The VLDB Journal (1999) 8: 55–78

The VLDB Journal
© Springer-Verlag 1999

OPT++ : an object-oriented implementation for extensible database query optimization

Navin Kabra, David J. DeWitt

Computer Sciences Department, University of Wisconsin–Madison, Madison, WI-53706, USA. e-mail: {navin, dewitt}@cs.wisc.edu

Edited by M. Carey. Received October 1996 / Accepted January 1998

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, and also to change the search strategy. 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 evaluating alternative optimization techniques and strategies to decide what techniques are best suited for that database system. We present the results of a series of performance studies. These results validate our design and show that, in spite of its flexibility, OPT++ can be used to build efficient optimizers.

Key words: Object-relational databases – Query optimization – Extensibility – Software architecture

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 make it easy to add new query algebra operators/algorithms 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 evaluate various heuristics that can limit the search space explored by the optimizer. The optimizer-implementor should also be able to explore different search strategies, and, if necessary, to mix multiple strategies in a single optimizer. Finally, this flexibility should be achieved without sacrificing performance – *i.e.*, an optimizer built in this extensible framework should not be much 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.

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.
Correspondence to: Navin Kabra

Furthermore, the search strategy itself is a class with virtual methods that can be overridden. 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.

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.

OPT++ is an easy-to-use, flexible and extensible toolkit for building database query optimizers. OPT++ comes equipped with many of the most common optimization techniques and search strategies, and can thus relieve the optimizer-implementor of the job of implementing them. Using OPT++, the optimizer-implementor can thus concentrate on tailoring it to the needs of the specific database system. Alternatively, the OPT++ architecture can be viewed as guidance to optimizer builders on how to structure their optimizer for extensibility. The modularity and clean program decomposition of the OPT++ architecture not only makes the whole optimizer easy to implement and understand, but also promotes sharing of code among different optimization schemes and implementations; leading, in turn, to improved maintainability.

OPT++ can also provide a smooth transition path for systems that already have a System-R style or rule-based optimizer, but which need to be upgraded. Initially, OPT++ could be used to implement exactly the same optimization scheme as the existing optimizer, using code from the old optimizer to implement the derived classes in the OPT++-based optimizer. Once this OPT++-based optimizer is working and stable, the optimizer-implementor can slowly start taking advantage of the other features of OPT++. This could be a more acceptable solution for an optimizer-implementor afraid of replacing a working optimizer with a completely new optimizer.

Although a number of the ideas incorporated in OPT++ are not new (see Sect. 4), OPT++ puts them all together into a clean architecture. It is easy to come up with a design for extensibility, but the difficulty lies in the details. Deciding upon how much abstraction is good is a difficult problem. There is a trade-off between making the abstract classes very general or very specific. Making the abstractions very general is great for extensibility. Since the abstract classes are very general, they can be extended to handle almost any kind of optimization algorithm. On the other hand, the abstractions have to be restrictive to allow for efficiency, and code re-use. Specifically, if the abstract classes are restrictive, the search strategy (which has to be written entirely in terms of these abstract classes) has more information available to it. Hence, it can use this information to implement algorithms and data structures that are more efficient than would have been possible without that information. Further, if an abstract class is too general, most of the code has to be written in the derived classes. Hence, the optimizer-implementor ends up doing a lot of unnecessary work to implement an optimizer. On the other hand, having restrictive abstract classes makes the system less extensible and might end up defeating the whole purpose of the "extensible" architecture. This paper makes a contribution by describing a detailed architecture that is extensible enough to be able to incorporate most of the major optimization techniques, and at the same time not sacrificing efficiency.

As described in the previous paragraph, the OPT++ architecture represents a compromise between extensibility and efficiency. The abstractions in OPT++ were made restrictive for the purposes of efficiency. Consequently, there are some special-purpose non-standard optimization algorithms that cannot be modelled using the OPT++ abstractions. Thus, the join order enumeration algorithms described in [GLPK94], [VM96] and [KBZ86] cannot be easily incorporated into OPT++. While some of the ideas and data structures of these algorithms can be incorporated into search strategies implemented in OPT++, the algorithms in their entirety cannot be incorporated in a reasonably extensible way. Hence, the use of OPT++ would preclude the use of such special-purpose algorithms. While it might be possible to build fast and efficient query optimizers for very specific database systems using some of these algorithms, it is unclear whether these algorithms can be extended to apply to domains other than the one they were originally intended for. Hence, even though the optimizers built using OPT++ might not be as efficient as these algorithms, from the point of view of extensibility, we do not see this as a shortcoming of the OPT++ architecture.

The remainder of this paper is organized as follows. Section 2 describes the design of our optimizer. Section 3 discusses our experiences using our optimizer framework, illustrating the ease of use as well as the efficiency of OPT++. We also compare the performance of various optimization search strategies and various optimization heuristics in terms of optimization time taken and the improvement in estimated cost of the optimal plan. These results are also presented in Sect. 3. Related work is presented in Sect. 4. In Sect. 5, we present our conclusions.

2 OPT++ system design

2.1 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, Fig. 1a shows an SQL query and Fig. 1b 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

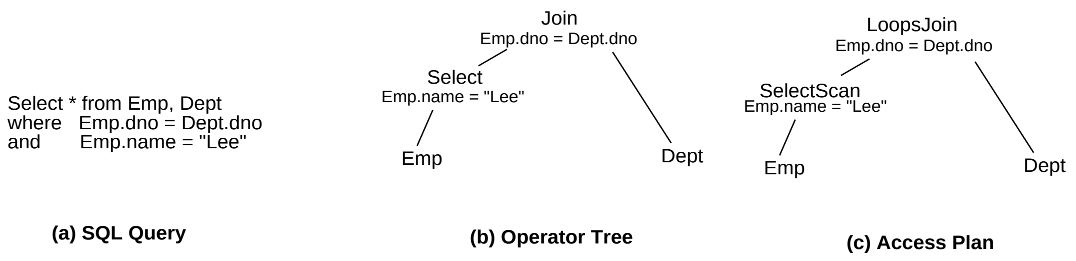

Fig. 1a–c. Query representations

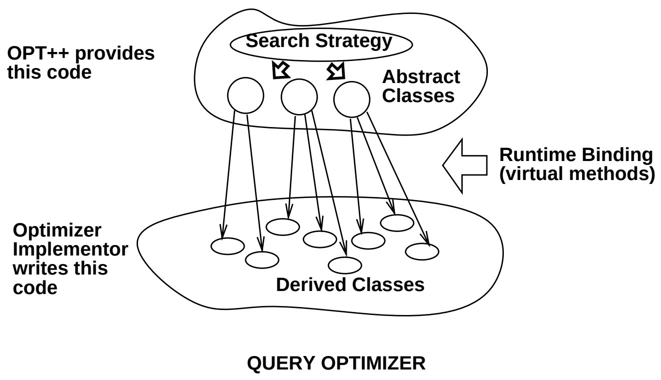

Fig. 2. Basic system design

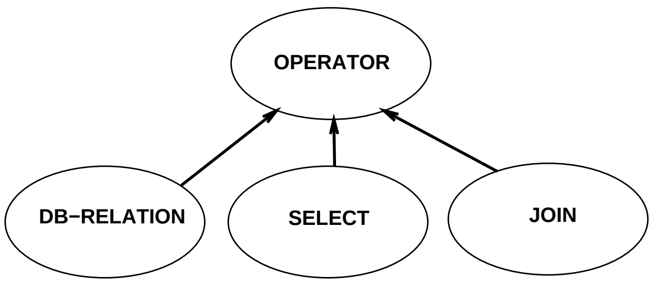

Fig. 3. Operator class hierarchy for a relational optimizer

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 1c shows one possible access plan corresponding to the operator tree in Fig. 1b. Each operator tree will, in general, have a number of corresponding access plans.

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.

2.2 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.

2.2.1 The `OPERATOR` class

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 Fig. 3[1]. The `SELECT` and `JOIN` classes represent the relational select and the relational join operators, respectively. The `DB-RELATION` operator is explained in the next paragraph. In this SQL optimizer, an instance of the `SELECT` operator will represent an application of the relational select operator to one input relation, and an instance of the `JOIN` operator will represent an application of the relational 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).

Dummy operators serve as leaf nodes of the operator tree, representing 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 `DB-RELATION` operator in all our examples.

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

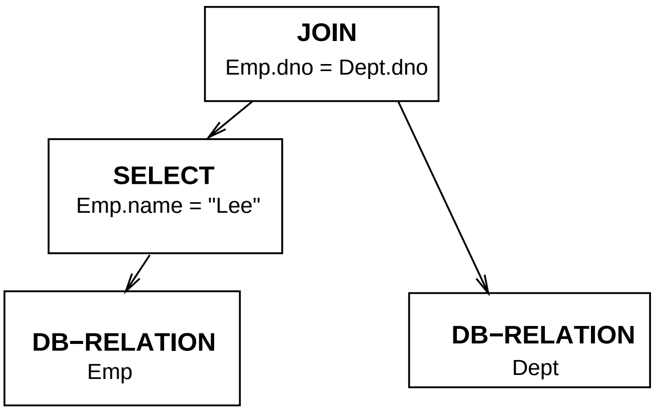

Fig. 4. An example operator tree

Figure 4 shows an example of an operator tree[2] corresponding to the query shown in Fig. **??**. The two instances of the DB-RELATION class represent the two relations in the from clause of the query – Emp and Dept. The instance of the SELECT class represents a selection on the Emp relation, and the instance of the JOIN class represents the Dept relation being joined to the result of the selection.

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-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 Fig. 4 showing the TREEDESCRIPTOR instances associated with each operator instance. In this example, each TREEDESCRIPTOR instance lists the names of the relations that have been joined in and the predicates applied.

With the TREEDESCRIPTOR class the optimizer-implementor must 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 TREEDESCRIPTOR must also have an IsCompleteQuery method that determines whether the corresponding operator tree represents the whole query or just a sub-computation. Finally, the TREEDESCRIPTOR class must also have a HASHVALUE method that returns a hash value for the TREEDESCRIPTOR. This method can be used

[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.

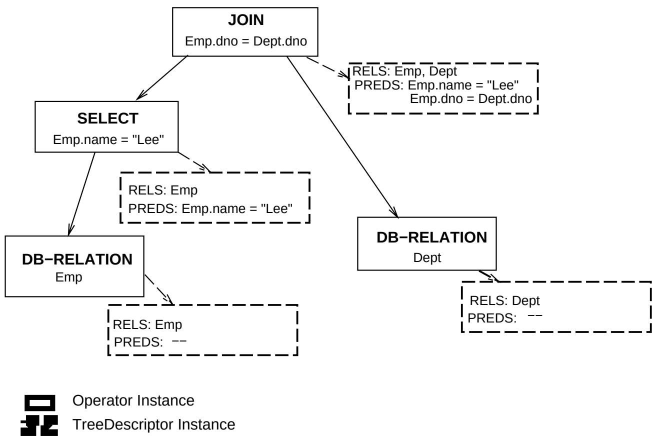

Fig. 5. An example operator tree with its tree descriptors

by OPT++ to build hash tables of TREEDESCRIPTORs, and to efficiently search for trees with equivalent TREEDESCRIPTORs.

The OPERATOR class includes a virtual method called DERIVETREEDESCRIPTOR. This method 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 input operators.

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 require any information about the actual class of each instance. Through runtime binding, the proper DERIVETREEDESCRIPTOR 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. The IsCompleteQuery, IsEquivalent and the CANBEAPPLIED methods can be used to analyze the generated operator trees.

2.2.2 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.

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

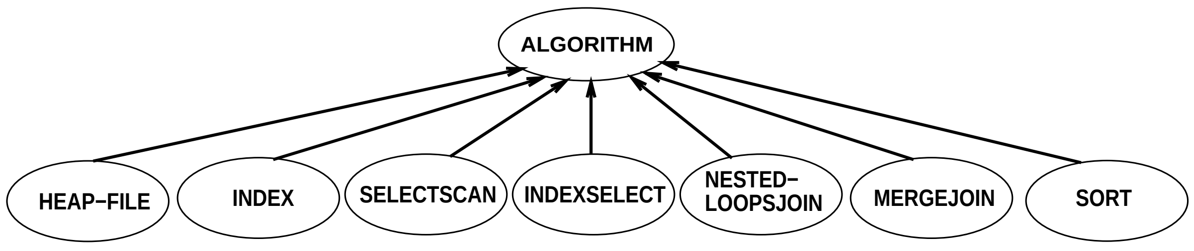

Fig. 6. Algorithm class hierarchy for a relational optimizer1.3in4in

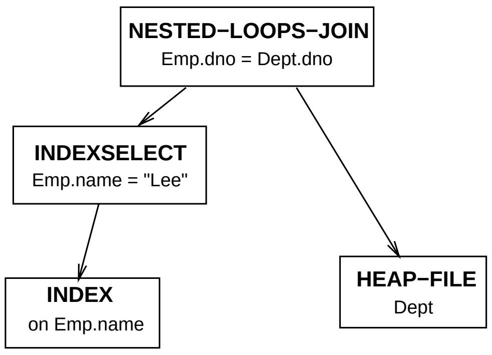

Fig. 7. An example access plan

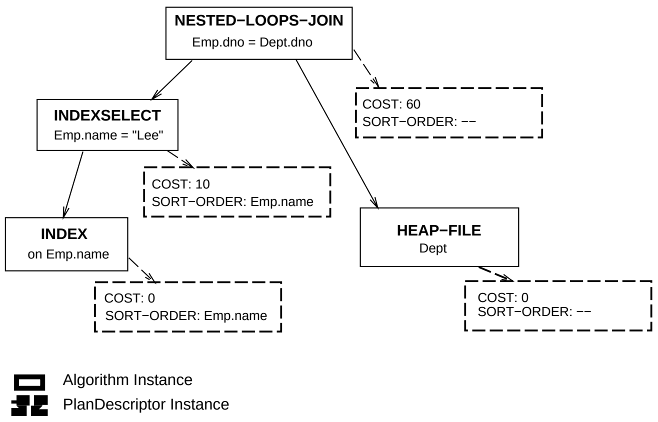

Fig. 8. An example access plan with its plan descriptors3.5in

and `INDEX` dummy algorithm classes to represent these cases in our examples. Note that these algorithm classes are associated with the dummy `DB-RELATION` operator class defined in the previous section.

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 `DB-RELATION` operator as explained earlier. The `SELECTSCAN` algorithm used to implement the `SELECT` operator represents a sequential scan of a `HEAP-FILE` that outputs tuples satisfying a select-predicate. The `INDEXSELECT` 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.

Figure 7 shows an example access plan. This is an access plan corresponding to the operator tree in Fig. 4[3]. 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++ employs a `PLANDESCRIPTOR` class to store the physical properties associated with 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 Fig. 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[4]. Similar to the `TREEDESCRIPTOR`, the `PLANDESCRIPTOR` must also provide a `HASHVALUE` method that can be used for efficient hashing.

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 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 out-

[3] In this section, we are only concerned with being able to represent an access plan. How OPT++ translates an operator tree into an access plan is described in Sect. 2.3.

[4] 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].

puts that are required for subsequent query-processing algorithms. These are referred to as *enforcers* by 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.

2.3 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.

2.3.1 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 produce a larger 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 an operator to it and generating a new tree is accomplished by using one of the TREETOTREEGENERATOR classes.

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 as shown in Fig. 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

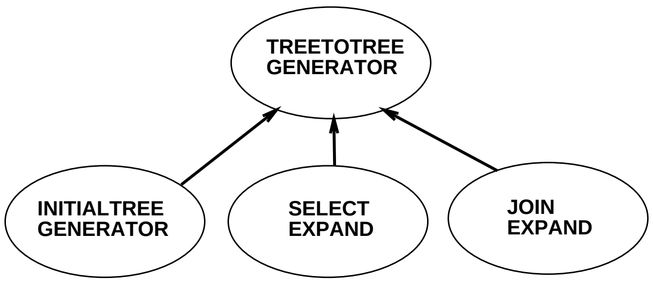

Fig. 9. Example TREETOTREEGENERATOR class hierarchy

JOIN instances representing different ways of applying a join to the given input.

Figure 10b illustrates the JOINEXPAND::APPLY method being invoked during the optimization of the query in Fig. 10a. 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 with either the Dept relation or 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.

There is also a method called APPLYMULTIPLETIMES that can be used to determine whether a particular TREETOTREEGENERATOR (such as SELECTEXPAND) can be applied a second time to an operator tree that resulted from the application of the same TREETOTREEGENERATOR class. This is useful for avoiding redundant work. For example, consider an optimizer in which all the select predicates are pushed down as far as possible and they are all applied by a single select operator. In this case, when the SELECTEXPAND generator is invoked for a given relation, it produces a single select operator to apply all the select predicates that apply to that relation. There is no need to apply the SELECTEXPAND generator again to this new select operator as there will not be any new select predicates to apply (until at least another relation is joined in). Thus, the SELECTEXPAND should return FALSE when APPLYMULTIPLETIMES is invoked. By contrast, the JOINEXPAND should return TRUE, because you can keep applying joins until there are no more relations left.

The APPLYMULTIPLETIMES method simply indicates that a particular TREETOTREEGENERATOR should not be applied twice in a row. The same functionality can also be achieved by appropriately coding the CANBEAPPLIED method. However, using APPLYMULTIPLETIMES is more efficient.

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 the optimization process. For the relational optimizer, the INITIALTREEGENERATOR creates one DB-RELATION instance for each relation in the from clause. After that, the search strat-

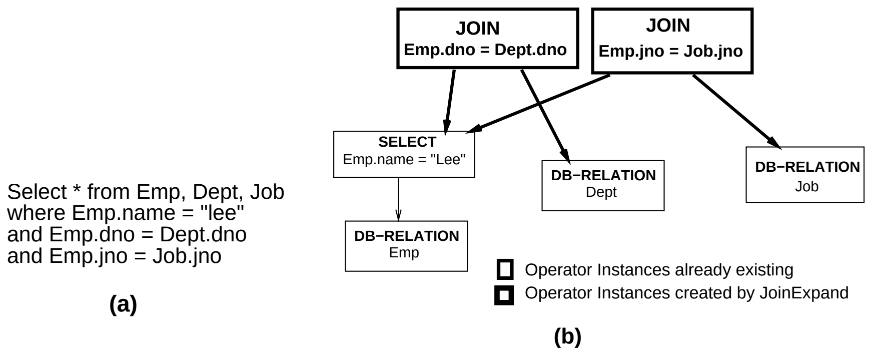

Fig. 10a,b. Application of JOINEXPAND::APPLY

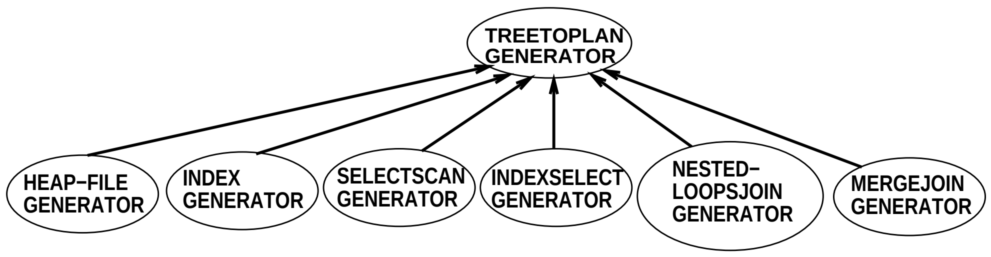

Fig. 11. Example TREETOPLANGENERATOR class hierarchy

egy 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.

2.3.2 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.

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

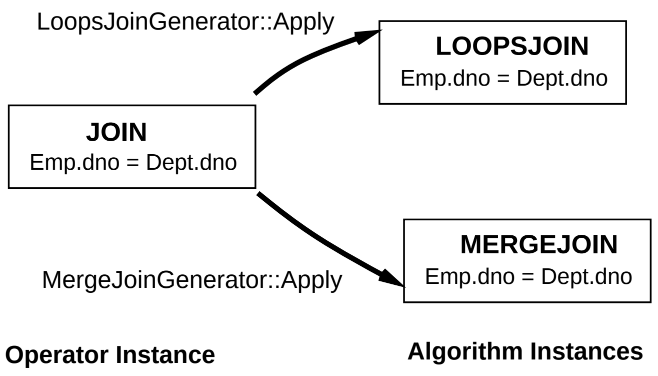

Fig. 12. Examples of TREETOPLANGENERATOR::APPLY

that operation. Figure 11 shows the classes derived from the TREETOPLANGENERATOR class.

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 TREETOPLANGENERA-

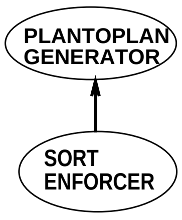

Fig. 13. PLANTOPLANGENERATOR class hierarchy

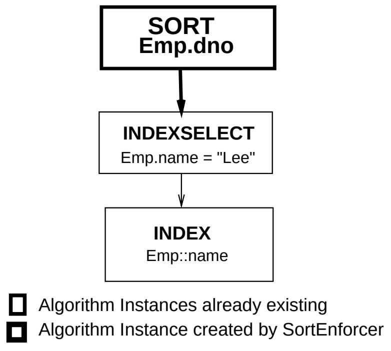
SORT
Emp.dno
INDEXSELECT
Emp.name = "Lee"
INDEX
Emp::name
Algorithm Instances already existing
Algorithm Instance created by SortEnforcer

Fig. 14. Use of SORTENFORCER::APPLY to enforce a sort order

TOR 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.

2.3.3 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 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 (enforcer) as shown in the figure.

The PLANTOPLANGENERATOR class also has a CAN-BEAPPLIED 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 TREETO-PLANGENERATOR classes, it invokes the INPUTCONSTR-AINT method whenever a new algorithm instance is created. If it turns out that the inputs of that algorithm instance do not satisfy its input constraints, it attempts to rectify the situation by applying an appropriate PLANTOPLANGENERA-TOR. 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 automatically get applied without the optimizer-implementor having to worry about them.

2.4 The search strategies

Thus 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 SEA-RCHSTRATEGY 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. See [LV91] to see how this is accomplished. In this section, we concentrate on describing how the various search strategies are implemented in terms of the OPERAT-OR, 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.

In the section below, we describe the various search strategies that have been implemented in OPT++ so far. 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.

2.4.1 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 TREETO-TREEGENERATORs can be applied to the operator instance at its root by exhaustively invoking the CANBEAPPLIED method of all the 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[5].

Generation of new operator trees stops when none of the operator trees can be further *expanded*. At this point, optimization is complete after all the applicable TREETOPLANGENERATORs and PLANTOPLANGENERATORs are applied to the existing operator trees. The cheapest access plan that represents the complete input query can now be returned as the optimal plan. The IsCompleteQuery method is used to determine whether or not an access plan represents the complete input query. To be able to implement the IsCompleteQuery method, the optimizer-implementor must have access to some internal representation of the original input query. This must then be compared with the TREEDESCRIPTOR and PLANDESCRIPTOR associated with a given access plan to determine whether it represents the complete query. The optimizer-implementor is responsible for implementation of the IsCompleteQuery method, and providing it with access to some efficient internal representation of the input query.

2.4.2 The transformative search strategy

In Sect. 2.3.1, we have only given examples of TREETOTREEGENERATORs that *expand* a given tree by applying a new operator to it. However, an optimizer constructed using OPT++ can also include TREETOTREEGENERATOR classes that transform one 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 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 search strategy keeps track of which TREETOTREEGENERATORs were used to generate each operator instance. This is useful in reducing the amount of redundant work done by the algorithm. First, if the APPLYMULTIPLETIMES method for a TREETOTREEGENERATOR returns FALSE, then this generator is not applied to a given operator instance if that operator instance was generated using the same generator. For example, two applications of the JOINCOMMUTATIVITY generator would result in the same tree as the original, and hence the APPLYMULTIPLETIMES method of this generator should return FALSE. Also, whenever a new operator instance is generated by a TREETOTREEGENERATOR, the search strategy finds out whether another operator instance which is exactly equivalent to it exists. If it does, the new instance is pruned. This ensures that applications of TREETOTREEGENERATORs do not lead to cycles.

Unfortunately, due to the generality of the OPT++ design, it cannot do as good a job of identifying equivalence classes as the Volcano Optimizer Generator. For this, it has to rely upon the IsEquivalent method provided by the optimizer-implementor. This is a shortcoming of OPT++ compared to the Volcano approach. Implementing the IsEquivalent method can be difficult for a general algebra. All database systems that have a System-R-style optimizer are faced with the problem of implementing such an operation. In practice, this has not been a problem for most declarative query languages.

In spite of the above limitation, the transformative search strategy of OPT++ can still capture the equivalence classes of Volcano. After using the IsEquivalent method to find equivalent operator instances, it puts them in the same equivalence class. Now, if two operator instances are exactly the same, and they only differ in the fact that their inputs point to different operators in the same equivalence class, then one of these operator instances can be pruned. This can be done because all the combinations of inputs can be easily generated by enumerating the various operators instances in an equivalence class. This allows OPT++ to get the same space efficiency as Volcano. Of course, to be able to use this trick successfully, the search strategy should be able generate the various fragments of operator trees necessary

[5] 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.

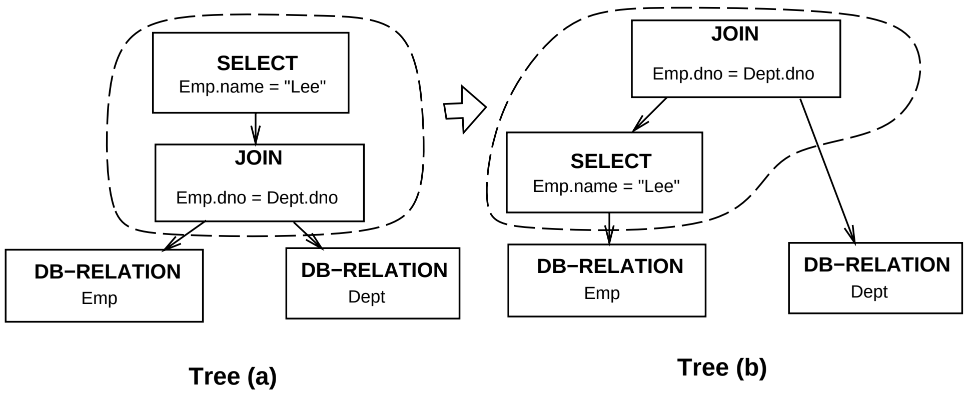

Fig. 15a,b. A rule-based transformation

for applying transformations. The search strategy does this "instantiation" of equivalence classes on demand. Whenever the CANBEAPPLIED or the APPLY method of an operator tries to examine one of its inputs by invoking the OPERATOR::INPUT method, the search strategy instantiates it with one of the operator instances in the corresponding equivalence class. On the next invocation of CANBEAPPLIED or APPLY, the OPERATOR::INPUTmethod returns the next operator instance from the same equivalence class. The search strategy continues this process until all the operator instances in the equivalence classes of the inputs are exhausted.

We note that the "instantiation" of equivalence classes is done on demand: *i.e.*, only those equivalence classes that are actually examined by the CANBEAPPLIED or the APPLY method are instantiated by the search strategy. In spite of that, this results in unnecessary instantiations. For example, a SELECTPUSHDOWN transformation that pushes a select below a join operator only needs to be instantiated with the JOINOPERATOR instances in its input. Since the search strategy does not know this, it instantiates the input with all possible operator instances. If the exact structure of the sub-tree that is required by a TREETOTREEGENERATOR is known beforehand, we can avoid this inefficiency. In such a case, the TREETOTREEGENERATOR can register itself with the search strategy at system startup time by specifying a sub-tree expression consisting of operator names. The search strategy then ensures that only sub-trees matching the specified expression are instantiated. The above discussion applies to TREETOPLANGENERATORs and PLANTOPLANGENERATORs as well. We note that in the Volcano Optimizer Generator, such a tree expression is always specified (in the transformation and implementation rules), whereas in OPT++ it is needed only for efficiency.

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].

2.4.3 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 TREETOTREEGENERATOR 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 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.

2.5 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 3 has some examples of such extensions as applied to a real optimizer.

2.5.1 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

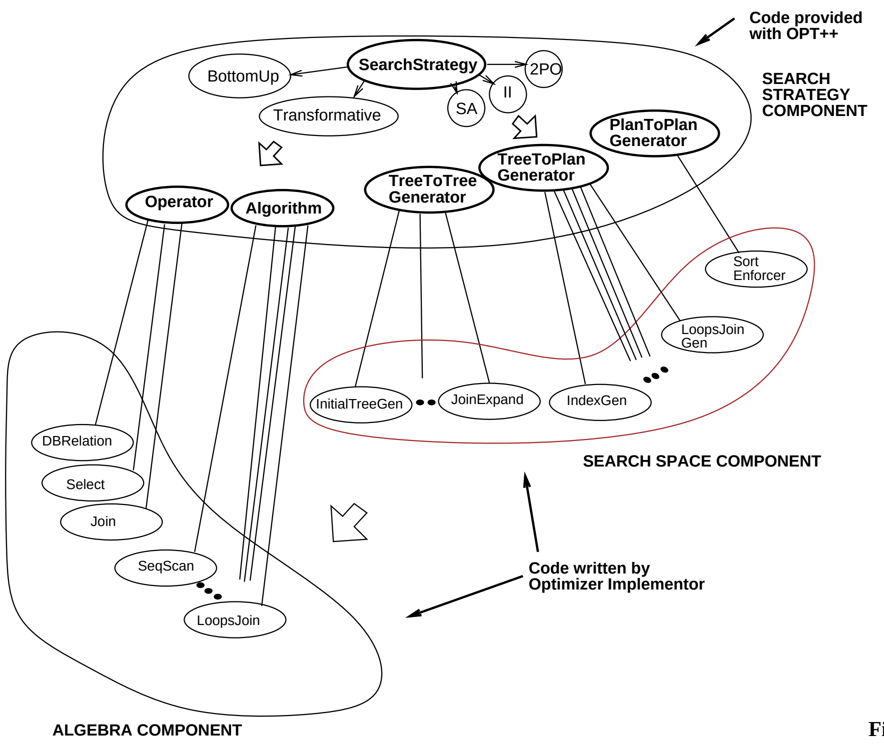

Fig. 16. Implementing an Optimizer in OPT++

to implement a particular optimizer. Thus, a large fraction of the code required for an optimizer is already provided with OPT++.

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).

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 BUSHYJOINEXPAND class that considers composite inners will generate all bushy trees.

We note that the implementation of some of the TREETOTREEGENERATOR classes can be made more efficient if they make specific assumptions about the semantics of a particular search strategy, or if they directly access the data structures of the search strategy class. In such a case, that TREETOTREEGENERATOR becomes specific to that particular search strategy, and cannot be re-used with any other search strategy. Hence, for example, we have two implementations of the BUSHYJOINEXPAND generator: one that does not assume anything about the search strategy, and one that uses the data structures of the bottom-up search strategy to efficiently organize and retrieve operator trees with a specific number of join operators. Thus, although some efficiency is lost due to the abstractions of OPT++, a specific implementation may still override the abstractions and achieve efficiency (at the cost of extensibility). In fact, the various join enumeration algorithms described in [OL90] can each be implemented in OPT++ as a class derived from the TREETOTREEGENERATOR class.

2.5.2 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 ALGORITHM 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. The existing code usually does not need 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 must be added as described above.

Sometimes, it is possible that adding a new operator or algorithm might require that the TREEDESCRIPTORs or PLANDESCRIPTORs need to store additional information. For example, when MERGEJOIN is added to the system, information about whether the output of a particular algorithm is sorted or not needs to be added. In this case, the DERIVETREEDESCRIPTOR, or the DERIVEPLANDESCRIPTOR methods of all the operators or algorithms might have to be changed to reflect this new property. This admittedly goes against the OPT++ philosophy, and is a shortcoming. However, we believe this cannot be avoided without compromising the efficiency of OPT++. Further, these changes are localized to only the DERIVETREEDESCRIPTOR or DERIVEPLANDESCRIPTOR methods.

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 of OPT++, and all the code that depends only on the query algebra is in the algebra component, the search space component is only a small amount of code. Thus, changing generator code or adding a new generator is easy.

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*. Unfortunately, this is not always true. 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 represent the transformation rules. However, since there is very little code in the classes derived from the TREETOTREEGENERATOR classes, this change is rather easy. Further, note that only the TREETOTREEGENERATOR classes need to be rewritten. All the code in the "Algebra" component, the TREETOPLANGENERATORs, and the PLANTOPLANGENERATORs remain unchanged. Hence, although this change in search strategy does require some new code to be added, a lot of old code can be re-used. We describe a specific example in Sect. 3.

3 Experiences with OPT++

In this section, we describe various optimizers that we have constructed 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 several 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.

We also performed a study of the various search strategies and optimization techniques that have been implemented in OPT++ to study their relative effectiveness in the presence of the object-relational operators.

The purpose of this section is twofold. First, it gives an idea of the kind of optimizers and optimization techniques that can be implemented using OPT++. This speaks for the extensibility and flexibility of OPT++. Second, it illustrates the kind of experimentation that can easily be done by an optimizer implementor when trying to evaluate different optimization techniques. Studies like this would be decisive in fast prototyping of new optimizers and optimization techniques, and for exploring new ideas.

3.1 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, as well as cartesian products. The purpose of this section is to just show the baseline case (a relational optimizer that can do different kinds of join enumerations). In the later sections, we extend the base optimizer to handle more complex cases.

Since all the examples used in Sect. 2 describe this simple relational optimizer, we will not repeat the details here. Briefly, the DB-RELATION, 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 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).

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 TREETOTREEGENERAT-

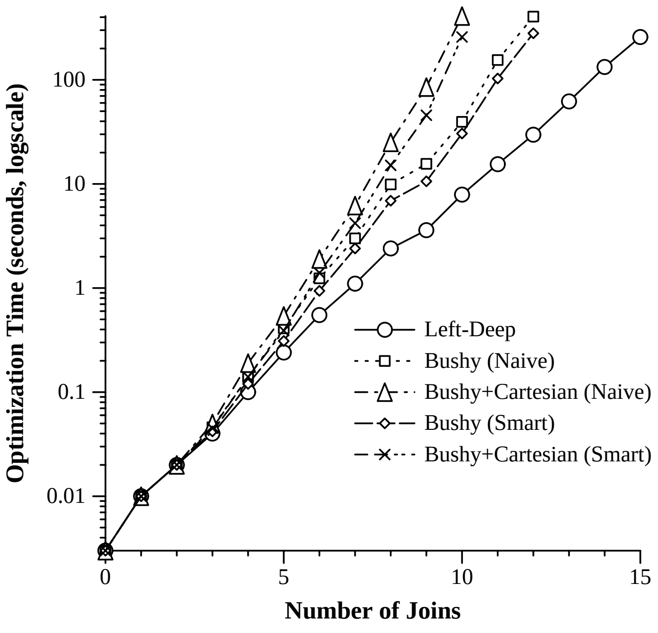

Fig. 17. Comparison of search spaces: optimization times (log-scale)

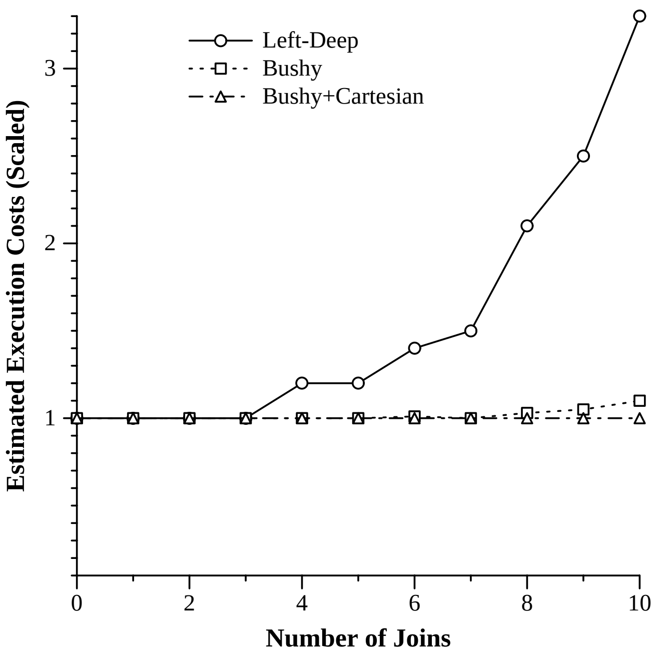

Fig. 18. Comparison of search spaces: estimated costs (scaled)

OR, 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 spread across 10 classes) makes it very easy to evaluate various optimization techniques.

We decided to modify the search space explored to include both bushy join trees and join trees that contain cartesian products. As described in Sect. 2.5.1, these enumerators can be implemented in two ways. A naive implementation that makes no assumptions about the underlying search strategy results in code that is more re-usable but less efficient. To do this, we derived the NAIVEBUSHYJOINENUMERATOR and NAIVECARTESIANJOINENUMERATOR 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. A smarter implementation (built with access to the internal data structures of the System-R-dynamic-programming-style search strategy that was used) was also coded to give better performance. This resulted in the SMARTBUSHYJOINENUMERATOR and SMARTCARTESIANJOINENUMERATOR classes which are based on the schemes described in [OL90].

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 200 MHz Pentium Pro processor with 128 MB of memory. However, virtual memory was limited to 32 MB (using the *limit* command).

Figure 17 illustrates the effect of different search spaces on the optimization time. Figure 18 shows the effect on the relative estimated execution costs of the optimal plans produced[6]. (Note that optimization times are shown on a logarithmic scale.)

3.2 A more complex query algebra

In this section, we describe how the base 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.

[6] These numbers just confirm the results of [OL90]

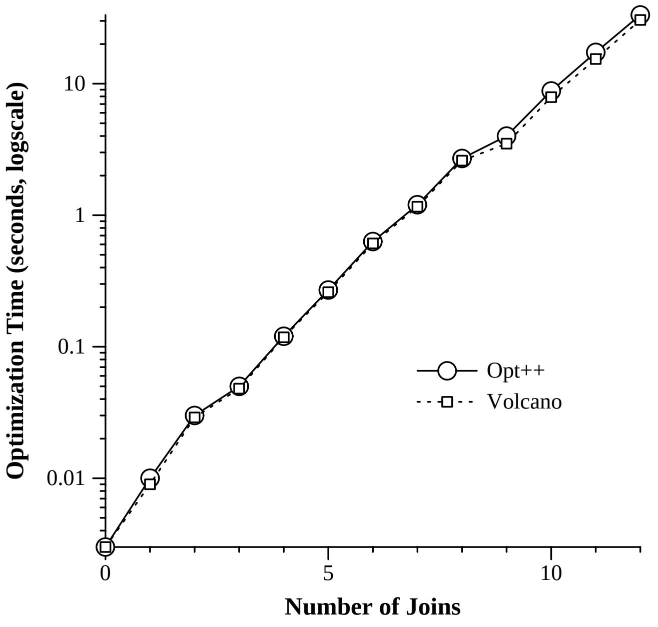

Fig. 19. OPT++ *vs.* Volcano: optimization times (log-scale)

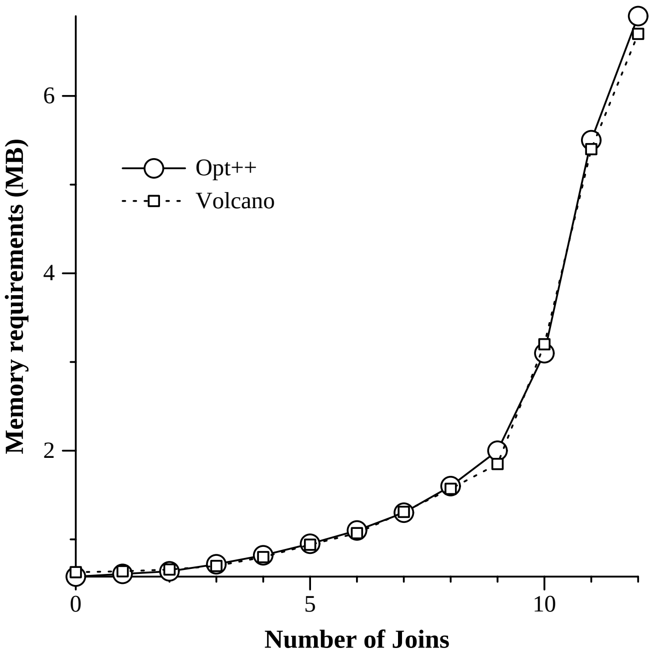

Fig. 20. OPT++ *vs.* Volcano: memory requirements

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 sometimes be evaluated using a path index without really having to materialize the individual components of the path expression. For example, if there exists a path index on city.mayor.name, then 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[7].

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.

3.3 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[8]. 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 PLANTOPLANGENERATOR 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, 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 200 MHz Pentium Pro processor with memory limited to 32 MB. The figures show us that the performance

[7] 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 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].

[8] 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.

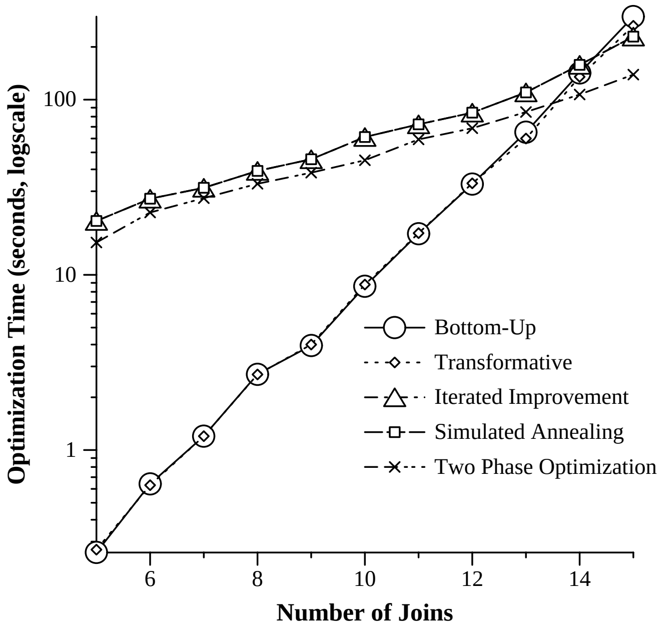

Fig. 21. Comparing search strategies: optimization times (log-scale)

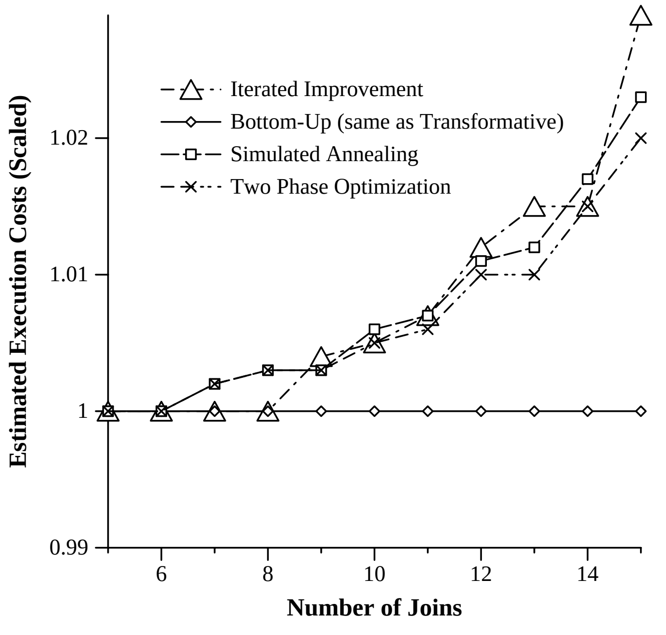

Fig. 22. Comparing search strategies: estimated costs (scaled)

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.

3.4 Randomized strategies

Finally, we modified the transformative optimizer to use the randomized search strategies available with OPT++. To do this, we replaced the Transformative Search Strategy object by an object of the required randomized search strategy. Thus, switching from a transformative search strategy to either Simulated Annealing, Iterated Improvement or Two-

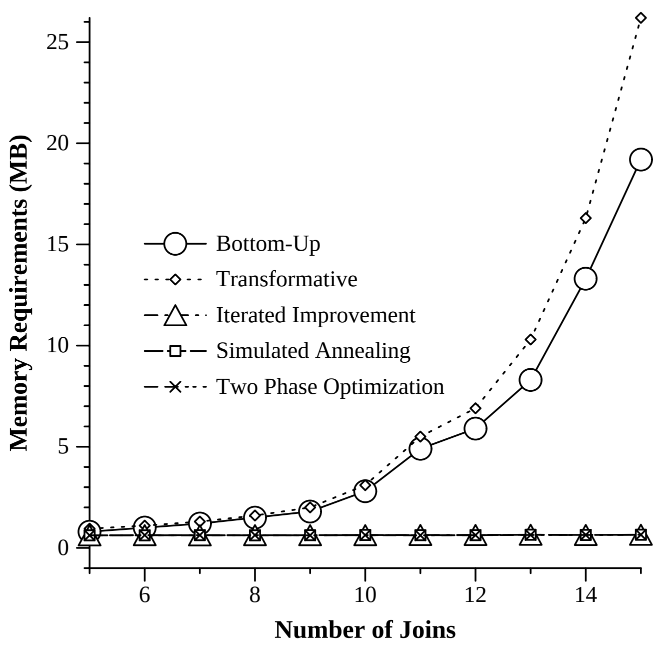

Fig. 23. Comparing search strategies: memory requirements

Phase Optimization (or *vice versa*) can be trivially accomplished by changing one line of code.

We compared the performance of these search strategies with each other and with the dynamic-programming-based search strategies. This is one illustration of the kind of experiments that can be very easily conducted using OPT++. This section also serves as a validation of our implementation of these search strategies in OPT++, as we obtain results similar to those found in the literature.

3.5 Comparison of search strategies

We compared the performance of each of 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, 22 show the performance results obtained. Since the bottom-up and the transformative strategies produce exactly the same plans, Fig. 22 shows only one curve for both of them. 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 either Simulated Annealing or Iterated Improvement.

In Fig. 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 Fig. 21, the randomized strategies

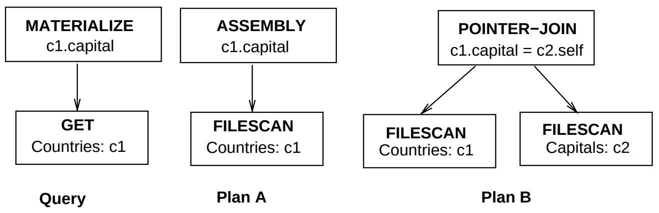

Fig. 24. Converting materializes to joins

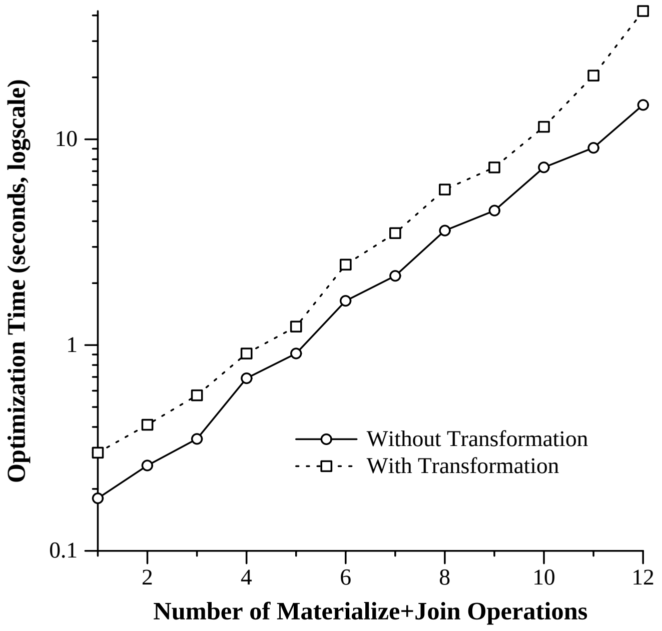

Fig. 25. Converting materializes to joins: optimization times (log-scale)

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 terms of space consumption as it can perform more aggressive pruning of operator trees.

3.5.1 Comparing optimization techniques

In this section, we describe and evaluate a variety of different optimization techniques. Using randomly generated queries of varying sizes, we compared the optimization time required for the optimizer with the feature turned "on" to that of the optimizer with the feature turned "off". We report a summary of the results in this section. Each feature was evaluated with all the remaining features turned on.

The reason for describing the features and their performance in the section are (1) to give an example of the kind of optimizations OPT++ is capable of handling, and (2) to study the effect each feature has upon the speed of the search engine. We have also reported the estimated costs of the resulting access plans to provide an idea about the trade-offs involved.

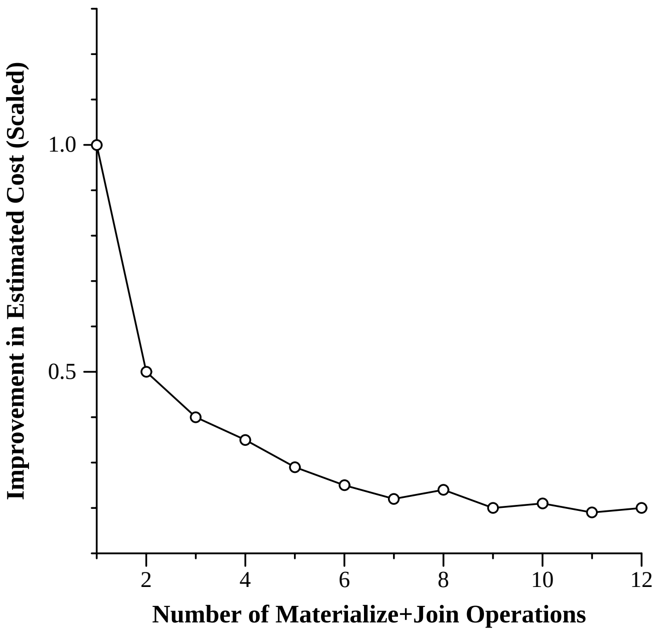

Fig. 26. Converting materializes to joins: improvement in estimated costs (scaled)

Convert materialize to join. Instead of the materialize operator, a pointer-based join [SC90] can be used to "follow" inter-object references. Figure 24 illustrates the use of this feature. Plan A gives an example of a plan that can be generated when this feature is turned "off", and plan B show an example of a plan that can be generated when this feature is turned "on"[9]. (This convention will be used in the rest of the examples in this section.) Note that the `self` method on any object returns the OID of that object. Hence, the pointer-join on `c1.capital() = c2.self()` is equivalent to materializing the `c1.capital()` method.

Figure 25 shows the effect on the optimization time for various randomly generated queries. The number of materialize operations in the query was varied, while the number of select predicates was kept constant at 4 (there were no explicit joins in these queries). Figure 26 plots the ratios of the estimated costs of the generated optimal plans, and thus shows the improvement in the estimated cost when the feature is turned "on". Due to the increase in number of alternative plans to be considered, the optimization time increases significantly (about 50% when there are eight materialize operators in the query) when this transformation was turned "on". On the other hand, the generated optimal plans were much cheaper (in terms of estimated cost).

Overall, this experiment seems to indicate that, although there is an increase in optimization cost involved in considering "pointer-join" as a possible method for computation of the materialize operator, there are large benefits in terms of reduction of execution cost. Hence, this is a useful optimization technique to implement for a query algebra that allows it.

[9] This does not mean that plan A will necessarily be rejected in favor of plan B. Plan B will be considered and then accepted or rejected based on the cost estimates.

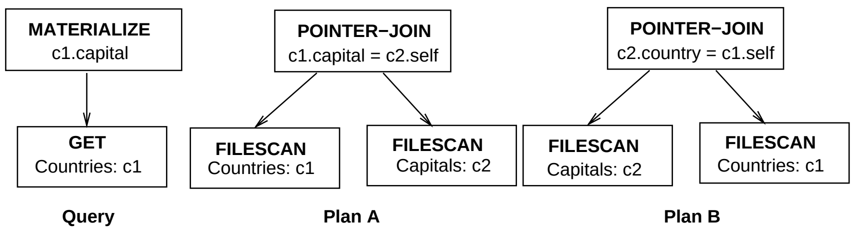

Fig. 27. Use of inverse links

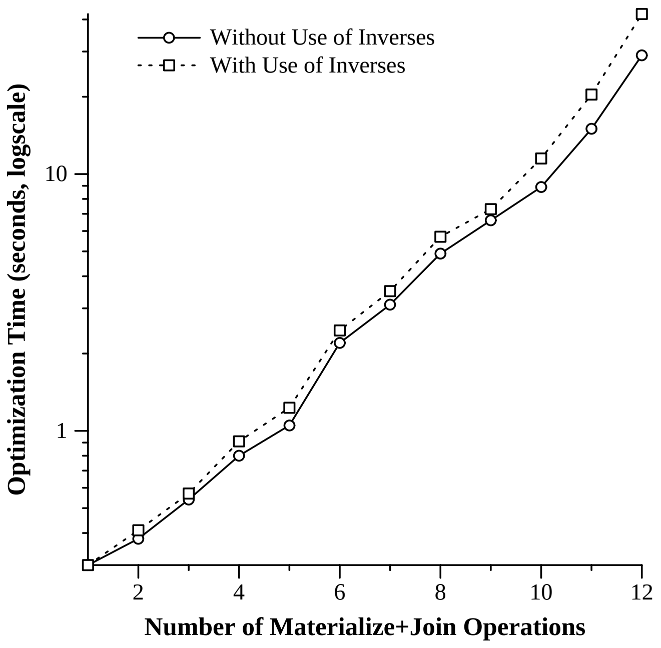

Fig. 28. Use of inverse links: optimization times (log-scale)

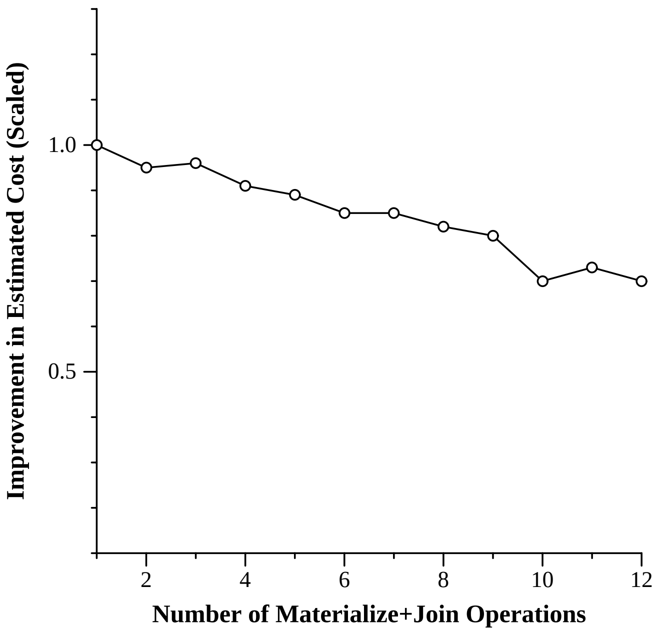

Fig. 29. Use of inverse links: improvement in estimated costs (scaled)

Use of inverse links. A join that uses an inter-object reference method to join its inputs can be converted to a join that uses the inverse of that method, if one exists. Figure 27 illustrates the use of this optimization. (We assume that the `Capital.country()` method is the inverse of the `Country.capital()` method. Figures 28, 29 show the effect on performance. In the random queries generated for this experiment, any method that was referred to had a 10% chance of having an inverse. Use of inverse links causes a 10–20% increase in the optimization time for queries that contain methods that have inverses. The estimated execution cost of the optimal plans shows a 10–30% improvement.

In the previous experiment, for any reference in the input query, there was only a 10% chance of the existence of a corresponding inverse link. Obviously, this figure affects the performance that we see. We repeated this experiment with a setting in which *all* the references in the input queries had inverse links. Figures 30, 31 show the new performance. We see that now the increase in optimization time is higher (30–35%). The estimated execution cost of the optimal plans shows a much higher (up to 50%) improvement.

Collapse multiple materializes. A string of materialize operator applications can be collapsed into a single materialize operator application. Figures 32, 33, 34 show the use and performance of this feature. The number of materialize operations in the randomly generated queries was varied, while the number of select predicates was kept constant at four (there were no explicit joins in these queries). An increase in optimization time of about 20–30% was observed. This increase can be directly attributed to the increase in the number of alternative operator trees that have to be considered. We did not observe any significant improvement in the estimated execution costs for this setup.

For the previous experiment, the optimization of converting materialize operators to joins was turned on. To see how that affected the results, we repeated the same experiment with this optimization turned off. (This involved commenting out one line of code in the optimizer.) Figures 35, 36 show the results of the new experiment. We see that with this setup the use of the complex assembly operator does give significant (about 30%) improvements in the estimated execution cost of the query.

Thus, this experiment indicates that, for the cost model we used, considering the complex assembly operator is a considerable improvement over naive materialization, but does not help very much if pointer-joins can be used.

Path indices. A select-materialize-filescan sequence might be collapsed into a single index scan with a predicate if a path index exists on the path expression in the select predicate[10]. Figure 37 shows how this can be useful. Note that the path index scan shown in plan B does not retrieve any `mayor` objects from the disk. Thus, if there were a select

[10] There can be more than one materialize operation between the select and the filescan.

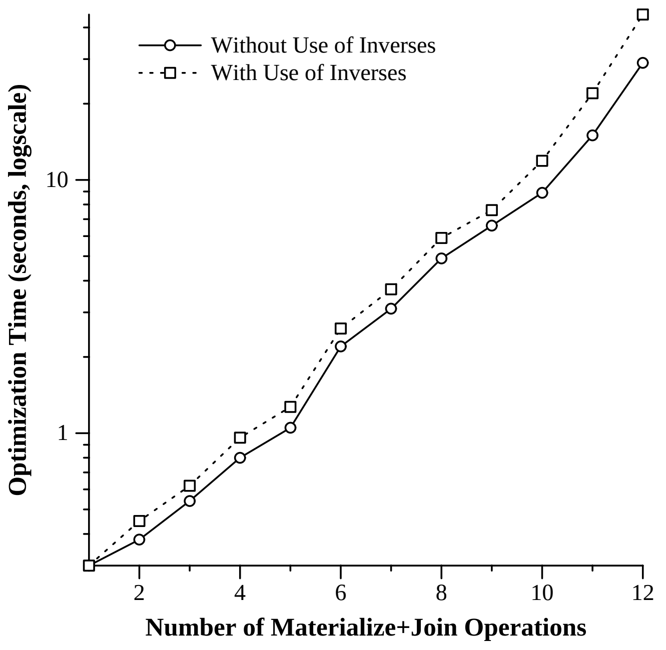

Fig. 30. Use of inverse links when all references have inverses: optimization times (log-scale)

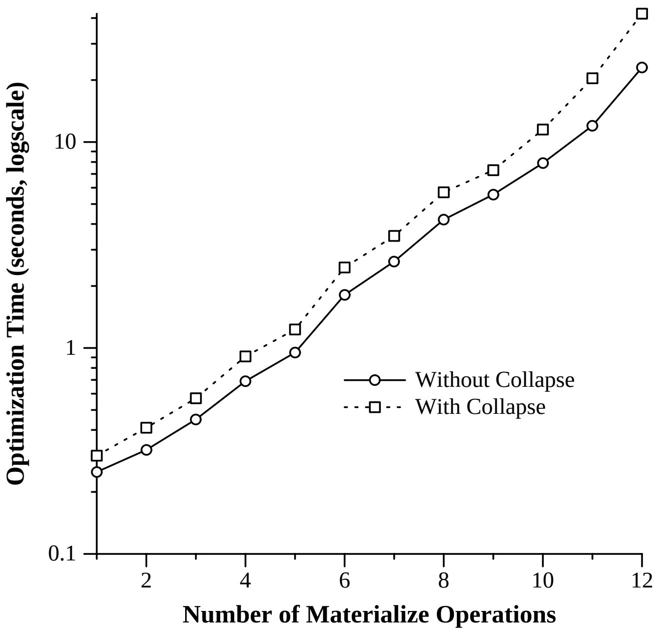

Fig. 33. Collapsing materializes: optimization times (log-scale)

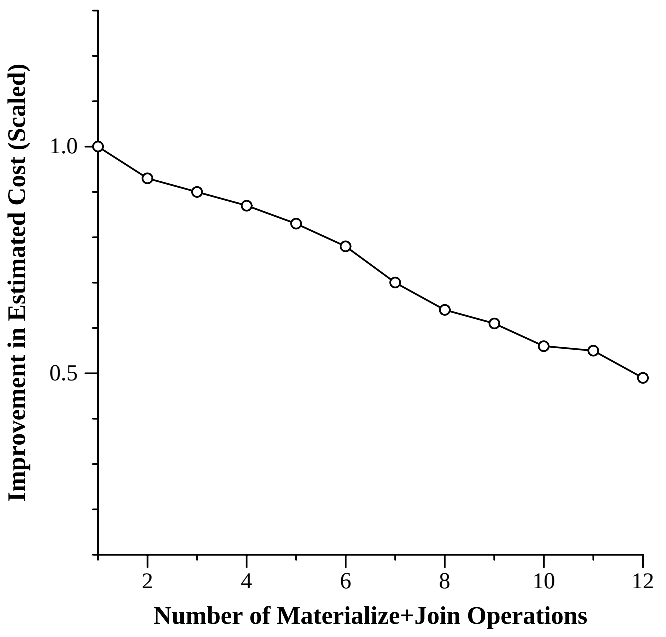

Fig. 31. User of inverse links when all references have inverses: improvement in estimated costs (scaled)

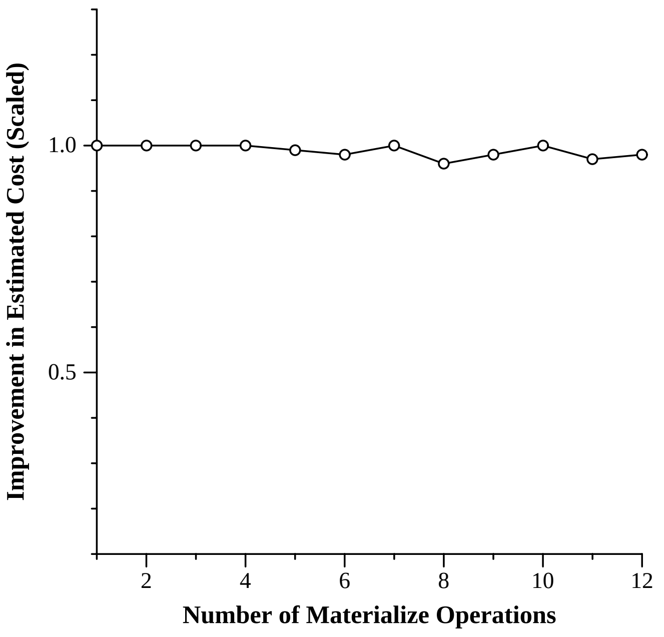

Fig. 34. Collapsing materializes: improvement in estimated costs (scaled)

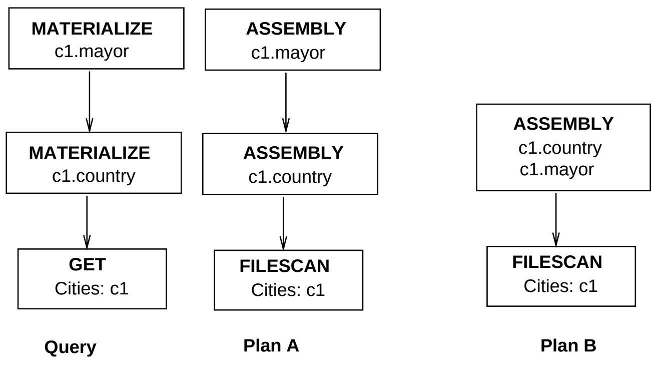

Fig. 32. Collapsing materializes

predicate on the mayor object, then the mayor objects would actually have to be materialized from disk. An *assembly enforcer* is required to make this work. See [BMG93] for a detailed discussion of this issue. We have incorporated in our optimizer the assembly enforcer as described in [BMG93]. We conducted experiments to study the effect of path indexes upon optimization time and the estimated execution cost. In these experiments, there was a 20% chance of a path index being available for evaluating any given predicate. The selectivity of these predicates varied uniformly from 0 to 100%. Figures 38, 39 indicate that, while the effect of this feature on the optimization time is negligible (less than 5%), its use can significantly reduce the estimated cost of the optimal plan.

If a suitable path index exists, then the improvement in execution cost is often very large. On the other hand, if there is no such index, then the improvement is zero. Also, the amount of improvement depends upon the selectivity of the selection predicate involving the use of the path index. Due to this, there is a large variance in the scaled execution costs. This accounts for the erratic behavior seen in Fig. 39. We repeated this experiment with controlled settings of selectivity of the predicate and availability of path index to study their effect upon the performance.

In one experiment, we varied the selectivity of the predicate (used for the path index) from 1% to 50%, while keeping the availability of the path index constant at 50%. Figure 40 shows that, for lower selectivities, there are significant gains in the estimated execution cost of the query. These gains decrease as the selectivity is increased. We have not reported the optimization times, since they are not affected by the selectivity of the predicate.

In the next experiment, we kept the selectivity of the predicate constant at 10% and varied the availability of the path index. Figure 42 shows the improvement in execution cost of the query as the availability is increased. Figure 41 shows that, although the increase in optimization time depends upon the availability of path indexes (the greater the availability, the greater the number of options to consider), it is never worse than 10%.

These experiments indicate that considering path indexes only marginally increases the optimization time of a query, while providing a dramatic reduction in query execution time if the predicate is suitably selective. Hence, this can be a very effective optimization technique.

Unnest. Methods of objects (attributes or links) can be set-valued. In that case, the unnest operator can be used to flatten such set-valued methods. Figures 43, 44, 45 show the various kinds of query-processing alternatives that need to be considered by the optimizer. Since unnest is a necessary "feature" and cannot be turned "off", we do not present a performance comparison here. All the previous experiments in this section included the unnest operator. Any method (attribute or link) referred to in the randomly generated queries had a 10% chance of being a set-valued method to which an unnest operator was applied.

3.6 Summary

In this section, we have described our experiences building optimizers using OPT++. We have seen how different operators and algorithms can be added to the optimizer. We have also seen how different optimization policies (for example, left-deep *vs.* bushy, select pushdown *vs.* exhaustive positioning) can be implemented in OPT++. In addition to System-R-style bottom-up building of operator trees, we have also been able to incorporate algebraic transformation rules in our optimizer. This flexibility has been achieved without sacrificing optimizer efficiency.

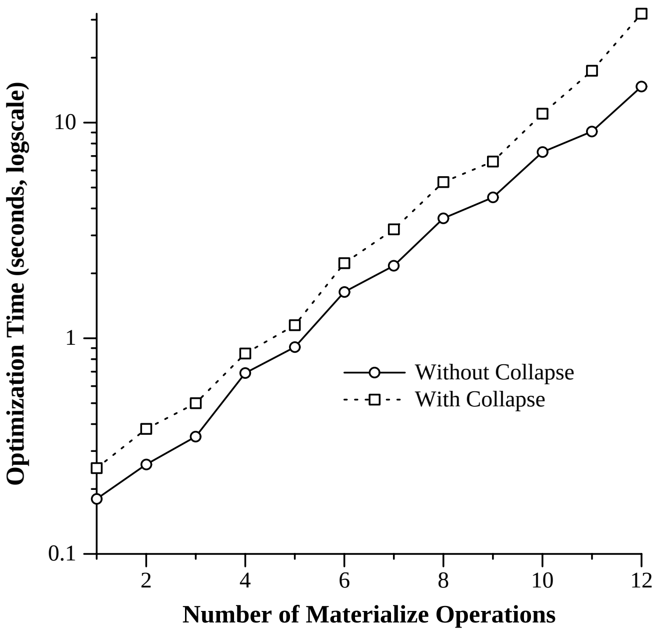

Fig. 35. Collapsing materializes in absence of pointer joins: optimization times (log-scale)

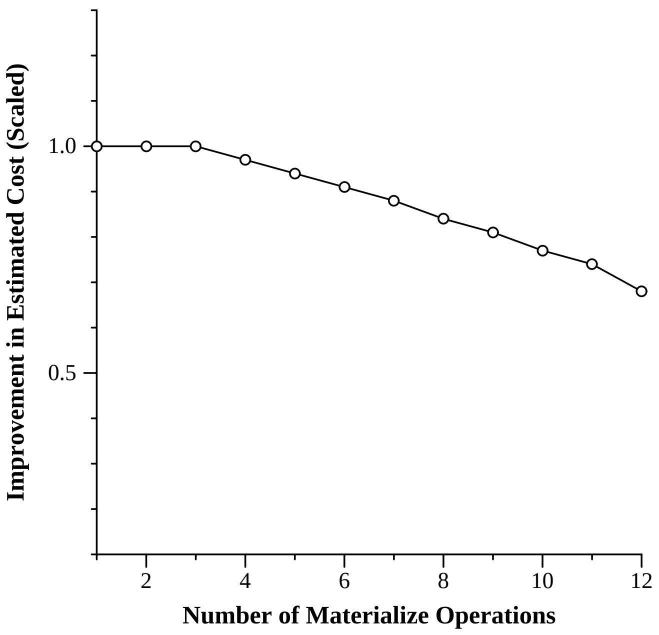

Fig. 36. Collapsing materializes in absence of pointer joins: improvement in estimated costs (scaled)

4 Related work

Extensible query optimizers proposed in the literature fall mainly into two categories: those that offer a fixed search strategy and make it easy to add 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 creating a design in which the search strategy itself is extensible, and, for any search strategy implemented using this framework, the addition of new algorithms and operators is easy.

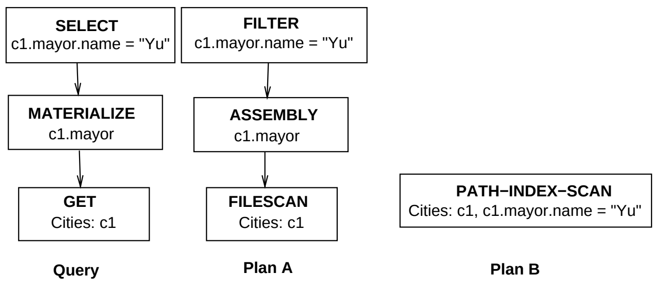

Fig. 37. Use of path indices

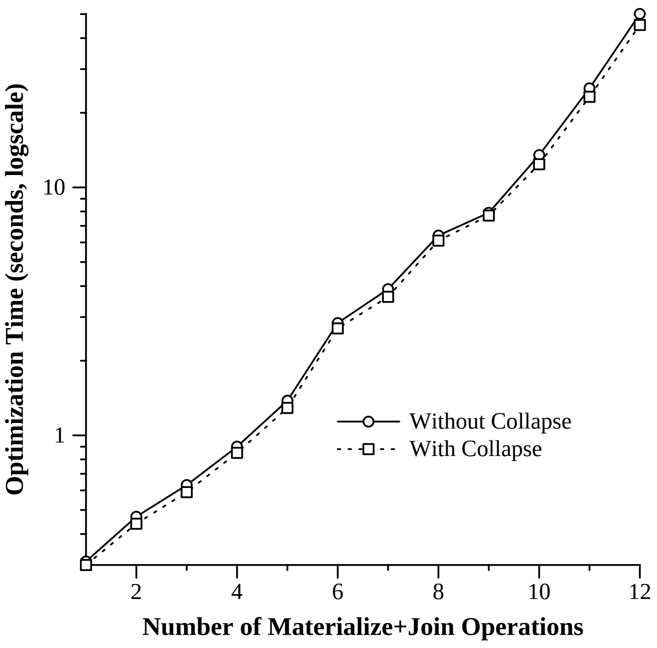

Fig. 38. Use of path indices: optimization times (log-scale)

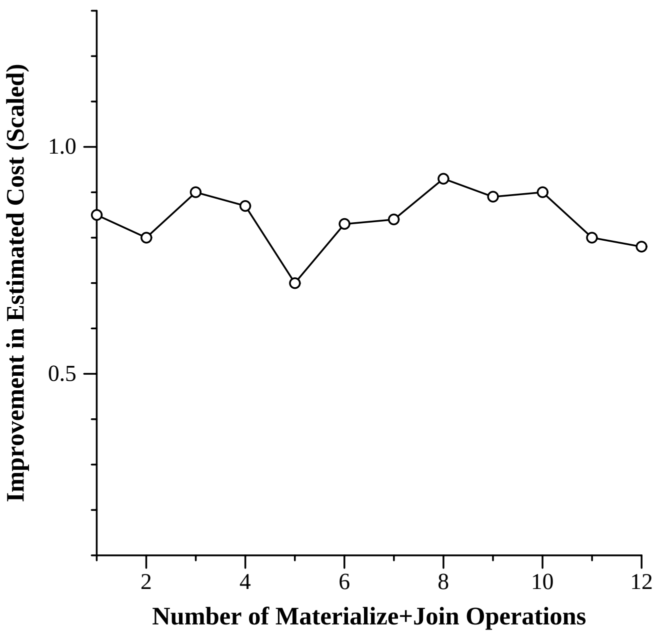

Fig. 39. Use of path indices: improvement in estimated costs (scaled)

Most optimizers that allow extensibility of the query algebra employ some form of a rule-based system that uses re-write 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 which are applied in order. The first set of rules is used to convert the query into an algebraic tree. The next set of rules is used to generate access paths. Another set generates join orderings. And a final set of rules determines what join methods to use.

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 heuristically transform the query into an equivalent 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. There can be multiple production rules (suggesting execution alternatives), and conditions of applicability associated with each “non-terminal” in the grammar. 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 re-write phase (first one) uses equivalence transformations to re-write 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.

[GLSW93] describes the query optimization in IBM’s DB2 family of products. These optimizers are partially based on the Starburst architecture. They allow the search strategy and the search space to be modified in a limited way on a per-query basis. Specifically, compile-time options permit the user to specify either the dynamic-programming algorithm or a greedy heuristic [Loh88] for join enumeration during optimization of each query. The user can also specify whether to allow composite inner tables (“bushy” plans) or not, and to defer cartesian products to the end or to permit them anywhere in the plan sequence. However, the basic search strategy remains *generative* (as opposed to *transformative*). The OPT++ architecture allows for much more flexibility in modifying the search strategy of an optimizer.

The optimizers generated by the Exodus Optimizer Generator [GD87], the Volcano Optimizer Generator [GM93] and the Cascades Framework [Gra95] use algebraic equivalence rules to transform an operator tree for a query into other, equivalent operator trees. Implementation rules are used to determine what algorithms can be used to implement the various operators. The algebraic transformation rules are used to generate all possible operator trees that are equiva-

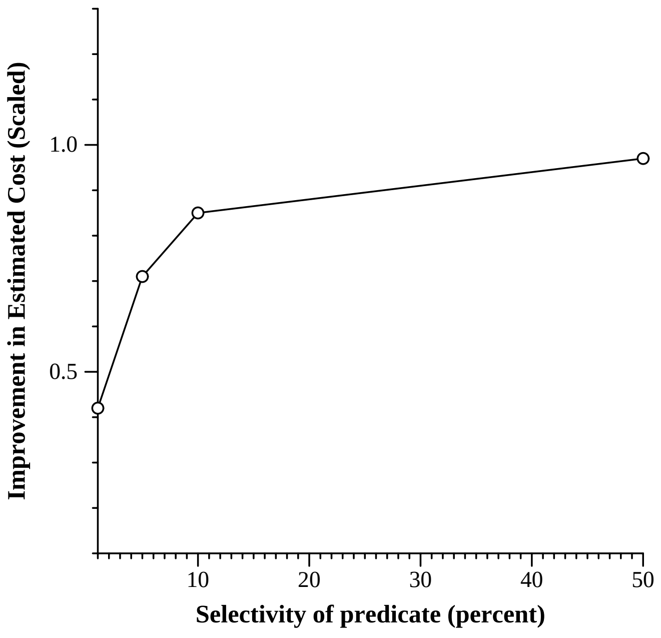

Fig. 40. Path indices: effect of selectivity on estimated costs (scaled)

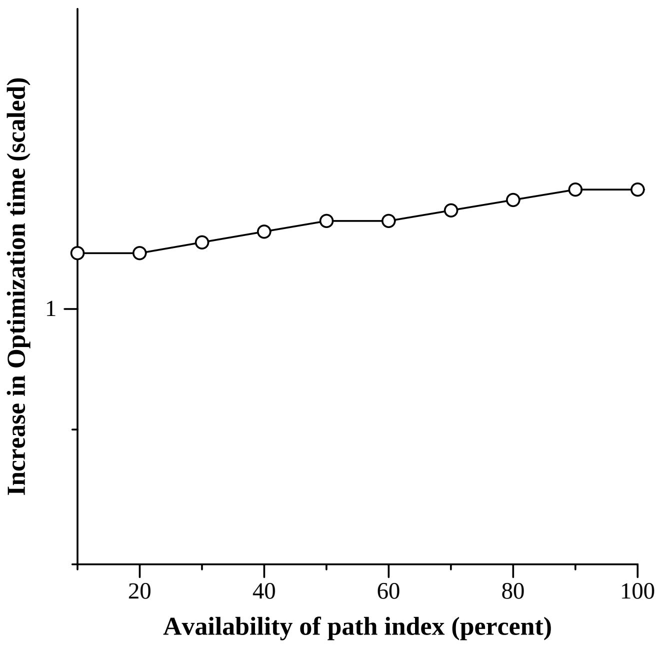

Fig. 41. Path indices (effect of availability): increase in optimization times

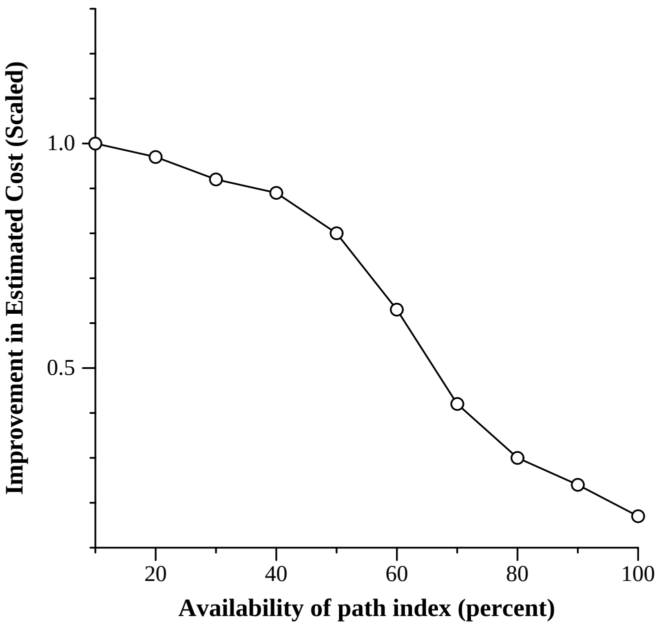

Fig. 42. Path indices (effect of availability): improvement in estimated costs (scaled)

lent 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++. In fact, OPT++ can also be used to implement the transformation rules and implementation rules of Volcano and the re-write rules and production rules of Starburst. In addition, the search strategy in OPT++ is extensible and can be modified to fit the optimization problem, if necessary. Our experience with the implementation of an optimizer using OPT++ shows that this flexibility is achieved without sacrificing performance.

The Cascades Framework [Gra95] is similar to the Volcano Optimizer Generator, but it uses C++ classes to represent the transformation rules, implementation rules and predicates. It also allows the search strategy to be “guided” through the use of user-defined guidance classes that can heuristically control the application of the transformation rules. However, the basic search strategy remains a transformative strategy that uses transformation rules to generate equivalent plans. It can be “guided”, but cannot be changed or replaced. For example, a System-R-style bottom-up optimizer cannot be implemented using the Cascades Framework.

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 [SS90] and the blackboard architecture of Kemper et al. [KMP93] are all based on the concept of dividing an optimizer into modules or regions that carry out different parts of the optimization. A query then has to pass through these various modules to be optimized. Flexibility of search can now be achieved by associating different behaviors with the different modules of the optimizer. These architectures differ in the methods used to pass control between the various modules.

In [SS90], control passes from one module to another in a fixed sequence. [MDZ93] uses a hierarchy of regions in which the parent module dynamically controls the sequence of regions through which the various alternative query evaluation plans pass while being optimized. In the blackboard

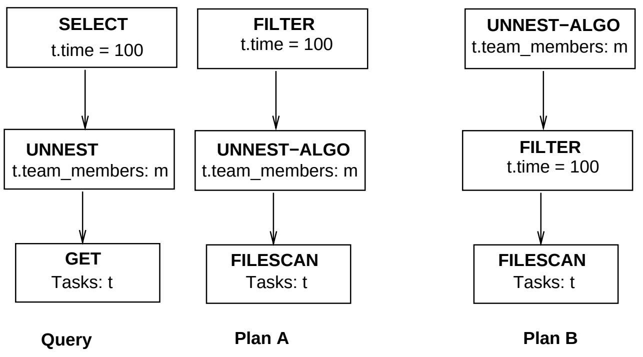

Fig. 43. Unnest and select operators

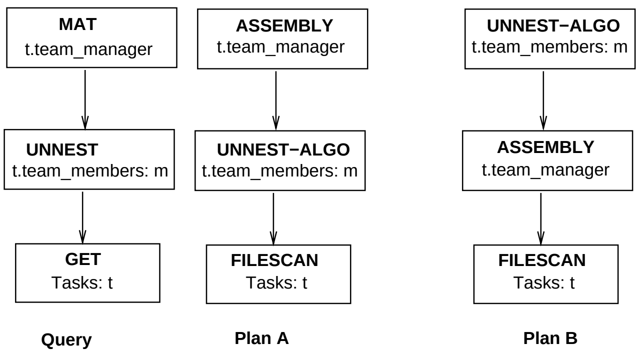

Fig. 44. Unnest and materialize operators

approach [KMP93], knowledge sources are responsible for moving the query evaluation plans between regions.

All these architectures describe very general frameworks for extensible query optimization. There are very few restrictions on the kinds of processing that can be implemented as a part of a module/region. Hence, almost any search strategy can be implemented in these frameworks. In OPT++, we sacrifice some of this generality to improve code re-use, and fine-grained extensibility. By imposing limitations on the kinds of manipulations that are allowed on the operator trees and access plans, OPT++ is able to put a significant fraction 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 spite of these limitations, we have shown in Sects. 2, and 3, that a number of different search strategies described in literature can easily be implemented in OPT++. Further, by making all OPERATORs, ALGORITHMs, GENERATORs, and the SEARCHSTRATEGIES objects with virtual methods, OPT++ makes it easier to to modify existing optimizer components to create new behaviors.

The query optimizer used in the [BG92] system uses a formal concept of a many-sorted relational algebra to design a rule-based optimizer that is extensible and can handle new data types. However, the architecture is based on algebraic equivalence rules. Hence, unlike OPT++, it limits the optimizer-implementor to implement only transformation-based optimization schemes.

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. The query optimizer of the TIGUKAT Object-base Management System [OMS95] is another optimizer that uses abstract base classes and virtual methods to achieve extensibility. OPT++ differs from both these systems in its modeling of the query algebra and the search space. Unlike OPT++, neither of these systems has a clear separation between the logical algebra (operator trees) and the physical algebra (access plans). We believe this separation is necessary for the efficiency of the optimizer, as well as for clarity and extensibility. [LV91] discusses extensibility of the search strategy in detail. However, it is not clear how extensible their design is in terms of adding new operators and algorithms, modifying the search space, or how such changes would interact with one another or with the search strategy. [OMS95] does differentiate between the search strategy, search space and the query algebra. However, the only TREETOTREEGENERATORs allowed are algebraic equivalence rules, hence System-R-style generative optimizers cannot be implemented using this approach.

The EROC toolkit for building optimizers [MBHT96] comes closest in terms of design philosophy to OPT++. EROC is a toolkit for building query optimizers based on components that are C++ abstract classes that they have identified as central to query optimization. These classes provide System-R- and Volcano-style search strategies, implementation of common algebraic equivalence rules, derivation of properties and handling of predicate manipulations, catalog information and types. At the current time, EROC does not have implementations of any other search strategies, but randomized algorithms and greedy heuristics are planned future work.

Although the basic principles of EROC are very similar to those of OPT++, there are fundamental differences between the two architectures. First, EROC does not differentiate between the search space and the search strategy components. There is an enumerator abstract class that determines both the search space that will be searched, and what search strategy will be used. We believe, that by separating these two components, OPT++ provides for more re-use of code and easier extensibility. Second, in OPT++ the mapping from the logical algebra (operator trees) to the physical algebra (access plans) is done on a per-operator-tree basis, by a number of different classes[11], each of which handles one specific type of mapping. By contrast, in EROC, the whole space of generated operator trees is transformed to access plans by a single call to a “Mapper” class. We believe, this model misses some opportunities at modularization and fine-grained control, and this would make it more difficult to modify or extend this operation. Finally, we would like to point out that the EROC architecture also contains abstractions to handle predicates, catalog information, types, and other “support” functions needed for implementing an optimizer. This is an issue not currently addressed in OPT++.

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

[11] Different classes derived from the TREETOPLANGENERATOR classes, as explained in Sect. 2.3

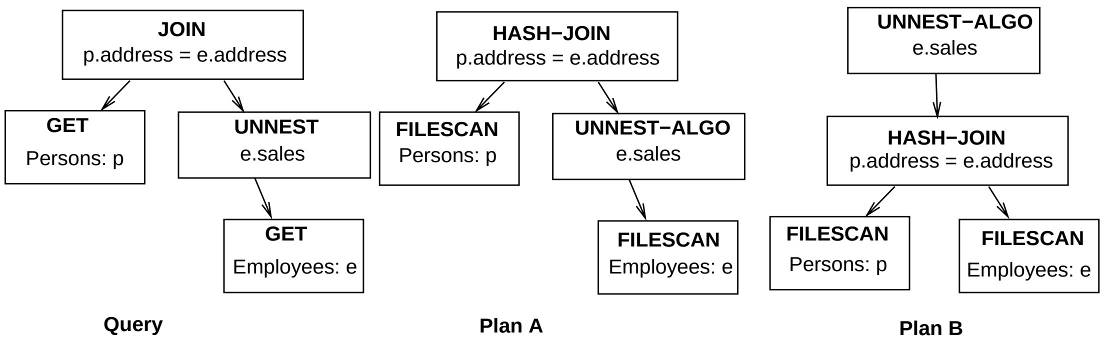

Fig. 45. Unnest and join operators

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.

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 optimizer for a new database system as well as to evaluate different optimization techniques and search strategies. This process can be very useful to an optimizer-implementor in 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 additional 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.

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 and anonymous reviewers for suggesting improvements to drafts of this paper.

References

[BG92] Becker L, Guting RH (1992) Rule-Based Optimization and Query Processing in an Extensible Geometric Database System. ACM Trans Database Syst 17: 2

[BMG93] Blakeley JA, McKenna WJ, Graefe G (1993) Experiences Building the Open OODB Query Optimizer. In: Buneman P, Jajodia S (eds) Proc. ACM SIGMOD Conference, May 1993, Washington, DC. ACM Press, New York, NY, pp 287–296

[FG91] Finance B, Gardarin G (1991) A Rule Based Query Rewriter in an Extensible DBMS. Proc. 7th International Conference on Data Engineering, 1991. IEEE CS Press, Piscataway, N.J., pp 248–256

[Fre87] Freytag JC (1987) A Rule-Based View of Query Optimization. In: Dayal U, Traiger IL (eds) Proc. ACM SIGMOD Conference, May 1987, San Francisco, Calif. ACM Press, New York, NY, pp 173–180

[GD87] Graefe G, DeWitt DJ (1987) The EXODUS Optimizer Generator. In: Dayal U, Traiger IL (eds) Proc. ACM SIGMOD Conference, May 1987, San Francisco, Calif. ACM Press, New York, NY, pp 160–172

[GLPK94] Galindo-Legaria C, Pellenkoft A, Kersten ML (1994) Fast, Randomized, Join-Order Selection – Why Use Transformations. In: Bocca JB, Jarke M, Zaniolo C (eds) Proc. 20th VLDB Conf., 1994, Santiago de Chile, Chile. Morgan Kaufmann, San Francisco, CA, pp 85–95

[GLSW93] Gassner P, Lohman GM, Schiefer KB, Wang Y (1993) Query Optimization in the IBM DB2 Family. Data Eng Bull 16(4): 4–18, December 1993

[GM93] Graefe G, McKenna WJ (1993) The Volcano Optimizer Generator: Extensibility and Efficient Search. In: Elmagarmid EK, Neuhold EJ (eds) Proc. IEEE Conf. on Data Eng, pp 209–218 Vienna, Austria, 1993. IEEE CS Press, Piscataway, NJ

[Gra87] Graefe G (1987) Rule-Based Query Optimization in Extensible Database Systems. PhD thesis. University of Wisconsin, Madison, Wis., November 1987

[Gra95] Graefe G (1995) The Cascades Framework for Query Optimization. Bull Tech Comm Data Eng 18(3): 19–29, September 1995

[Hel94] Hellerstein JM (1994) Practical Predicate Placement. In: Snodgrass RT, Winslett M (eds) Proc. ACM SIGMOD Conference, pp 267–276, Minneapolis, Minn. May 1994. ACM Press, New York, NY

[HP88] Hasan W, Pirahesh H (1988) Query Rewrite Optimization in Starburst. Research Report RJ 6367 (62349). IBM, Armonk, N.J.

[IK90] Ioannidis YE, Kang YC (1990) Randomized Algorithms for Optimizing Large Join Queries. In: Garcia-Molina H, Jagadish HV (eds) Proc. ACM SIGMOD Conference, pp 312–321. Atlantic City, NJ, May 1990. ACM Press, New York, NY

[IW87] Ioannidis YE, Wong E (1987) Query Optimization by Simulated Annealing. In: Dayal U, Traiger IL (eds) Proc. ACM SIGMOD Conference, San Francisco, Calif., June 1987. ACM Press, New York, NY, pp 9–22

[Kan91] Kang YC (1991) Randomized Algorithms for Query Optimization. Technical Report TR–1053. Computer Sciences Department, University of Wisconsin, Madison, Wis.

[KBZ86] Krishnamurthy R, Boral H, Zaniolo C (1986) Optimization of Nonrecursive Queries. In: Chu WW, Gardarin G, Ohsuga S, Kambayashi Y (eds) Proc. 12th VLDB Conf, August 1986, Kyoto, Japan. Morgan Kaufmann, San Francisco, CA, pp 128–137

[KGM91] Keller T, Graefe G, Maier D (1991) Efficient Assembly of Complex Objects. In: Clifford J, King R (eds) Proc. ACM SIGMOD Conference, May 1991, Denver, Colo. ACM Press, New York, NY, pp 148–157

[KMP93] Kemper A, Moerkotte G, Peithner K (1993) A Blackboard Architecture for Query Optimization in Object Bases. In: Agrawal R, Baker S, Bell DA (eds) Proc. 19th VLDB Conf, 1993. Morgan Kaufmann, San Francisco, CA, pp 543–554

[LFL88] Lee MK, Freytag JC, Lohman GM (1988) Implementing an Interpreter for Functional Rules in a Query Optimizer. In: Bancilhon F, DeWitt DJ (eds) Proc. 14th VLDB Conf, 1988, Los Angeles, Calif. Morgan Kaufmann, San Francisco, CA, pp 218–229

[Loh88] Lohman GM (1988) Heuristic Method for Joining Relational Database Tables. IBM Tech Disclosure Bull 30(9): 8–10

[LV91] Lanzelotte RSG, Valduriez P (1991) Extending the Search Strategy in a Query Optimizer. In: Lohman GM, Sernadas A, Camps R (eds) Proc. 17th VLDB Conf, September 1991, Barcelona, Spain. Morgan Kaufmann, San Francisco, CA, pp 363–373

[MBHT96] McKenna WJ, Burger L, Hoang C, Truong M (1996) EROC: A Toolkit for Building NEATO Query Optimizers. In: Vijayaraman TM, Buchmann AP, Mohan C, Sarda NL (eds) Proc. 22nd VLDB Conf, 1996, Mumbai (Bombay), India. Morgan Kaufmann, San Francisco, CA, pp 111–121

[MDZ93] Mitchell G, Dayal U, Zdonik SB (1993) Control of an Extensible Query Optimizer: A Planning Based Approach. In: Agrawal R, Baker S, Bell DA (eds) Proc. 19th VLDB Conf, 1993, Dublin, Ireland. Morgan Kaufmann, San Francisco, CA, pp 517–528

[OL90] Ono K, Lohman GM (1990) Extensible Enumeration of Feasible Joins for Relational Query Optimization. In: McLeod D, Sacks-Davis R, Schek H-J (eds) Proc. 16th VLDB Conf, August 1990, Brisbane, Queensland, Australia. Morgan Kaufmann, San Francisco, CA, pp 314–325

[OMS95] Özsu MT, Muñoz A, Szafron D (1995) An Extensible Query Optimizer for an Objectbase Management System. In: Pissinou N, Silberschatz A, Park EK, Makki K (eds) Proc. Fourth Int. Conf. on Information and Knowledge Management (CIKM), November 1995, Baltimore, Md. ACM Press, New York, NY, pp 188–196

[Pea84] Pearl J (1984) Heuristics. Addison-Wesley, Reading, Mass.

[PHH92] Pirahesh H, Hellerstein JM, Hasan W (1992) Extensible/Rule Based Query Rewrite Optimization in Starburst. In: Snodgrass RT, Winslett M (eds) Proc. ACM SIGMOD Conference, May 1992, Minneapolis, MN. ACM Press, New York, NY, pp 39–48

[SAC$^+$79] Selinger P, Astrahan M, Chamberlin D, Lorie R, Price T (1979) Access Path Selection in a Relational Database Management System. In: Bernstein PA (eds) Proc. ACM SIGMOD Conference on Management of Data, May 1979, Boston, MA. ACM Press, New York, NY, pp 23–34

[SC90] Shekita EJ, Carey MJ (1990) A Performance Evaluation of Pointer-Based Joins. In: Garcia-Molina H, Jagadish HV (eds) Proc. ACM SIGMOD Conference, May 1990, Atlantic City, N.J. ACM Press, New York, NY, pp 291–299

[SG88] Swami A, Gupta A (1988) Optimization of Large Join Queries. In: Boral H, Larson P-Å (eds) Proc. ACM SIGMOD Conference, 1988, Chicago, IL. ACM Press, New York, NY, pp 8–17

[SS90] Sciore E, Seig J jr (1990) A Modular Query Optimizer Generator. Proc. IEEE Conf. on Data Engineering, February 1990, Los Angeles, Calif., February 1990. IEEE CS Press, Piscataway, NJ, pp 146–153

[Swa89] Swami A (1989) Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. In: Clifford J, Lindsay BG, Maier D (eds) Proc. ACM SIGMOD Conference, June 1989, Portland, Ore. ACM Press, New York, NY, pp 367–376

[VM96] Vance B, Maier D (1996) Rapid Bushy Join-Order Optimization with Cartesian Products. In: Jagadish HV, Mumick IS (eds) Proc. ACM SIGMOD Conference, 1996, Montreal, Canada. ACM Press, New York, NY, pp 35–46