algorithms described in [OL90] can each be implemented in OPT++ as a class derived from the `TreeToTreeGenerator` class.

3.5.2 Modifying the Optimizer

Changing the logical or physical Algebra : To modify the optimizer to incorporate a new physical implementation algorithm, a new class corresponding to that algorithm must be derived from the `Al-gorithm` class. A new class also must be derived from the `TreeToPlanGenerator` class to indicate how this new algorithm can be used to implement the corresponding operator. Thus, adding an algorithm only involves adding some new classes to the optimizer. None of the existing code needs to be changed. For instance, a hash-join algorithm can be incorporated into our simple relational optimizer by deriving a `HashJoin` class from the `Algorithm` class, and a `HashJoinGenerator` class from the `TreeToPlanGenerator` class.

Similarly, adding an operator requires deriving a new class from the `Operator` class and deriving one or more new classes from the `TreeToTreeGenerator` class. Algorithms used to implement the new operator also have to be added as described above.

Changing the Search Space : As mentioned earlier, the search space explored by any search strategy is controlled by the `Generator` classes. It can be changed by adding a new `Generator` class, or by removing or modifying an existing `Generator` class. For example, in our simple relational optimizer, the search space can be changed from the space of left-deep join trees to the space of bushy join trees by adding a `BushyJoinExpand` class.

Since all the search strategy code is in the Search Strategy Component, and all the code that depends only on the query algebra is in the Algebra Component, there is very little code left in the Search Space Component. Thus changing generator code or adding a new generator is easy. In other words, experimenting with various different search spaces or optimization techniques is considerably simplified by the OPT++ architecture.

Changing the Search Strategy : OPT++ offers a choice of search strategies, and makes it relatively easy to switch from one search strategy to another. Often, one search strategy can be replaced by another without changing any of the code in the “Algebra” or “Search Space” component. This is the case if the search strategy is changed from the Transformative Strategy to one of the randomized strategies, or *vice versa*. Sometimes changing from one search strategy to another might require writing new `TreeToTreeGenerator` classes. For example, switching from a bottom-up System-R-like strategy to a transformative strategy requires replacing all the `TreeToTreeGenerator` classes (that are based on the concept of *expanding* an operator tree) with new `TreeToTreeGenerator` classes that representing the transformation rules. However, since there is very little code in the `TreeToTreeGenerator` classes, this change is rather easy. We describe a specific example in Section 4.

4 Experiences with OPT++

In this section, we describe some experiences we had implementing optimizers using OPT++. We started with a simple relational optimizer that does System-R style join enumeration and then modified it in various ways – to change the search space; to extend it to accept a more complex query algebra; and to change the search strategy used for optimization. This was done with the intention of illustrating the ease of use and extensibility of OPT++. We also report on some performance studies – including a performance comparison with an optimizer generated using the Volcano Optimizer Generator [GM93] – to show that, in spite of its flexibility, OPT++ is efficient.

4.1 Join Enumeration

In this section we consider a simple relational optimizer that does System-R style join enumeration, and describe how it was easily extended to consider the space of bushy join trees, and also to consider cartesian products.

Since all the examples used in Section 3 describe this simple relational optimizer, we will not repeat the details here. Briefly, the `DBRelation`, `Select`, and `Join` classes were derived from the `Operator` class to represent the relational operators, and the `HeapFile`, `Index`, `SelectScan`, `IndexSelect`, `NestedLoopsJoin`, and `MergeJoin` classes were derived from the `Algorithm` class to represent the corresponding physical implementation algorithms. `SelectExpand` and `JoinExpand` were derived from the the `TreeToTreeGenerator` class. `HeapFileGenerator`, `IndexGenerator`, `SelectScanGenerator`, `IndexSelectGenerator`, `NestedLoopsJoinGenerator`, and `MergeJoinGenerator` were derived from `TreeToPlanGenerator` to indicate how the corresponding algorithms could be used to implement the associated operators. `SortEnforcer` is derived from `PlanToPlanGenerator` to enforce sort orders.

We note that the `SelectExpand::Apply` method was written so as to apply all selection predicates as soon as possible (the "select pushdown" heuristic) and the `JoinExpand::Apply` method allowed only single relations as the inner (right-hand) input for the join operation (the "left-deep join trees only" heuristic). Thus the search space consisted only of operator trees that had the selections pushed down as far as possible, there were no cartesian products, and the join trees were left deep.

The "Algebra" component that includes the various operator and algorithm classes as well as the `TreeDescriptor` and `PlanDescriptor` classes consists of about 900 lines of code. The "Search Space" components that includes classes derived from the `TreeToTreeGenerator`, `TreeToPlanGenerator`, and `PlanToPlanGenerator` classes consists of 150 lines of code. In contrast, the "Search Strategy" component, which consists entirely of code that is provided with OPT++ (*i.e.*, the Optimizer–Implementor does not have to write this code) was about 2500 lines of code. The fact that the search strategy code is already provided and does not have to be written or modified by the Optimizer–Implementor considerably simplified the task of writing the optimizer. Further, as will become clear later, the fact that the "Search Space" component is very small (150 lines of code divided over 10 classes) makes it very easy to experiment with various optimization techniques.

We decided to modify the search space explored to include bushy join trees as well as join trees that contain cartesian products. To do this we derived the `BushyJoinEnumerator` and `CartesianJoinEnumerator` classes from the `TreeToTreeGenerator` class to generate instances of the `Join` operator that allowed composite inners (*i.e.*, the inner operand is allowed to be the result of a join), and those containing cartesian products[5]. This resulted in the addition of about 150 lines of code to the "Search Space" component. (Refer to Figure 16.)

As an experimental evaluation of the optimizer, we studied its performance (optimization time and estimated execution cost) as a function of the number of joins in the input query. For each query size (number of joins) 10 different queries were generated randomly and optimized. The experiments were run on a Sun SPARC-10/40 with 32MB of memory. Virtual memory was also limited to 32MB (using the *limit* command).

Figure 17 shows the effect of different search spaces on the time taken for optimization, and Figure 18 shows the effect on the relative estimated execution costs of the optimal plans produced. (Note that optimization times are shown on a logarithmic scale.)

[5]These join enumerator classes are based on the schemes described in [OL90].

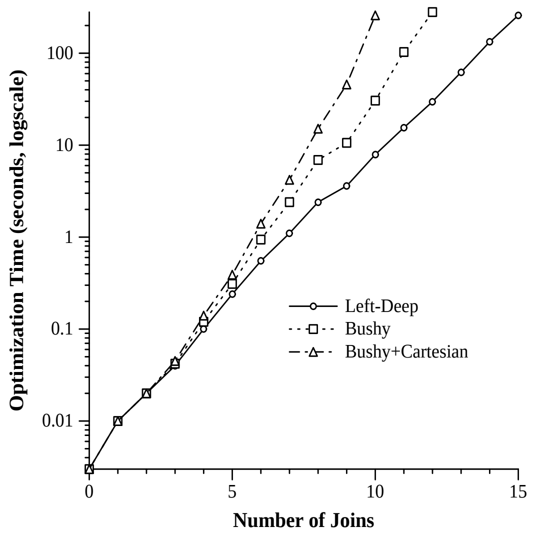


Figure 17: Comparison of Search Spaces: Optimization Times (Log-scale)

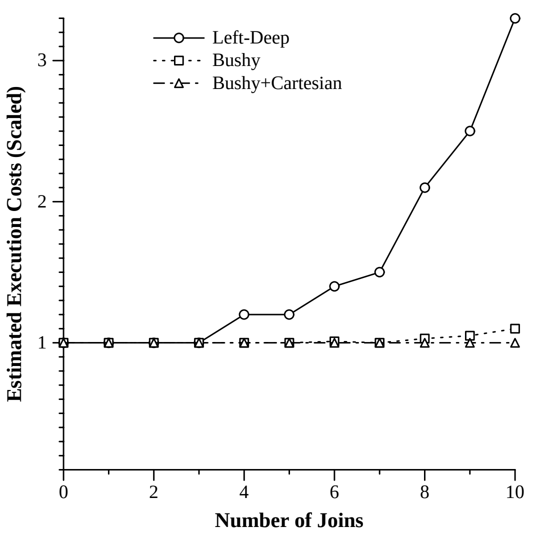


Figure 18: Comparison of Search Spaces: Estimated Costs (scaled)

4.2 A More Complex Query Algebra

In this section, we describe how the optimizer was extended to handle a more complex query algebra. The new algebra allows reference-valued attributes, set-valued attributes, and the use of path-indices.

We extended the optimizer to implement the optimization techniques described in [BMG93]. We added a `Materialize` query algebra operator that represents materialization of a reference-valued attribute (in other words, dereferencing a pointer). A corresponding `Assembly` algorithm class is used to represent the physical execution algorithm used to implement `Materialize` [KGM91]. An `Unnest` operator class and the corresponding `UnnestAlgorithm` class is used to represent unnesting of set-valued attributes.

The `MaterializeExpand` class derived from the `TreeToTreeGenerator` class takes an operator tree and expands it by adding a materialize operation that dereferences a reference-valued attribute present in its input.

Materialization of a reference-valued attribute can also be achieved using a pointer-based join [SC90]. We specialized the `JoinExpand` class by deriving a new `PointerJoinExpand` class from it. This new class creates instances of the `Join` operator that actually correspond to materialization of reference-valued attributes using a pointer-based join.

The `UnnestExpand` class derived from `TreeToTreeGenerator` takes an operator tree and expands it by adding to it an unnest operation that unnests a set-valued attribute present in its input.

The optimizer also had to be extended to handle path-indices. A select predicate involving a path-expression (like `city.mayor.name = "Lee"`) can be sometimes evaluated using a path-index without really having to materialize the individual components of the path-expression. For example, if a path-index exists on city.mayor.name, the predicate `city.mayor.name = "Lee"` can be evaluated without having to materialize the `city` or `mayor` objects (see [BMG93] for details).

A new `PathIndexSelect` algorithm was derived from the `Algorithm` class to capture such path-index scans. A `PathIndexScanGenerator` class was derived from the `TreeToPlanGenerator` class to replace occurrences of a string of materialize operators followed by a select operator in an operator tree by a single `PathIndexSelect` algorithm, if possible[6].

This extension of the optimizer to handle the new query algebra constructs resulted in an addition of about 350 lines of code to the "Algebra" component (most of it for cost and selectivity estimation) and about 100 lines of code to the "Search Strategy" component. Considering the complexity of the extensions to the algebra, and compared to the size of the whole optimizer, the changes were rather easy.

4.3 A Transformative Optimizer

As a third test of OPT++, we decided to change the optimizer from a bottom-up dynamic programming optimizer to one that uses algebraic transformation rules. In other words, a shift from the "Bottom-Up" strategy to the "Transformative" strategy. This change required that new classes be derived from the `TreeToTreeGenerator` class to represent the transformation rules. One class was used for each transformation rule. For instance, the `JoinAssociativity` class was used to represent the associativity of the join operator, while the `SelectPushDown` class was used to capture the property that selects can be pushed down under joins.

Modifying the whole optimizer to use the transformative paradigm required the addition of about 250 lines of code in the form of `TreeToTreeGenerators` representing the transformation rules[7]. We note that no code in the "Algebra" component had to be changed, while in the "Search Space" component, only new `TreeToTreeGenerators` had to be added. The old `TreeToPlanGenerator` and `PlanToPlan-Generator` classes were used unchanged.

The Transformative Search Strategy in OPT++ is based upon the search engine of the Volcano Optimizer Generator. To validate our implementation of that strategy, and to show that its performance does not suffer even though it has been implemented in the more flexible OPT++ framework, we compared it to an optimizer generated using Volcano. Using the Volcano Optimizer Generator we implemented an optimizer equivalent to our Transformative Optimizer. The two optimizers were equivalent in the sense that they used the same transformation rules and exactly the same code for cost estimation, selectivity estimation, *etc.*.

Figures 19 and 20 compare the two optimizers in terms of optimization times and memory consumed for randomly generated queries of increasing sizes. As before, the experiments were run on a Sun SPARC-10/40 with 32MB of memory. The figures show us that the performance of the Transformative Search Strategy of OPT++ is almost as good as that of the Volcano search engine. We see approximately a degradation of about 5% in the optimization times, while space utilization is roughly equivalent. (The advantage of OPT++ of course lies in the fact that it is a more general framework and in addition to all the features that Volcano provides, it also provides flexibility in terms of the search strategy including the ability to mix-and-match different search strategies.)

[6]In the interests of space and clarity, we do not describe our implementation of the mechanism by which components of the path that are not materialized into memory in because of the existence of the path-index are automatically materialized if they are needed for some other operation. The implementation is very similar to the scheme described in [BMG93].

[7]In the next section we shall see that a switch from the Transformative strategy to one of the Randomized strategies is much easier than this.

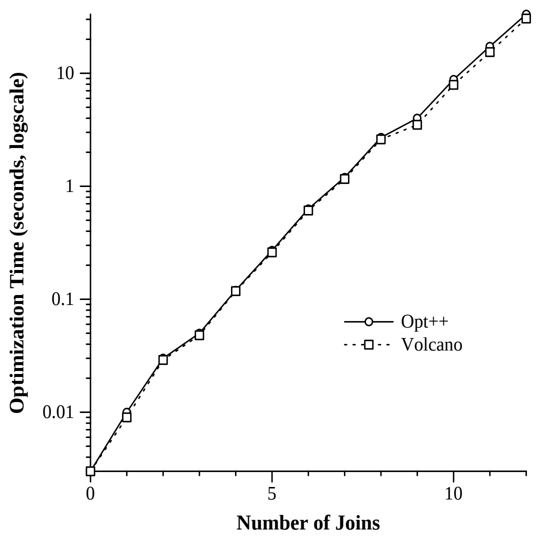


Figure 19: OPT++ *vs.* Volcano: Optimization Times (Log-scale)

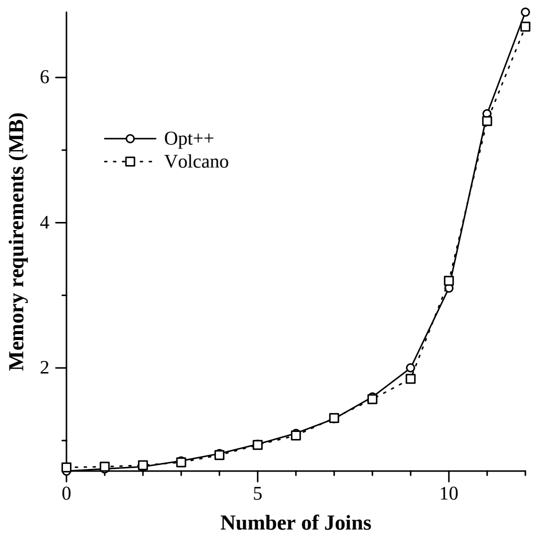


Figure 20: OPT++ *vs.* Volcano: Memory Requirements

4.4 Randomized Strategies

Finally, we modified the transformative optimizer to use the randomized search strategies available with OPT++. The only change required for this is to replace the Transformative Search Strategy object by an object of the required Randomized search strategy. Thus, changing from the Transformative search strategy to either Simulated Annealing, Iterated Improvement or Two Phase Optimization (or *vice versa*) can be trivially accomplished by changing one line of code.

We compared the performance of all the different search strategies in terms of the time taken to optimize randomly generated queries of increasing sizes, and the quality of the plans produced. The stopping conditions and other parameters for the randomized search strategies were as described in [IK90]. Figures 21 and 22 show the performance results obtained. Qualitatively, they confirm the findings of [Kan91] that for smaller queries the exhaustive algorithms consume much less time for optimization than the randomized algorithms and yet produce equivalent or better plans, while for larger queries, the randomized algorithms take much less time to find plans that are almost as good as those found by the exhaustive algorithms. They also confirm the findings of [IK90] that Two Phase Optimization performs better than Simulated Annealing or Iterated Improvement.

In Figure 23, the memory requirements of the different strategies are presented. The randomized strategies require a negligible amount of memory irrespective of the size of the input query, while the exhaustive strategies require exponentially increasing amounts of memory. Hence, for queries larger than those shown in Figure 21, the randomized strategies will continue to give reasonable performance while the exhaustive strategies will fail due to lack of enough memory. We also note that although the Bottom-Up and Transformative search strategies have comparable performance in terms of optimization time and quality of plans produced (because both are exhaustive strategies and explore the same search space), the Bottom-Up strategy has a significant advantage in space consumption as it can perform more aggressive pruning of operator trees.

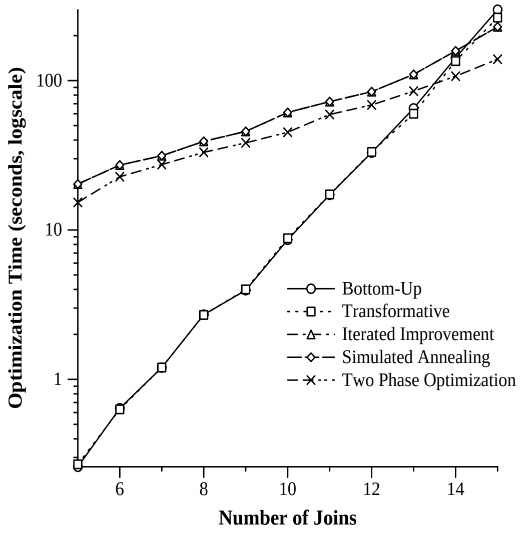


Figure 21: Comparing Search Strategies: Optimization Times (Log-scale)

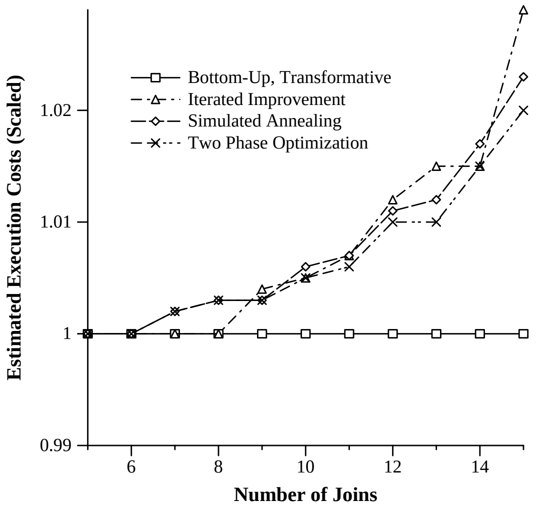


Figure 22: Comparing Search Strategies: Estimated Costs (Scaled)

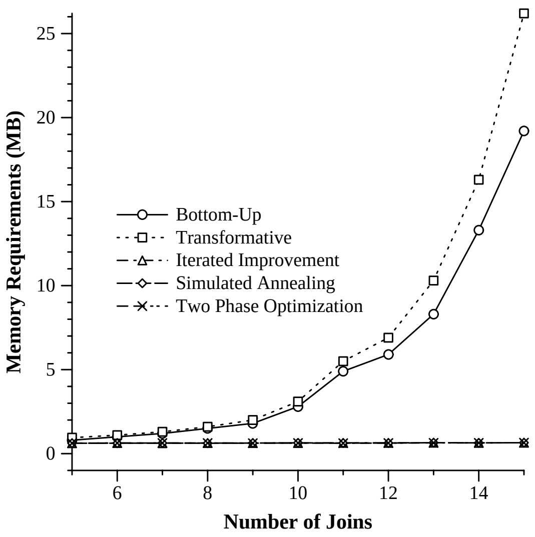


Figure 23: Comparing Search Strategies: Memory Requirements

5 Conclusions and Future Work

In this paper, we have described a new tool for building extensible optimizers. It uses an object-oriented design to provide extensibility through the use of inheritance and late binding. The design makes it easy to implement a new optimizer as well as to modify existing optimizers implemented using OPT++. Extensibility is provided in the form of the ability to easily extend the logical or physical query algebra, to easily modify the search space explored by the search strategy, and to even change the search strategy. Our experiences with the implementation of optimizers using OPT++ show that, in addition to being easy to use and extend, it is also efficient.

We believe that these features of OPT++ will make it a very useful tool for building query optimizers. First, it can be used for quickly building an efficient optimizer for a new database system and experimenting with a number of different optimization techniques and search strategies. Such experimentation can be very useful to an Optimizer–Implementor to compare different optimization strategies before deciding what strategy is best suited to that database system. Further, having multiple search strategies provides the option of dynamically determining the search strategy based on the input query and other criteria. For example, an optimizer could use an exhaustive strategy for small queries and a randomized strategy for large queries, or it could use bushy join tree enumeration for small queries and left-deep join tree enumeration for larger queries. Thus OPT++ can be used to build a smart query optimizer that dynamically customizes its optimization strategy depending upon the input.

We plan to add some more search strategies to the repertoire of strategies available in OPT++. In particular, the A* heuristic [Pea84, KMP93], and the heuristics described in [Swa89] seem promising.

We also plan to add debugging support to OPT++. Debugging an optimizer remains a complex and time-consuming task. In particular, determining the source of a bug in an optimizer that produces sub-optimal plans is difficult. ([Hel94] discusses some of the difficulties with this.) We plan to incorporate support for debugging into OPT++, including visual optimizer execution tracing, and automated detection of potential sources of errors using hints from the Optimizer–Implementor.

Finally, we plan to use OPT++ to study the optimization of complex decision-support queries, like those included in the TPC-D benchmark [Raa95]. This is a difficult problem for which OPT++'s ability to easily experiment with different search strategies and search spaces will be very useful. In addition, this will also serve as a stress test for OPT++.

Acknowledgements

We would like to thank Joey Hellerstein for a number of useful discussions we had, and Mike Carey, Jeff Naughton, Yannis Ioannidis, Jignesh Patel and Manuvir Das for suggesting improvements to drafts of this paper.

References

[BMG93] José A. Blakeley, William J. McKenna, and Goetz Graefe. "Experiences Building the Open OODB Query Optimizer". In *Proceedings of the 1993 ACM-SIGMOD Conference*, Washington, DC, May 1993.

[FG91] Béatrice Finance and Georges Gardarin. "A Rule Based Query Rewriter in an Extensible DBMS". In *Proceedings of the 7th International Conference on Data Engineering.* IEEE, 1991.

[Fre87] Johann Christoph Freytag. "A Rule-Based View of Query Optimization". In *Proceedings of the 1987 ACM-SIGMOD Conference*, San Francisco, Californai, May 1987.

[GD87] G. Graefe and D. J. DeWitt. "The EXODUS Optimizer Generator". In *Proceedings of the 1987 ACM-SIGMOD Conference*, San Francisco, California, May 1987.

[GM93] G. Graefe and W. J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search". In *Proc. IEEE Conf. on Data Eng.*, Vienna, Austria, 1993.

[Gra87] Goetz Graefe. *"Rule-Based Query Optimization in Extensible Database Systems"*. PhD thesis, University of Wisconsin–Madison, November 1987.

[Hel94] Joseph M. Hellerstein. "Practical Predicate Placement". In *Proceedings of the 1994 ACM-SIGMOD Conference*, Minneapolis, Minnesota, May 1994.

[HP88] Waqar Hasan and Hamid Pirahesh. "Query Rewrite Optimization in Starburst". Research Report RJ 6367 (62349), IBM, 1988.

[IK90] Yannis E. Ioannidis and Younkyung Cha Kang. "Randomized Algorithms for Optimizing Large Join Queries". In *Proceedings of the 1990 ACM-SIGMOD Conference*, June 1990.

[IW87] Yannis E. Ioannidis and Eugene Wong. "Query Optimization by Simulated Annealing". In *Proceedings of the 1987 ACM-SIGMOD Conference*, San Francisco, California, June 1987.

[Kan91] Younkyung Cha Kang. "Randomized Algorithms for Query Optimization". Technical Report TR–1053, Computer Sciences Department, University of Wisconsin–Madison, 1991.

[KGM91] Tom Keller, Goetz Graefe, and David Maier. "Efficient Assembly of Complex Objects". In *Proceedings of the 1991 ACM-SIGMOD Conference*, Denver, Colorado, May 1991.

[KMP93] Alfons Kemper, Guido Moerkotte, and Klaus Peithner. "A Blackboard Architecture for Query Optimization in Object Bases". In *Proc. of the 19th VLDB Conf.*, 1993.

[LFL88] Mavis K. Lee, Johann Christoph Freytag, and Guy M. Lohman. "Implementing an Interpreter for Functional Rules in a Query Optimizer". In *Proc. of the 14th VLDB Conf.*, Los Angeles, California, 1988.

[LV91] Rosana S. G. Lanzelotte and Patrick Valduriez. "Extending the Search Strategy in a Query Optimizer". In *Proc. of the 17th VLDB Conf.*, Barcelona, September 1991.

[MDZ93] Gail Mitchell, Umeshwar Dayal, and Stanley B. Zdonik. "Control of an Extensible Query Optimizer: A Planning Based Approach". In *Proc. of the 19th VLDB Conf.*, Dublin, Ireland, 1993.

[OL90] K. Ono and G.M. Lohmann. "Extensible Enumeration of Feasible Joins for Relational Query Optimization". In *Proc. of the 16th VLDB Conf.*, August 1990.

[Pea84] Judea Pearl. *"Heuristics"*. Addison-Wesley Publishing Company, 1984.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. "Extensible/Rule Based Query Rewrite Optimization in Starburst". In *Proceedings of the 1992 ACM-SIGMOD Conference*, June 1992.

[Raa95] Francois Raab. *"TPC Benchmark D – Standard Specification, Revision 1.0"*. Transaction Processing Performance Council, May 1995.

[SAC$^+$79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. "Access Path Selection in a Relational Database Management System". In *Proc. SIGMOD*, May 1979.

[SC90] Eugene J. Shekita and Michael J. Carey. "A Performance Evaluation of Pointer-Based Joins". In *Proceedings of the 1990 ACM-SIGMOD Conference*, Atlantic City, New Jersey, May 1990.

[SG88] Arun Swami and Anoop Gupta. "Optimization of Large Join Queries". In *Proceedings of the 1988 ACM-SIGMOD Conference*, 1988.

[SJ90] Edward Sciore and John Seig Jr. "A Modular Query Optimizer Generator". In *Proc. IEEE Conf. on Data Engineering*, Los Angeles, California, February 1990.

[Swa89] Arun Swami. "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques". In *Proceedings of the 1989 ACM-SIGMOD Conference*, Portland, Oregon, June 1989.