Query Optimization for Object-Relational Database Systems

By

Navin Kabra

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1999

Abstract

Modern database systems are placing an increasingly heavy burden upon their query optimizers. Commercial vendors are all scrambling to add object-relational features to their database systems, but unfortunately, optimizer technology has not kept pace with these advances for a number of reasons. Writing an optimizer, debugging it, and evaluating different optimization strategies remains a time-consuming and difficult task. Another problem is that attempts to design optimizers that can easily be extended to incorporate new operators, algorithms, or search strategies have enjoyed limited success. Finally, even the best of optimizers very often produce sub-optimal query evaluation plans. This problem is further aggravated by the presence of novel data domains and user-definable data-types and functions which make it very difficult to maintain statistics and estimate query execution costs.

In this thesis, we present some solutions to the problems that are currently facing query optimizers. We describe OPT++ an architecture that significantly improves the extensibility and maintainability of a query optimizer. We use this as a tool to implement a number of relational and object-relational optimization techniques and search strategies and perform a study to compare their relative performance. We also describe *Dynamic Re-Optimization*, a technique that can be used to tackle sub-optimality of plans produced by a query optimizer.

OPT++ is an architecture for implementing extensible query optimizers. It 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.

Another problem facing query optimizers is that, even the best query optimizers can very often produce sub-optimal query execution plans, leading to a significant degradation of performance. This is especially true in databases used for complex decision support queries and/or object-relational databases. We describe *Dynamic Re-Optimization*, an algorithm that detects sub-optimality of a query execution plan during query execution and attempts to correct the problem. The basic idea is to collect statistics at key points during the execution of a complex query. These statistics are then used to optimize the execution of the query, either by improving the resource allocation for that query, or by changing the execution plan for the remainder of the query. To ensure that this does not significantly slow down the normal execution of a query, the Query Optimizer carefully chooses what statistics to collect, when to collect them, and the circumstances under which to re-optimize the query. We describe an implementation of this algorithm in the Paradise Database System, and we report on performance studies, which indicate that this can result in significant improvements in performance of complex queries.

Acknowledgements

I am deeply indebted to my advisor, Professor David J. DeWitt for all the advice, support and guidance he has provided me with. He was always there, providing motivation, enthusiasm, and direction to this research. I am also very thankful for the extraordinary amount of patience he has shown all these years as I procrastinated my way through my thesis. It has been a privilege to have worked with him.

Studying databases at the University of Wisconsin - Madison has allowed me to work with some of the best database researchers in the world. I would like to think Professors Yannis Ioannidis, Mike Carey and Jeff Naughton for the insights and guidance they provided during the course of this research, and the helpful comments and suggestions on various drafts of my papers and my thesis. I would also like to thank Professors Raghu Ramakrishnan and Rick Jenison for being on my committee and helping to improve the quality of this thesis.

A lot of this research was done in the context of the Paradise project. I have had a great time working on this project, and I would like to thank all the members of the Paradise team, especially Jignesh Patel, Jie-Bing Yu, Kristin Tufte, Biswadeep Nag, and Karthik Ramasamy, for their help and support. A lot of the achievements of the Paradise project would have been impossible without the SHORE project. I am grateful to the people who worked on the SHORE project, especially Nancy Hall, Jim Kupsch and Josef Burger (Bolo).

I would also like to thank Vishy Poosala, Joey Hellerstein, and Praveen Seshadri for the numerous discussions we've had. Speaking to them always helped clarify my thoughts.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation for This Research

One of the key reasons for the success of relational database technology is the use of declarative languages and query optimization. The user can just specify what data needs to be retrieved and the database takes over the task of finding the most efficient method of retrieving that data. It is the job of the query optimizer to evaluate alternative methods of executing a query, and selecting the cheapest alternative.

Notwithstanding the tremendous success of this approach, query optimization still remains a problem for database systems. Modern database systems are placing an increasingly heavy burden upon their query optimizers. Relational database systems are increasingly being used to execute complex decision support queries. In addition, commercial vendors are all scrambling to add object-relational features to their database systems. Unfortunately optimizer technology has not kept pace with these advances, and a number of the inadequacies of traditional query optimizers have become obvious.

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.

The architecture of the query optimizer should be such that it is easy to write, modify, extend and maintain the optimizer. It should be easy to implement and experiment with various different optimization techniques and strategies and evaluate their comparative performance. It should be easy to change the optimizer in case of changes in requirements, and easy to debug it in case of problems.

Another major problem facing query optimizers is that they often don't work. Even the best optimizers often produce sub-optimal query execution plans due to their inability to accurately estimate the cost of executing complex query evaluation plans.

There are a number of reasons why estimating the cost of query execution is difficult. Query optimizers use statistics stored in the system catalogs to estimate sizes and cardinalities of tables that participate in the query. This introduces an error in the estimates either due to the approximations involved, or because statistics are not kept up-to-date. As the number of joins in the query increases, these errors multiply and grow exponentially [IC91]. Another source of errors is the lack of sufficient information about the run-time system at query optimization time. The amount of available resources (especially memory), the load on the system, and the values of host language variables are things that differ for every execution of the query, and in some cases, change in the middle of query execution.

The problem is further aggravated in the case of object-relational database systems that allow users to define data-types, methods, and operators. Collection and storage of statistics (for example, histograms) for user-defined data-types (for example, spatial data-types like polygon, point) is an area that has not yet been addressed in database research literature. There are some primitive methods that have been proposed to deal with the estimation of the cost of execution for user defined functions/methods written in an external language (like C++) [SAH87], but these are far from adequate. Selectivity estimation for predicates involving user-defined methods/functions is an issue not yet considered in database research. All of this makes it really difficult to properly estimate the cost of execution of queries on object-relational databases.

Hence there is a pressing need for a query optimization framework that provides some solution to this problem.

1.2 Contributions of This Research

The goal of this research is to provide a new framework for query optimization which is better geared to tackle the problems posed by modern database systems. We present a novel architecture that provides extensibility, maintainability, and ease of experimentation. In addition, we also present an algorithm for dealing with sub-optimalities of query execution plans generated by optimizers. Specifically, the contributions of this research are in the following four areas:

Extensible Query Optimization

We describe OPT++ a architecture for extensible query optimization. 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.

Experimenting with Various Optimization Techniques

One very important facility that any optimizer should provide is the ability to easily try out new optimization techniques, experiment with various optimization alternatives and try to determine what are the best strategies to incorporate into an optimizer for a given database system. We show how OPT++ can be used for such experimentation. We have also conducted a detailed performance study of various optimization techniques that are currently available to optimizer implementors. We include our recommendations that can be of help to an Optimizer-Implementor in designing an optimizer for a specific database system.

Debugging Support for Optimizers

Debugging an optimizer can be a frustrating task. A large percentage of the time required to implement an new optimizer and maintain an existing optimizer is consumed by debugging. We present a novel technique in which the optimizer itself aids the Optimizer-Implementor in the debugging process. This greatly reduces the development time for optimizer implementation and results in a tremendous increase in the productivity.

Dealing with Sub-Optimal Query Execution Plans

We describe *Dynamic Re-Optimization*, an algorithm that tackles the problem of sub-optimal query execution plans by detecting sub-optimality of a query plan at execution time and dynamically re-optimizing the query in mid-execution to improve performance. We ensure that this is achieved without significantly compromising the performance of the system for queries that were not sub-optimal. We report the results of a performance study which indicates that this algorithm is quite effective and can result in significant improvements in performance of complex queries.

1.3 Outline of This Thesis

The remainder of this thesis is organized as follows.

Chapter 2 describes OPT++, the extensible, modifiable, maintainable architecture for building query optimizers.

Chapter 3 describes our experiences building optimizers using OPT++. This also includes a detailed performance study of various optimization techniques and search strategies that can be used for relational and object-relational optimization. This study would be valuable to an Optimizer-Implementor in determining what features to implement in an optimizer for a specific database system. It also shows that OPT++ is flexible, extensible and efficient.

Chapter 4 describes the debugging support built into OPT++ that can be used by an Optimizer-Implementor to track down bugs in the optimizer. It also describes a very general and powerful mechanism of giving hints to an optimizer to make it produce the right plan when it is producing sub-optimal query execution plans.

Chapter 5 describes the *Dynamic Re-Optimization* algorithm which detects sub-optimal query execution plans at query execution time and re-optimizes them to improve performance.

Chapter 6 compares this research with related work and in Chapter 7 we present our conclusions.

Chapter 2

The OPT++ Extensible Optimizer Architecture

2.1 The Need for Extensible Optimizers

In this chapter we describe the OPT++ architecture for extensible and maintainable query optimization. OPT++ exploits the object-oriented features of C++ to achieve this. It defines a few key abstract classes with virtual methods. These class definitions do not assume any knowledge about the query algebra or the database execution engine. The search strategy is implemented **entirely** in terms of these abstract classes. The search strategy invokes the virtual methods of these abstract classes to perform the search and the cost-based pruning of the search space.

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

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

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 minimal impact on the other components.

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. In some cases, it might also be possible to re-use 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 Chapter 6), 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 chapter 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

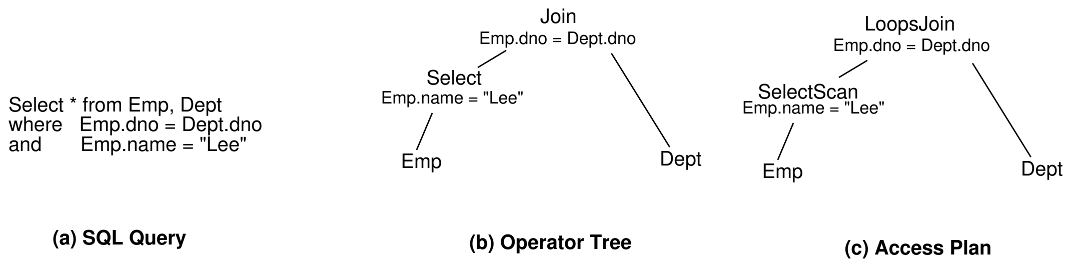


Figure 1: Query Representations

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.

2.2 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. 1(a) shows an SQL query and Fig. 1(b) shows that query represented as a tree of relational operators. A given query can be represented by one or more operator trees that are equivalent.

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

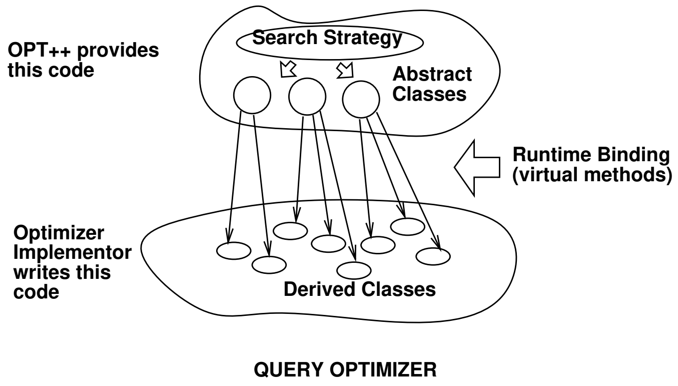


Figure 2: Basic System Design

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. Fig. 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.

2.3 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.

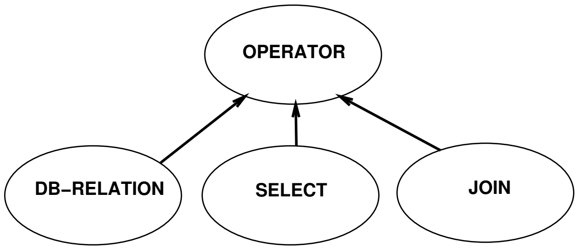


Figure 3: Operator Class Hierarchy for a Relational Optimizer

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.

2.3.1 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

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

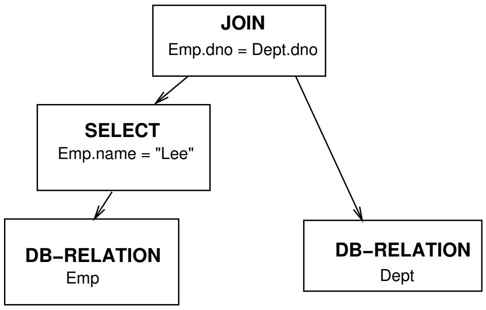


Figure 4: An Example Operator Tree

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.

Fig. 4 shows an example of an operator tree[2] corresponding to the query shown in Fig. 1. 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

[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.

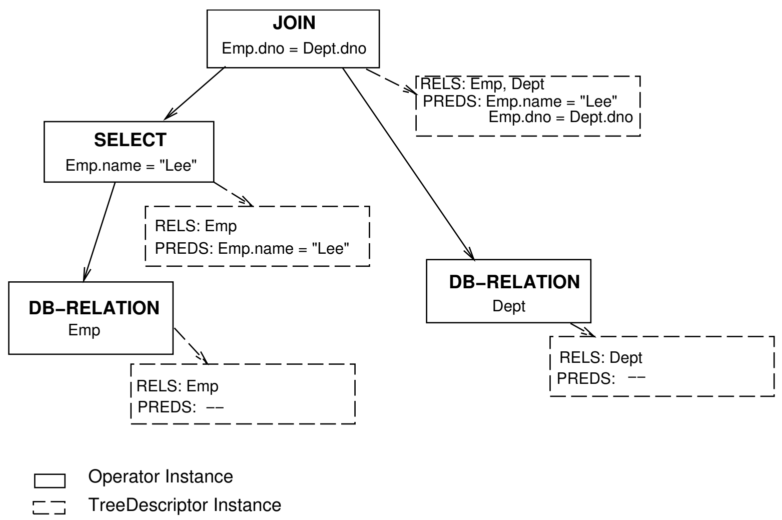


Figure 5: An Example Operator Tree with its Tree Descriptors

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. Fig. 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 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.

2.3.2 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

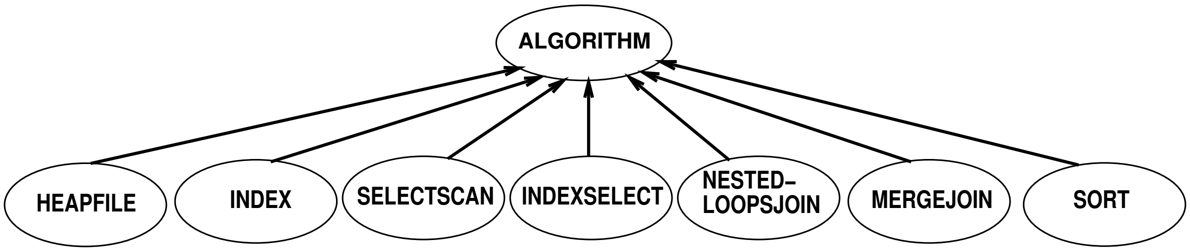


Figure 6: Algorithm Class Hierarchy for a Relational Optimizer

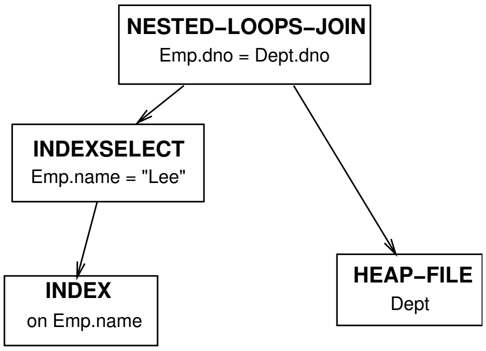


Figure 7: An Example Access Plan

operators in the database system. The Optimizer-Implementor is expected to derive one class from the `ALGORITHM` class for 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` and `INDEX`[3] 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.

Fig. 6 shows the algorithm classes that were derived from the abstract `ALGORITHM` class for our simple SQL optimizer. The `HEAPFILE` and `INDEX` algorithms are

[3]A system that has more than one type of index might use different classes to represent each index type. For example `BTREEINDEX` and `HASHINDEX`.

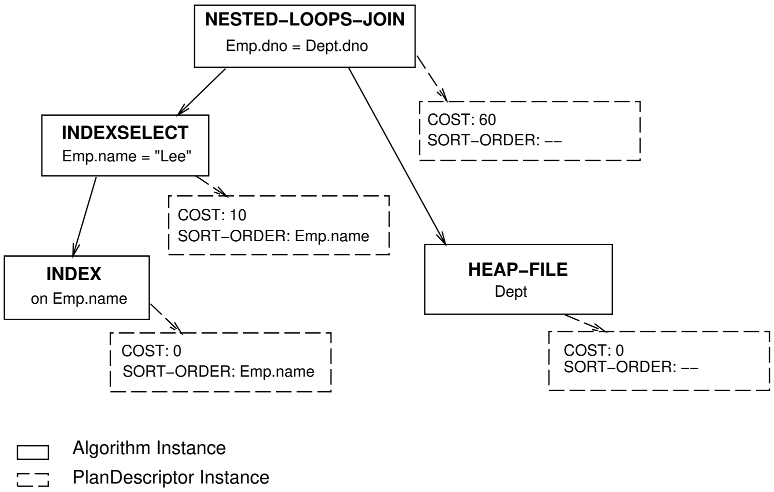


Figure 8: An Example Access Plan with its Plan Descriptors

dummy algorithms for the `DB-RELATION` operator as explained earlier. The `SELECT-SCAN` algorithm used to implement the `SELECT` operator represents a sequential scan of a `HEAPFILE` 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.

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

Similar to the `TREEDESCRIPTOR` class in the case of operator trees, OPT++ 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. Fig. 8 reproduces the access plan of Fig. 7 showing the

[4]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 Section 2.4.

`PLANDESCRIPTOR` instances associated with each algorithm instance.

The Optimizer-Implementor should provide an `IsEquivalent` method for the `PLAN-DESCRIPTOR` 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[5]. 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 outputs that are required for subsequent query

[5] 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].

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.

2.4 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.

2.4.1 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.

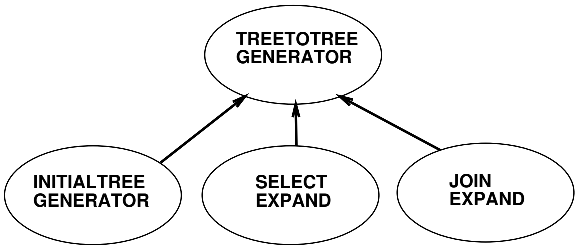


Figure 9: Example TREETOTREEGENERATOR Class Hierarchy

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

Fig. 10(b) illustrates the JOINEXPAND::APPLY method being invoked during the optimization of the query in Fig. 10(a). The figure shows an instance of the SELECT

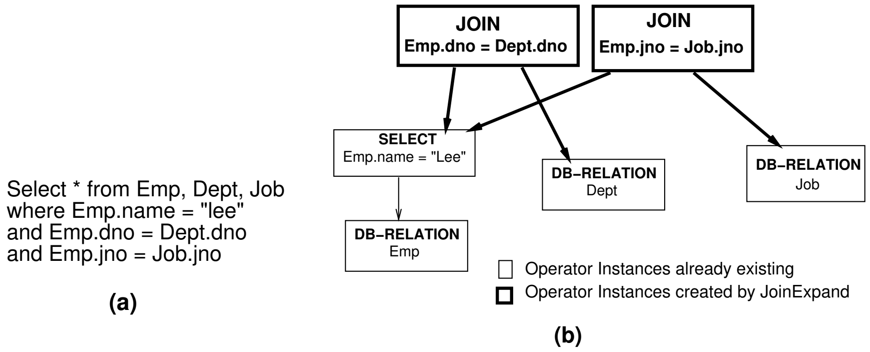


Figure 10: Application of `JOINEXPAND::APPLY`

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 `TREE-TOTREEGENERATOR` 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 strategy picks some operator instance (representing an operator tree) and generates new operator trees from it by invoking the APPLY method of various TREETOTREEGENERATOR classes on it. The CANBEAPPLIED method is used to determine whether the TREETOTREEGENERATOR should be applied to that operator instance. This process can be repeated to generate various operator trees corresponding to the input query.

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

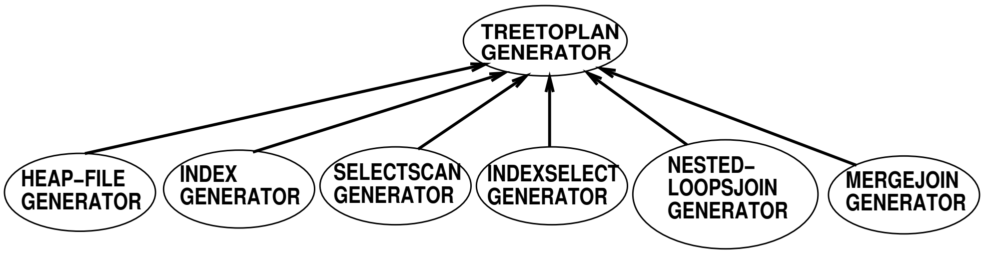


Figure 11: Example `TREETOPLANGENERATOR` Class Hierarchy

2.4.2 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 that operation. Fig. 11 shows the classes derived from the `TREETOPLANGENERATOR` class.

Fig. 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.

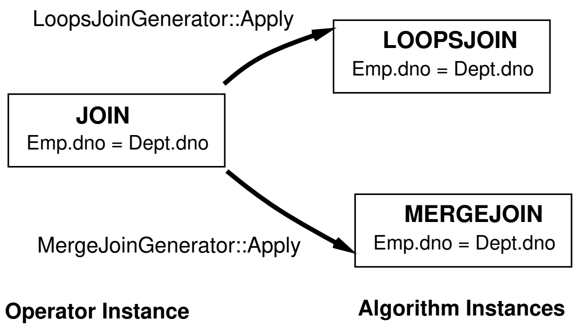


Figure 12: Examples of `TREETOPLANGENERATOR::APPLY`

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

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

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

2.4.3 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

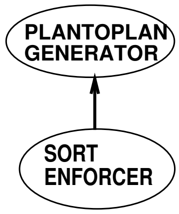


Figure 13: PLANTOPLANGENERATOR Class Hierarchy

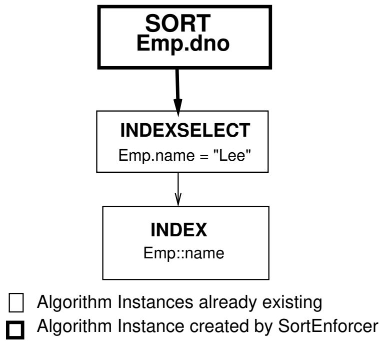


Figure 14: Use of SORTENFORCER::APPLY to enforce a sort-order

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.

Fig. 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 CANBEAPPLIED virtual method that determines whether the PLANTOPLANGENERATOR can be applied to the given input.

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

2.5 The Search Strategies

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

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

GENERATOR abstract classes, and in the next section we describe how the Optimizer-Implementor can easily switch from one search strategy to another.

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.

2.5.1 The Bottom-up Search Strategy

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

The INITIALTREEGENERATOR is invoked to initialize the collection of operator trees. To generate bigger trees, the search strategy picks an existing operator tree and *expands* it. To expand an operator tree, it determines what TREETOTREEGENERATORs can be applied to the operator instance at its root by 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[6].

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.

2.5.2 The Transformative Search Strategy

Section 2.4.1 only gave 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

[6]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.

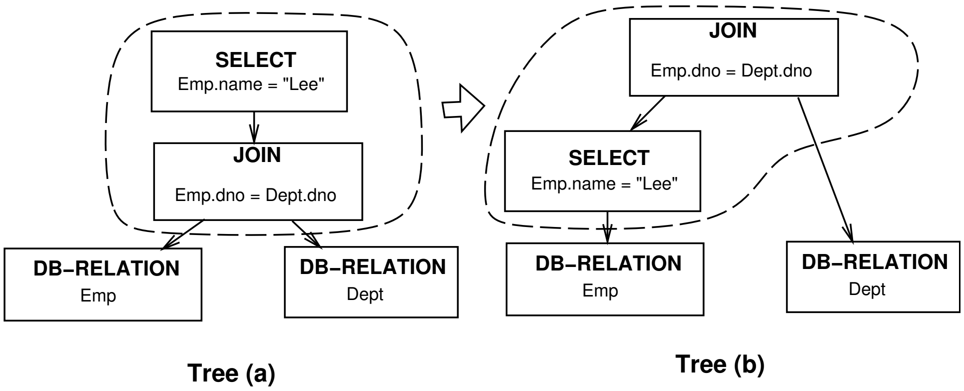


Figure 15: A Rule-based Transformation

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.

Fig. 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." Fig. 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 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::INPUT`, method 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 a the exact structure of the subtree 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 subtree expression consisting of operator names. The search strategy then ensures that only subtrees matching the specified expression are instantiated. The above discussion applies to `TREETOPLANGENERATOR`s and `PLANTOPLANGENERATOR`s 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].

2.5.3 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. See [Kan91] for details.

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` is chosen and applied to give rise to a new operator instance.

Then a random `TREETOPLANGENERATOR` that can be applied to the new operator instance, is chosen and used to generate a new algorithm 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.

After each step the *temperature* is decreased using some function which is an input parameter for this algorithm. This process is then repeated. Optimization continues until the temperature falls below a certain threshold or there is no improvement in the cost for some number of steps. At this point, the current plan is output as the optimal plan.

2.6 Extensibility in OPT++

This section summarizes what is involved in implementing a new optimizer, or extending or modifying an existing optimizer built using OPT++. Chapter 3 has some examples of such extensions as applied to a real optimizer.

2.6.1 Implementing a new Optimizer

Fig. 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

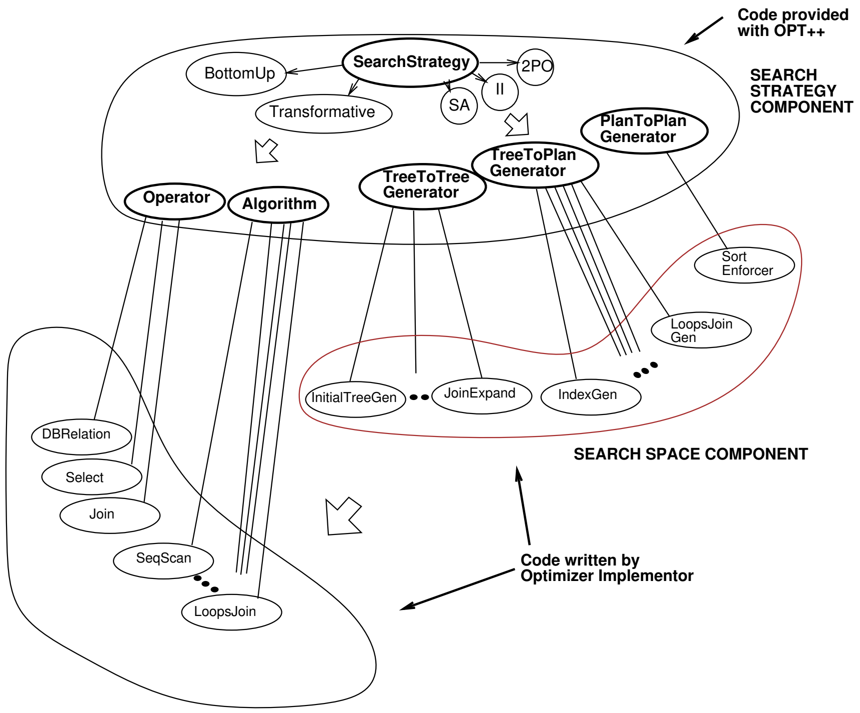


Figure 16: Implementing an Optimizer in OPT++

system, and therefore does not have to be modified 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 over-ride 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.

2.6.2 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 PLANTO-PLANGENERATORs 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 Chapter 3.

Chapter 3

Relational and Object-Relational Optimization Using OPT++

In this section, we describe our experiences building relational and object relational optimizers using OPT++. We first built a traditional System-R style relational optimizer and studied its performance. We then extended and modified it in various ways – to study various techniques for object-relational optimization; and to study the effects of different heuristics and search strategies on relational and object-relational optimizers. In each case, we describe how easy or difficult it was to extend the optimizer to incorporate the new technique. We also study the effect of each upon the performance of the optimizer.

The purpose of this section is twofold. First, it is a detailed study of the impact of many optimization techniques upon an optimizer (in terms of ease of implementation, effect upon the optimization time, and effect upon the quality of the plans produced). These results of this section can be used by an Optimizer-Implementor to determine what optimization features to include and what features to exclude from an optimizer for a particular database system. Second, it gives an idea of the kind of optimizers and optimization techniques that can be implemented using OPT++. The diversity of techniques that have been incorporated, and the ease with which most of them were implemented, attests to the extensibility and flexibility of OPT++. The performance figures indicate that this extensibility is achieved without sacrificing performance.

3.1 Relational Optimization

In this section we consider a simple relational optimizer that does System-R style join enumeration, and describe how it was 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.

3.1.1 Join Enumeration

Since all the examples used in Chapter 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 `MERGE-JOIN` 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`, `SEL-ECTSCANGENERATOR`, `INDEXSELECTGENERATOR`, `NESTEDLOOPSJOINGENERATOR`, and `MER-GEJOINGENERATOR` were derived from `TREETOPLANGENERATOR` to indicate how the corresponding algorithms could be used to implement the associated operators. `SORTEN-FORCER` 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 `TREETOTREEGENERATOR`, `TREETOPLANGENERATOR`, and `PLANTOPLANGENERATOR` classes consists of 150 lines of code. In contrast, the "Search Strategy" component, which consists entirely of code that is provided with OPT++ (*i.e.*, the Optimizer-Implementor does not have to write this code) was about 2500 lines of code. The fact that the search strategy code is already provided and does not have to be written or modified by the Optimizer-Implementor considerably simplified the task of writing the optimizer. Further, as will become clear later, the fact that the "Search Space" component is very small (150 lines of code 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 Section 2.6.1 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

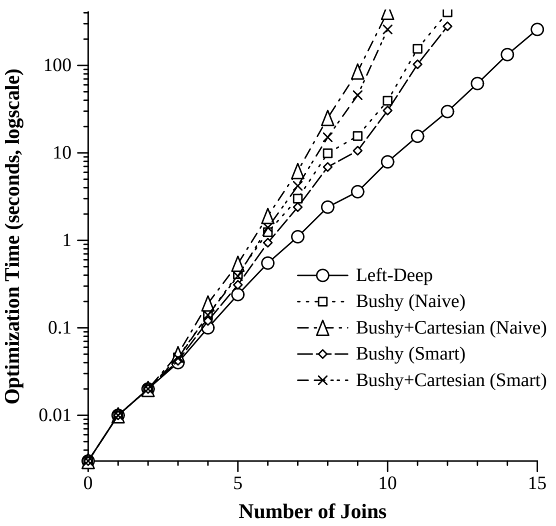


Figure 17: Comparison of Search Spaces: Optimization Times (Log-scale)

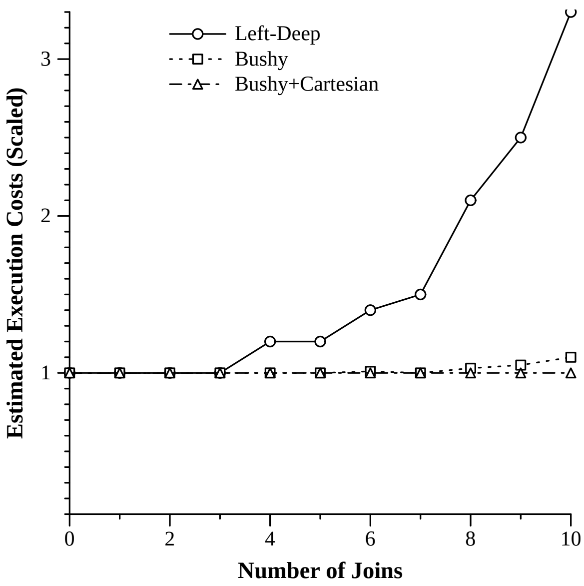


Figure 18: Comparison of Search Spaces: Estimated Costs (scaled)

generated randomly and optimized. The experiments were run on a 200Mhz Pentium Pro processor with 128MB of memory. However, virtual memory was limited to 32MB (using the *limit* command).

Fig. 17 illustrates the effect of different search spaces on the optimization time. Fig. 18 shows the effect on the relative estimated execution costs of the optimal plans produced[1]. (Note that optimization times are shown on a logarithmic scale.) Since these algorithms are quite easy to implement, an Optimizer-Implementor should definitely consider implementing them in an optimizer if the database execution engine is capable of executing bushy execution plans.

3.1.2 A Transformative Optimizer

In this section, we describe how the optimizer was changed from a bottom-up dynamic programming optimizer to one that uses algebraic transformation rules. In other words,

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

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[2]. 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 and 20 compare the two optimizers in terms of optimization times and memory consumed for randomly generated queries of increasing sizes. As before, the experiments were run on a 200Mhz Pentium Pro processor with memory limited to 32MB of memory. The figures show us that the performance of the Transformative

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

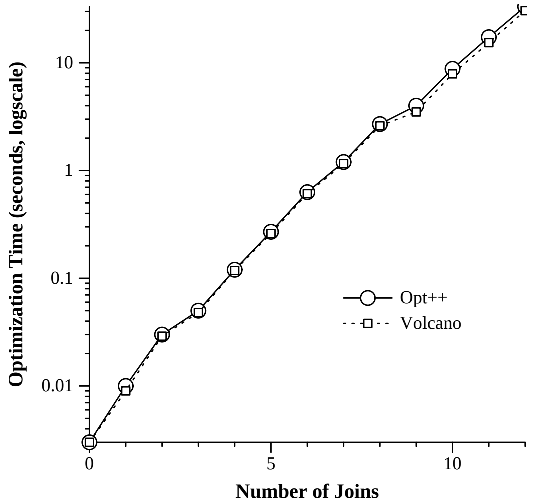


Figure 19: OPT++ *vs.* Volcano: Optimization Times (Log-scale)

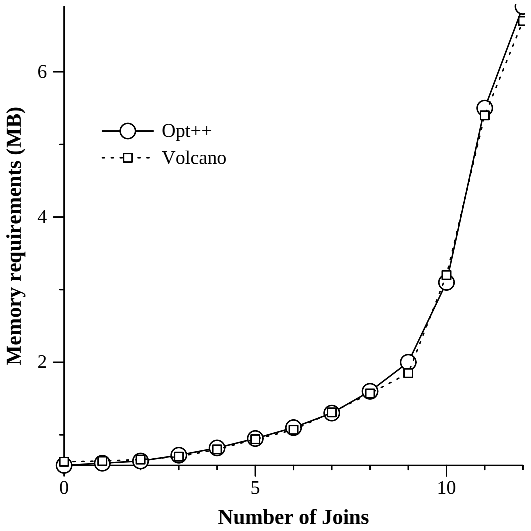


Figure 20: OPT++ *vs.* Volcano: Memory Requirements

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.

3.1.3 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 Phase Optimization (or *vice versa*) can be trivially accomplished by changing one line of code. The input parameters for the randomized strategies were chosen to be the same as those used in [Kan91].

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.

3.1.4 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]. Figs. 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 the two strategies. 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 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. For queries with more than 16 joins, the exhaustive algorithms quickly run out of memory, and don't produce any results at all, whereas the randomized algorithms continue functioning without any significant degradation in performance. The fact that the exhaustive algorithms produce no plans at all for large queries explains the sudden drop in the scaled costs of the randomized algorithms in Figure 22. They also confirm the findings of [IK90] that Two Phase Optimization performs better than either Simulated Annealing or Iterated Improvement.

In Figure 23, the memory requirements of the different strategies are presented. The randomized strategies require a negligible amount of memory irrespective of the size of the input query, while the exhaustive strategies require exponentially increasing amounts of memory. Hence, for queries larger than those shown in Figure 21, the

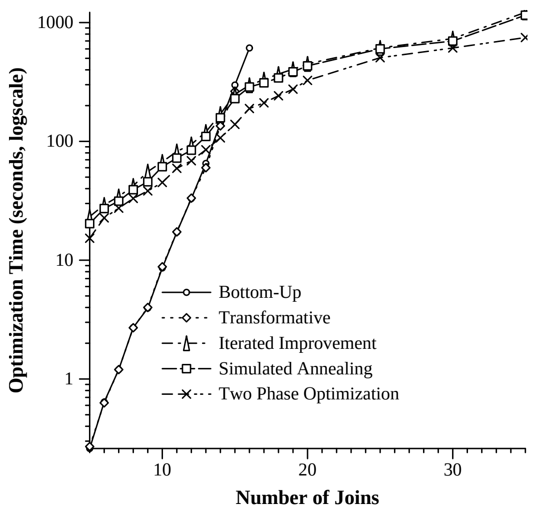


Figure 21: Comparing Search Strategies: Optimization Times (Log-scale)

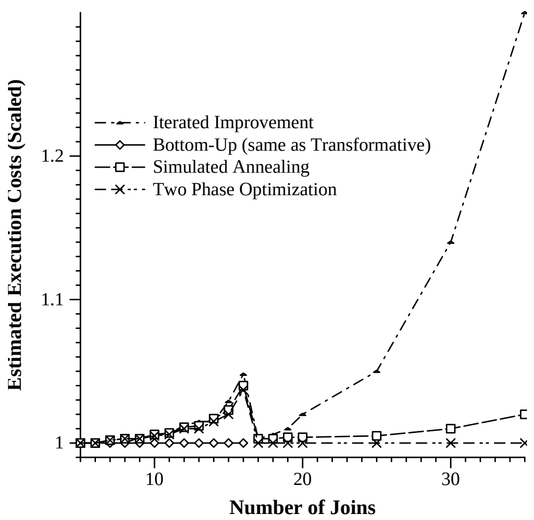


Figure 22: Comparing Search Strategies: Estimated Costs (Scaled)

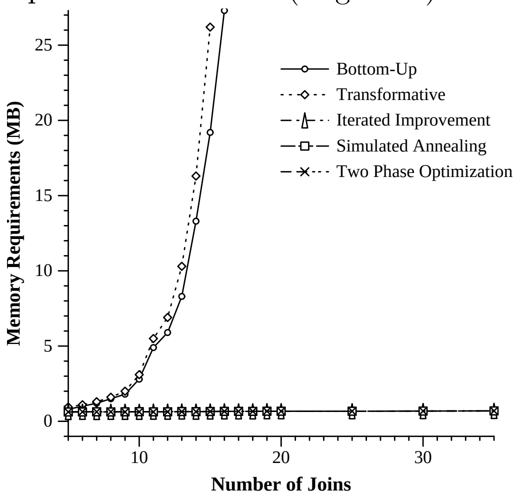


Figure 23: Comparing Search Strategies: Memory Requirements

randomized strategies will continue to give reasonable performance while the exhaustive strategies will fail due to lack of enough memory. We also note that although the Bottom-Up and Transformative search strategies have comparable performance in terms of optimization time and quality of plans produced (because both are exhaustive strategies and explore the same search space), the Bottom-Up strategy has a significant advantage in terms of space consumption as it can perform more aggressive pruning of operator trees.

3.2 Optimizing Object-Relational Operators

In this section, we describe a number of techniques that can be used for optimizing queries containing object-relational operations. Specifically we consider techniques to optimize queries containing path expressions, user defined methods (possibly expensive), set valued attributes, and generalized aggregates.

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.

3.2.1 Optimizing Queries Containing References

This section deals with queries containing the use of inter-object references. A number of techniques to optimize queries containing references are described in [BMG93]. We implemented them in our OPT++-based optimizer and conducted a performance study that is described in this section.

Specifically, 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].

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 POINTER-JOINEXPAND class from it. This new class creates instances of the JOIN operator that actually correspond to materialization of reference-valued attributes using a pointer-based join.

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

A new PATHINDEXSELECT algorithm was derived from the ALGORITHM class to capture such path-index scans. A PATHINDEXSCANGENERATOR class was derived from the TREETOPLANGENERATOR class to replace occurrences of a string of materialize operators followed by a select operator in an operator tree by a single PATHINDEXSELECT algorithm, if possible.

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

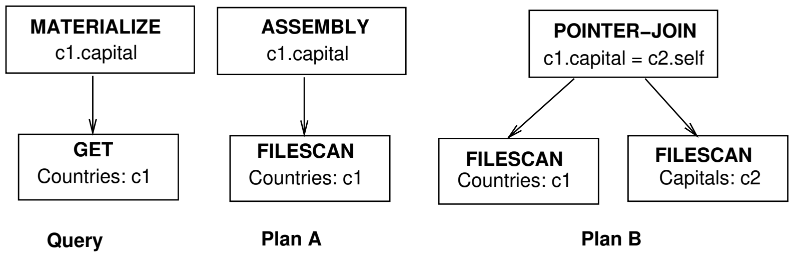


Figure 24: Converting Materializes to Joins

cost and selectivity estimation) and about 100 lines of code to the "Search Strategy" component, and was accomplished in about 3 weeks.

In the following sections, we describe specific optimizations involving references that may be incorporated into an object-relational optimizer. We describe each optimization in a little more detail and study its effect on the performance of the optimizer

Using randomly generated queries of varying sizes, we compared the performance of the optimizer with the feature turned "on", to that of the optimizer with the feature turned "off". To give an idea about the trade-offs involved, we have reported the effect upon the time required for optimization and the estimated costs of the resulting optimal query execution plans. Each feature was evaluated with all the remaining features turned on.

Converting Materializes to Joins

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"[3]. (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

[3]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.

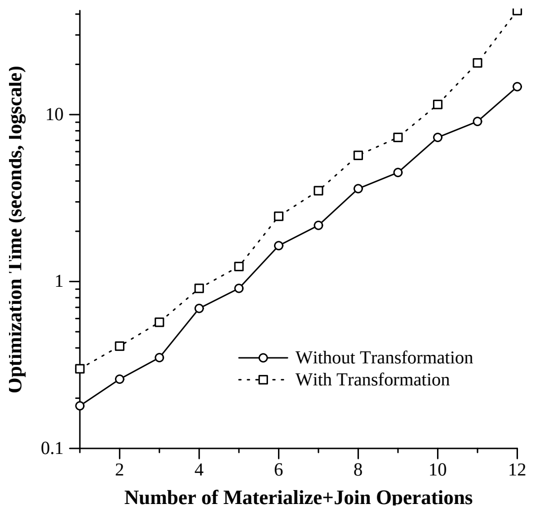


Figure 25: Converting Materializes to Joins: Optimization Times (Log-scale)

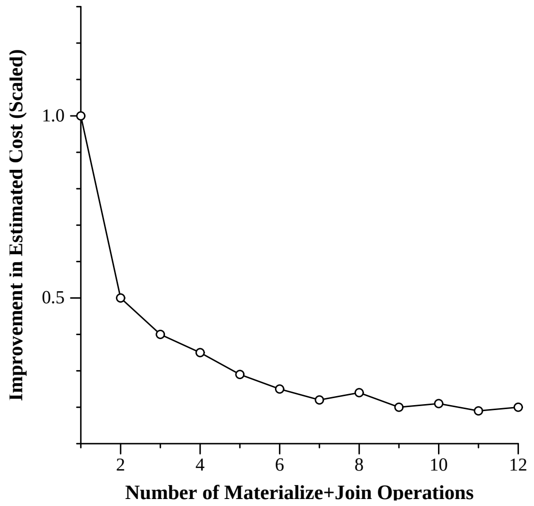


Figure 26: Converting Materializes to Joins: Improvement in Estimated Costs (Scaled)

`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 8 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.

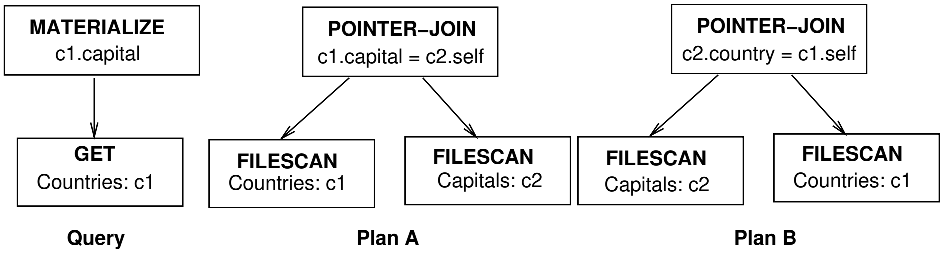


Figure 27: Use of Inverse Links

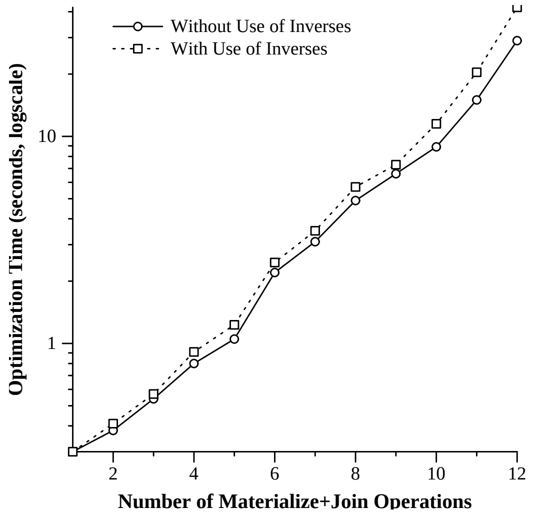


Figure 28: Use of Inverse Links: Optimization Times (Log-scale)

Figure 29: Use of Inverse Links: Improvement in Estimated Costs (Scaled)

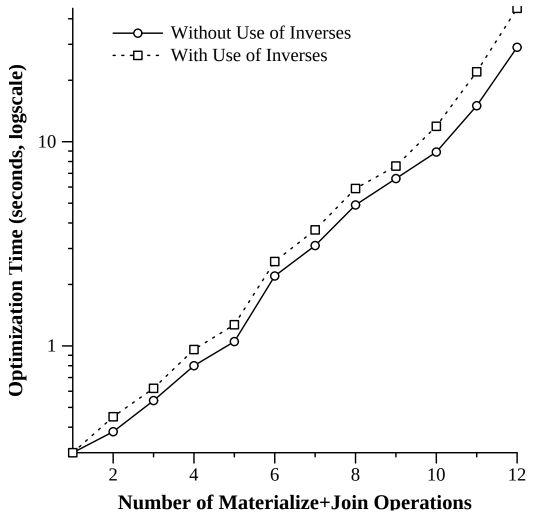


Figure 30: Use of Inverse Links when all references have inverses: Optimization Times (Log-scale)

Figure 31: Use of Inverse Links when all references have inverses: Improvement in Estimated Costs (Scaled)

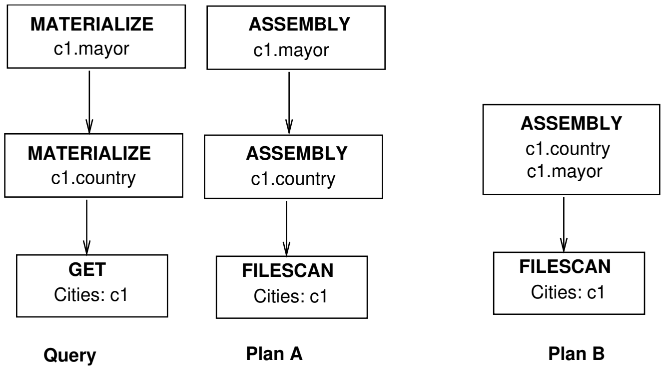


Figure 32: Collapsing Materializes

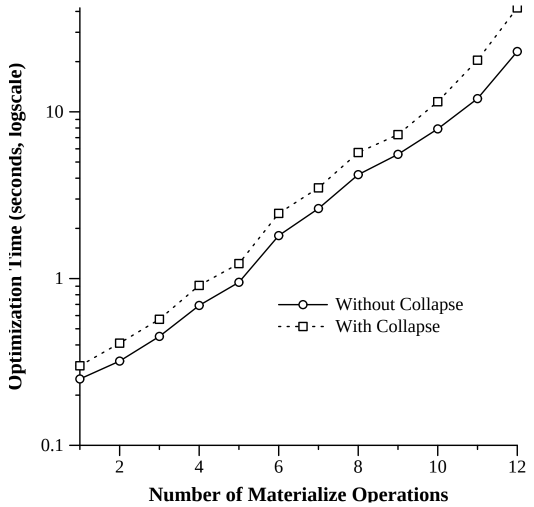


Figure 33: Collapsing Materializes: Optimization Times (Log-scale)

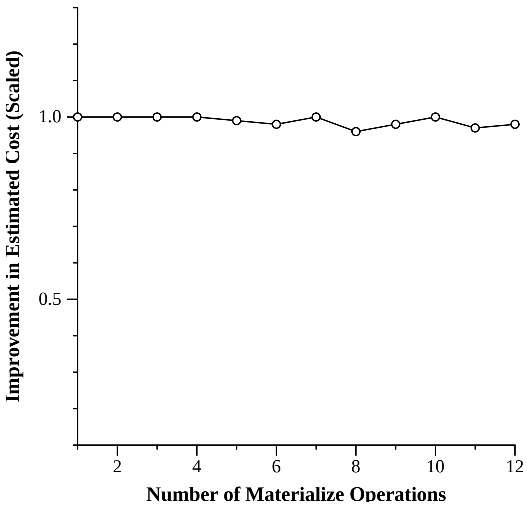


Figure 34: Collapsing Materializes: Improvement in Estimated Costs (Scaled)

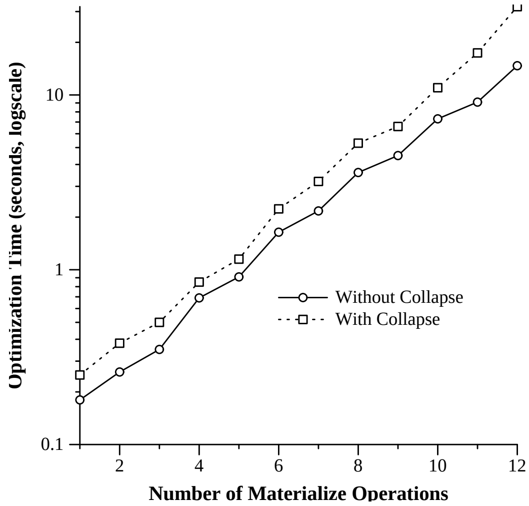


Figure 35: Collapsing Materializes in absence of Pointer Joins: Optimization Times (Log-scale)

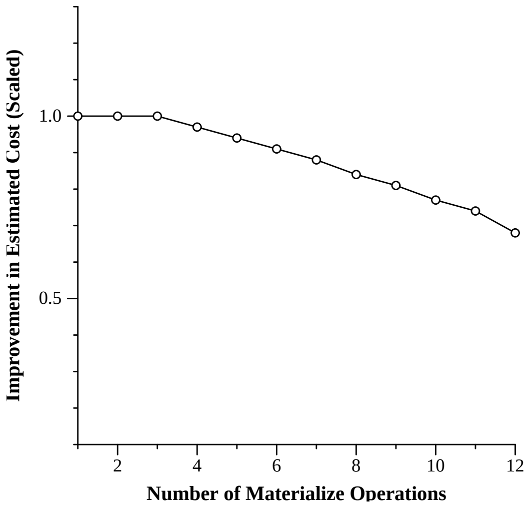


Figure 36: Collapsing Materializes in absence of Pointer Joins: Improvement in Estimated Costs (Scaled)

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 and 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 and 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 (upto 50%) improvement. In conclusion, this is an optimization that is very easy to implement, causes a modest increase in optimization time, and significant improvement in quality of plans produced.

Collapsing Multiple Materializes

A string of materialize operator applications can be collapsed into a single materialize operator application. Figures 32, 33 and 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 4 (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 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 and 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.

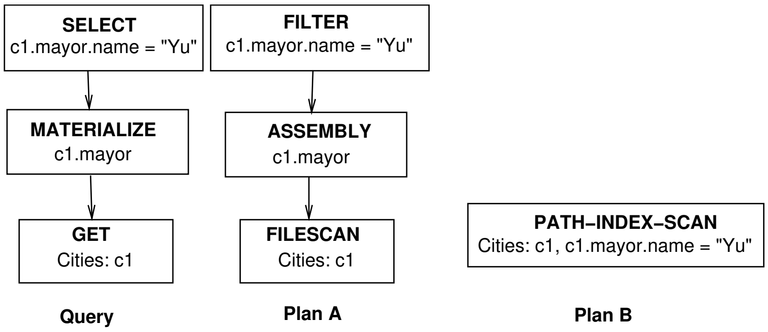


Figure 37: Use of Path Indices

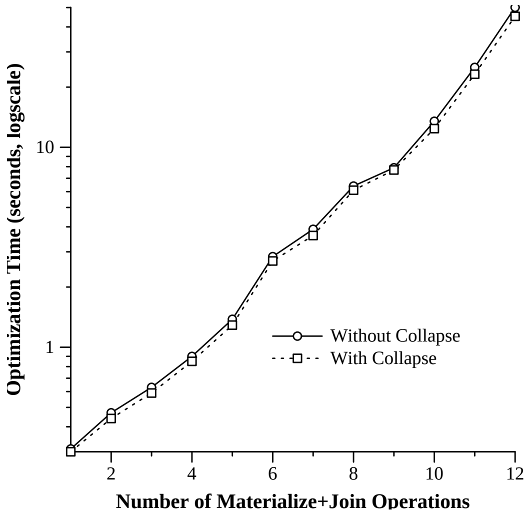


Figure 38: Use of Path Indices: Optimization Times (Log-scale)

Figure 39: Use of Path Indices: Improvement in Estimated Costs (Scaled)

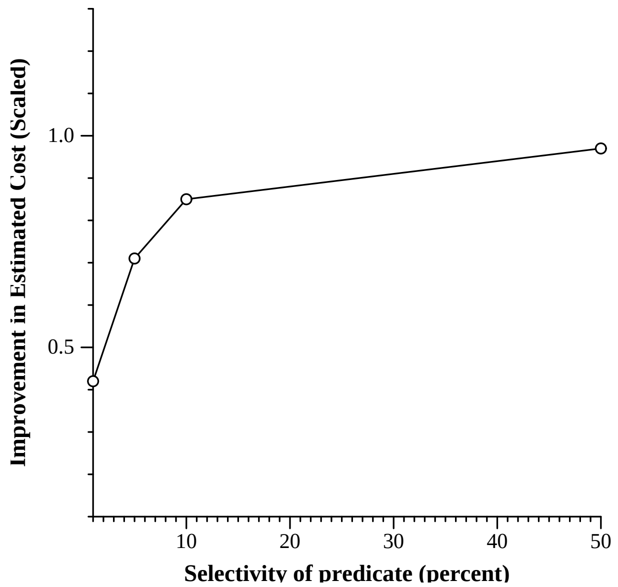


Figure 40: Path Indices: Effect of Selectivity on Estimated Costs (Scaled)

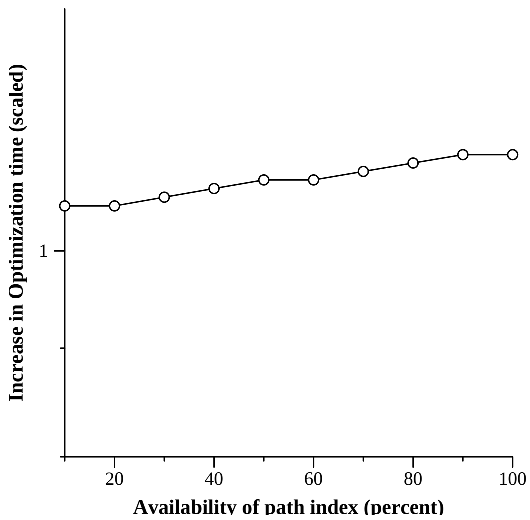


Figure 41: Path Indices (effect of availability): Increase in Optimization Times

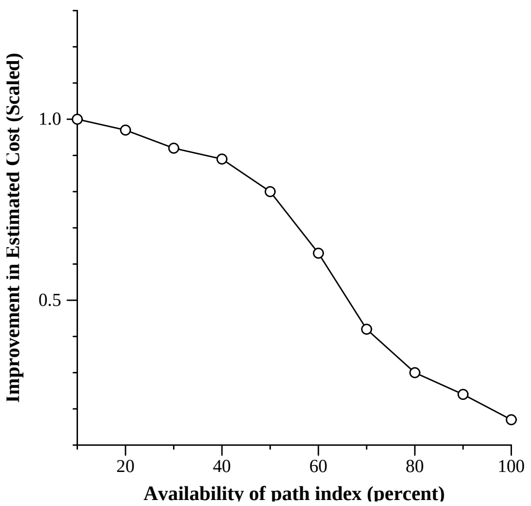


Figure 42: Path Indices (effect of availability): Improvement in Estimated Costs (Scaled)

Using 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[4]. 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 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 and 39 indicate that while the effect of this feature on the optimization time is negligible (less that 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 Figure 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%.

[4]There can be more than one materialize operations between the select and the filescan.

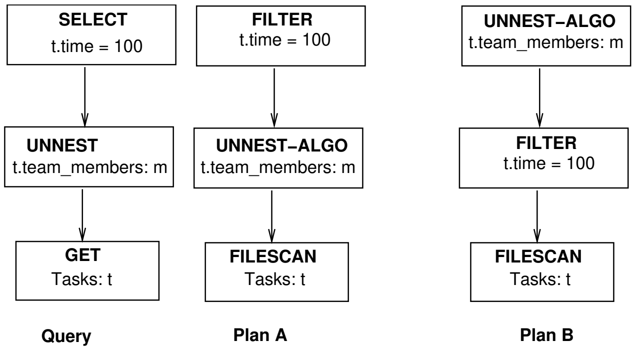


Figure 43: Unnest and Select Operators

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 the increase in optimization time depends upon the availability of path indexes. Every path index is an extra option to consider during optimization and hence it increases the size of the search space. However, the increase in optimization time 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.

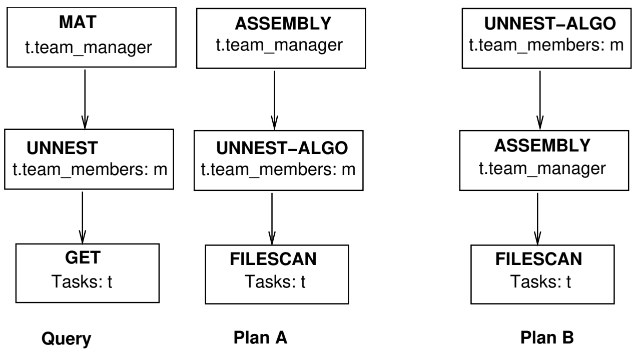


Figure 44: Unnest and Materialize Operators

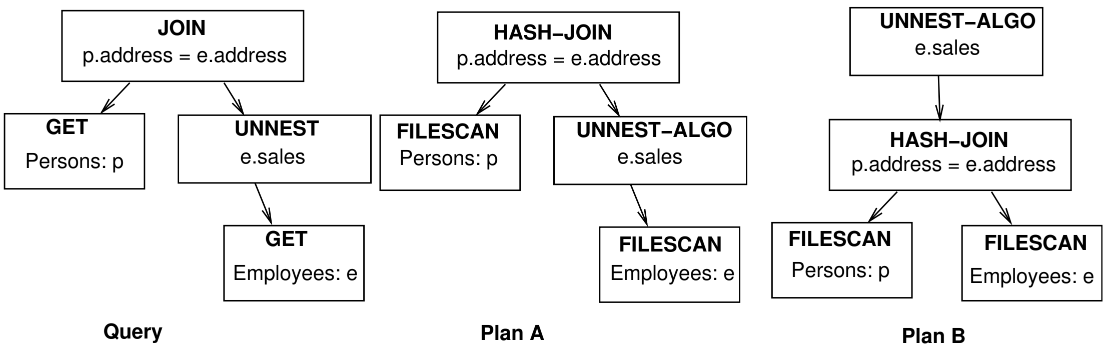


Figure 45: Unnest and Join Operators

3.2.2 Nesting and Unnesting Set-Valued Attributes

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, and 45 show the various kinds of query processing alternatives that need to be considered by the optimizer.

We derive an `UNNEST` operator class and a corresponding `UNNESTALGORITHM` class to represent unnesting of set-valued attributes.

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.

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.

The nest operator is the exact inverse of the unnest operator. However, we note that the nest operator can be considered to be a special case of an aggregate. Due to this, we do not describe the nest operator and the corresponding optimizations. All the optimizations described in Section 3.2.4 are applicable to the nest operator.

3.2.3 Optimizing Expensive Predicates

One of the consequences of allowing user-defined types and methods into a database system is that predicates in a query might invoke user-defined methods/functions that can be arbitrarily expensive. Due to this, the traditional optimization technique of pushing all selection predicates below joins isn't always the best possible plan.

The optimization of queries containing expensive predicates is another problem that must be tackled by object-relational database systems. Unfortunately, modifying the optimizer so that it considers all possible positionings of the expensive select predicates can be prohibitively expensive because that makes the optimization algorithm exponential in the number of selects[5]. [Hel94] describes some heuristics that can be used to optimize queries containing expensive predicates. Some of these heuristics are based on ideas and techniques first presented in [MS79, IK84, SI92, KBZ86].

These algorithms are based on the following observations.

- The more selective a predicate is, the greater are the benefits of pushing it down. This is because the applying the predicate causes the number of tuples to be reduced, reducing the future processing costs.

- The more expensive a predicate is, the greater are the benefits of pulling it up. This is because, pulling the predicate up causes it to be applied after other predicates and joins. Consequently, the expensive predicate is evaluated fewer times (since the other predicates and join filter out many tuples), reducing the cost of the query.

Based on these observations, the rank of a predicate is defined as

$$rank = \frac{selectivity - 1}{cost\_per\_tuple} \tag{1}$$

The rank of a predicate is used to determine whether to pull up a predicate or to push it down. The higher the rank of a predicate, the earlier it should be applied. In the PushDown algorithm, all selection predicates are pushed below all the joins, and then they are ordered according to rank. In the PullUp algorithm, all expensive selection predicates are pulled to the top of the query execution tree and are then

[5]Actually, exponential in the number of selects+join

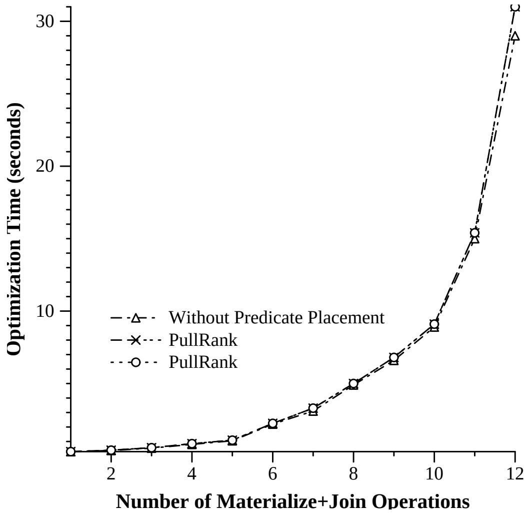


Figure 46: Expensive Predicates: Optimization Times

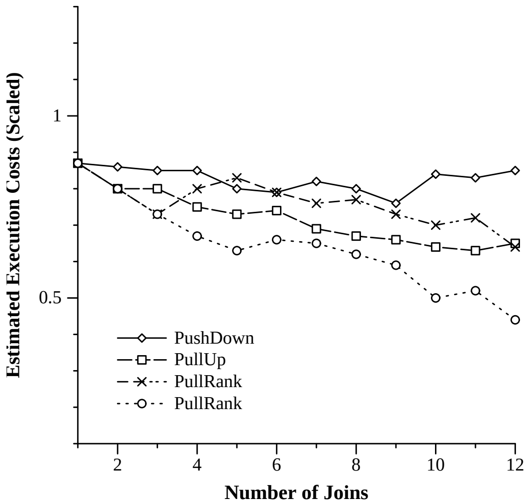


Figure 47: Expensive Predicates: Improvement in Estimated Costs (Scaled)

ordered according to rank. In the PullRank algorithm, an expensive selection predicate is pulled above a join only if the rank of the selection predicate is lower than the rank of the join predicate. This is a local decision based only on the ranks of that join predicate and that selection predicate. Finally, in the Predicate Migration algorithm an expensive selection predicate is pulled above a group of joins if (and only if) the combined rank of the group of joins is higher than the rank of the selection predicate. See [Hel94] for details of these algorithms.

Incorporating the PushDown and the PullUp heuristics into the optimizer required about a week of effort. The PullRank heuristic required about two weeks of work and the Predicate Migration algorithm required about a month (mainly debugging). Figures 46 and 47 show the results of our experiments with these techniques. These results basically confirm the results of [Hel94].

PullUp and PushDown are very easy to implement, but their performance is rather erratic and they often produce plans that are much worse than the plans produced by the other algorithms. PullRank is reasonably easy to implement and in most cases

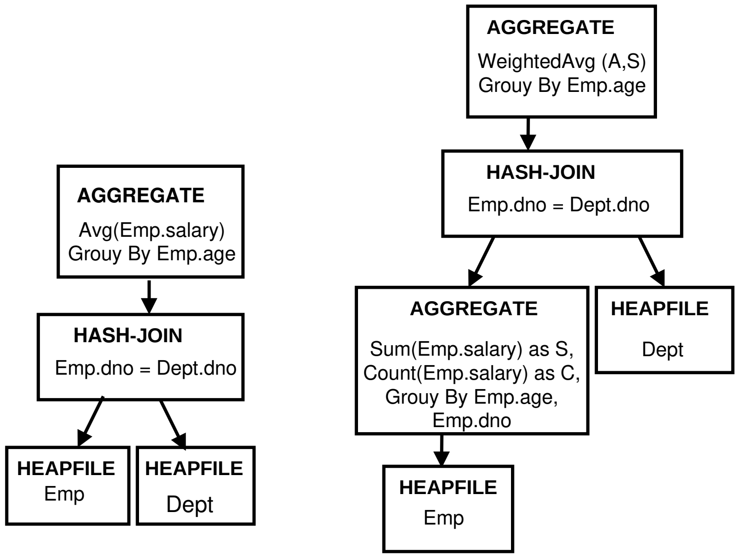


Figure 48: Splitting Aggregates

produces reasonable plans. For an Optimizer-Implementor who is willing to expend an moderate amount of effort, this is the best choice. However, if the database system makes heavy use of expensive predicates, and performance of queries containing expensive predicates critical, then the extra effort spent in implementing Predicate Migration is worth the trouble.

3.2.4 Optimizing Aggregates

Optimization of relational queries containing aggregates has received a lot of interest in recent times due to their extensive use in decision support systems. [YL95] and [CS96] represent the state-of-the-art in optimization of such queries. These systems describe how aggregates can be moved around in a query, or even split into two or more parts, leading to significant improvements in performance.

Figure 48 illustrates how an aggregate that follows a join split into two parts. This can only be done if the aggregate function happens to be decomposable.

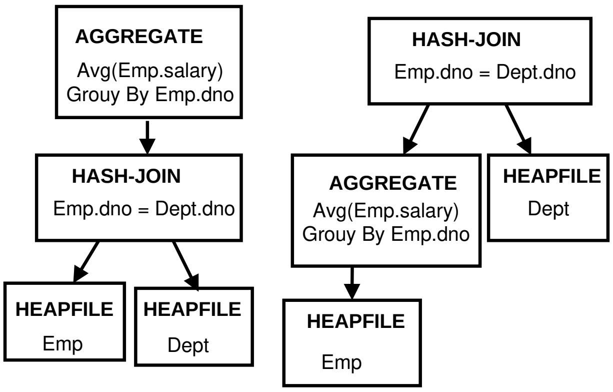


Figure 49: Pushing Aggregates below Joins

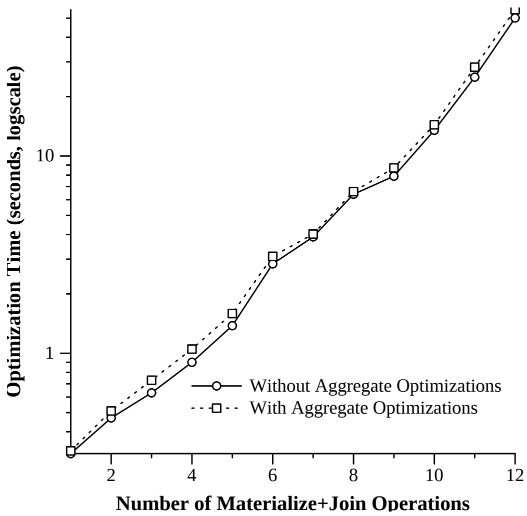


Figure 50: Optimizing Aggregates: Optimization Times (Log-scale)

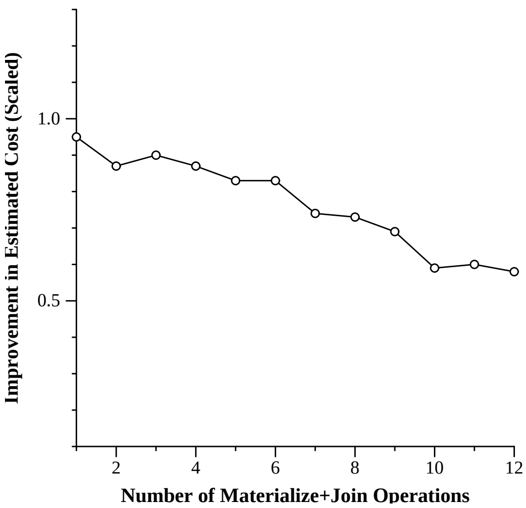


Figure 51: Optimizing Aggregates: Improvement in Estimated Costs (Scaled)

For the definition of a *decomposable* aggregate, consider the multiset S. Let F be an aggregate that operates on all the members of S and returns a single aggregate value A. This is denoted by $A = F(S)$. Suppose S can be represented as a union of a number of smaller multisets $S = S_1 \cup S_2 \cup \ldots \cup S_n$. Here $\cup$ is duplicate-preserving multi-set union.

Then, F is said to be decomposable if there exist a functions F_1 and F_2 such that

$$A = F(S) = F(S_1 \cup S_2 \ldots \cup S_n) = F_2(SA)$$

where SA is the set $\{F_1(S_1), F_1(S_2), \ldots, F_1(S_n)\}$.

In other words, F is decomposable if it can be broken up into two computations: first a partial grouping (using F_1) and then a final merging of groups (using F_2). Note: for full generality, we allow F_1 to actually return a composite tuple instead of a single value. Thus, in the example shown in Figure 48, the function F is "Avg", and the function $F_1(x)$ is actually a composite function that returns the composite value "[Sum(x), Count(x)]". The function F_2 takes the sums and counts produced by F_1 and produces their weighted average.

As an added optimization, we note that if the grouping attribute happens to be the same as the join attribute, and the join is a foreign key equi-join, then the second aggregate can be avoided. Figure 49 shows how an aggregate that follows a join can be moved below the join, assuming `Dept.dno` is a primary key of the `Dept` relation.

[YL95] describes the necessary and sufficient conditions under which splitting and moving aggregates is legal.

The situation gets complicated in object-relational systems that allow users to define arbitrary aggregate functions. [SAH87] describes how a user can define an arbitrary aggregate function in terms of three user-defined functions. These user-defined aggregates can then be used in queries wherever the traditional SQL aggregates can appear.

Some more information is needed to be able to use the aggregate move-around techniques described above. Specifically, it is necessary to know whether a given aggregate function F is decomposable or not. And if it is decomposable, the optimizer needs to know what aggregate function can be used as the function F_1 and F_2 for merging partial groups. Once this information is registered in the system catalogs for every user-defined aggregate function in the system, the techniques of [YL95] and [CS96] can be applied directly to user-defined aggregates in object-relational algebras.

To incorporate aggregate move-around techniques into an transformative optimizer, the transformations shown in the figure can be directly encoded as transformative `TREETOTREEGENERATORs`. For the bottom-up System-R search strategy, we have to implement a `AGGREGATEEXPAND` class derived from the `TREETOTREEGENERATOR` class. The `APPLY` method for this class essentially duplicates the algorithm described in [CS96]. Due to the very subtle semantics of the transformations involving the aggregates, debugging this was a difficult task, and incorporating this optimization into the optimizer required about two months of effort.

Figures 50 and 51 show the effect of these optimization on the optimizer. The increase in optimization time is modest while the improvement in estimated cost is as high as 50% in some cases. It requires a significant effort to incorporate this optimization into an optimizer, but the benefits can be significant if the query involves multiple joins and expensive aggregates. Incorporating this optimization into an optimizer is recommended to an Optimizer-Implementor if the system is likely to have a lot of complex decision support queries involving aggregates.

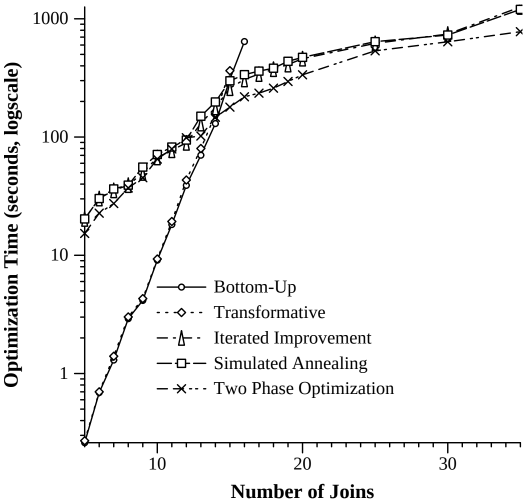


Figure 52: Comparing Search Strategies (Object-Relational): Optimization Times (Log-scale)

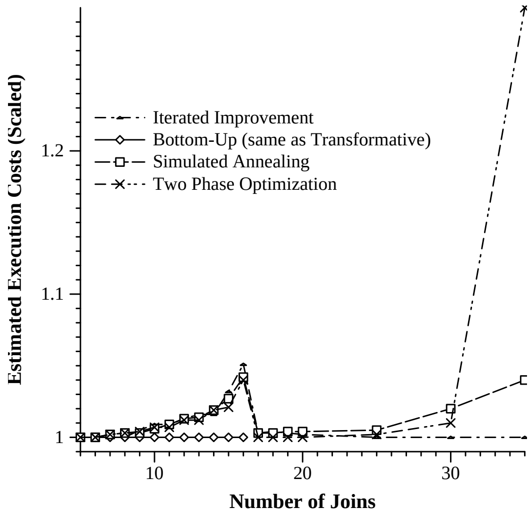


Figure 53: Comparing Search Strategies (Object-Relational): Estimated Costs (Scaled)

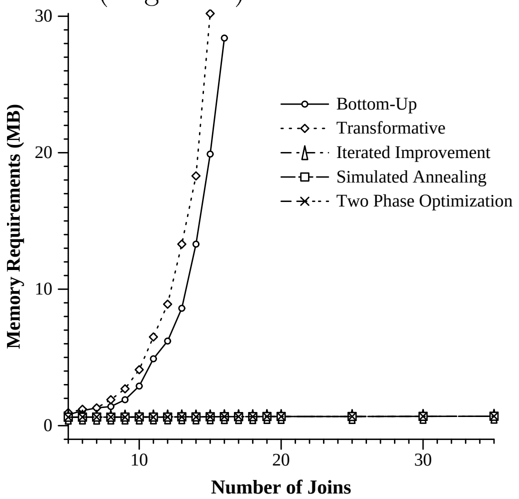


Figure 54: Comparing Search Strategies (Object-Relational): Memory Requirements

3.2.5 Effect upon Search Strategies

In Section 3.1.4 we performed a comparative study of the various search strategies for optimization of relational queriesl. We repeated the same experiment for the object-relational queries, with all the features described in the preceeding sections turned "on". Figures 52, 53 and 54 represent the results of that study. We notice a few changes in the results. For queries with less than 15 join/materialize operations, the System-R/Volcano search strategies continue to be the best choices. After that, the randomized strategies give better results. However, we notice that for queries containing more than 25 joins, Iterated Improvement gives better plans results than either Simulated Annealing or 2PO. This result is different from that of Section 3.1.4, indicating that a change in the query algebra affects the relative performace of the randomized algorithms.

3.3 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 the System-R style bottom-up construction 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.

Chapter 4

Debugging or Over-riding Faulty Optimizers

One of the most frustrating experiences for an Optimizer-Implementor is debugging an optimizer that produces sub-optimal query execution plans. Since the optimizer examines thousands, or even millions of plan alternatives during the optimization of a single query, it becomes very difficult to track down the cause of the error. Trying to make sense of the mass of information that is available is very difficult and time-consuming for small queries, and almost impossible for larger ones.

Based on our experiences, we believe that debugging the optimizer represents a very large percentage of the time spent by an Optimizer-Implementor in implementing a query optimizer. However, we are not aware of any work that addresses this issue. In this section, we describe how the optimizer itself can be used to greatly simplify the task of debugging the optimizer.

First we consider the simple problem of an optimizer that produces a sub-optimal query execution plan (*plan A*), whereas another plan (*plan B*) is known to be optimal. The job of the Optimizer-Implementor is to determine, with the aid of the optimizer, why *Plan B* was not produced. In the simplest case, *Plan B* is supplied to the optimizer as an input. During the course of optimization, the query optimizer compares *Plan B* with the various plans that it produces while exploring the search space. We describe in Section 4.1 how the optimizer can then indicate to the Optimizer-Implementor the most likely sources of bugs.

In reality, the situation is more complicated. Although an Optimizer-Implementor might suspect that the optimizer is producing sub-optimal plans, it is not always easy to find an alternative plan (*i.e.Plan B*) that is better than the plan produced by the optimizer. In Section 4.3 we describe how this problem is tackled. The Optimizer-Implementor provides the optimizer hints about the possible characteristics of the optimal plan, and lets the optimizer find a better plan. We describe *Partial Plan Specifications*, a flexible way to specify some characteristics of a query execution plan. We then describe how the optimizer can be debugged using *Partial Plan Specifications.*

All the debugging techniques described in Sections 4.1 and 4.3 apply to optimizers that use the System-R style dynamic programming search strategy. Since these techniques are not general enough to be directly applicable to different search strategies, we have chosen to exclude them from the general OPT++ framework. However, the algorithms described here are incorporated into the bottom-up search strategy available with OPT++. Hence, an Optimizer-Implementor can very easily modify a System-R style optimizer (built using OPT++) to incorporate the debugging techniques described in this chapter. In Section 4.5 we discuss how these techniques can be of some use for debugging optimizers that use other search strategies.

Partial Plan Specifications can also be used to over-ride an optimizer and force it to produce specific query execution plans that satisfy given constraints. In Section 4.4 we discuss why this would be a useful feature for database users and developers.

4.1 Optimizer Aided Debugging

In this section we will describe the basic ideas behind the optimizer-aided debugging technique through the use of an example. We describe how a System-R style dynamic

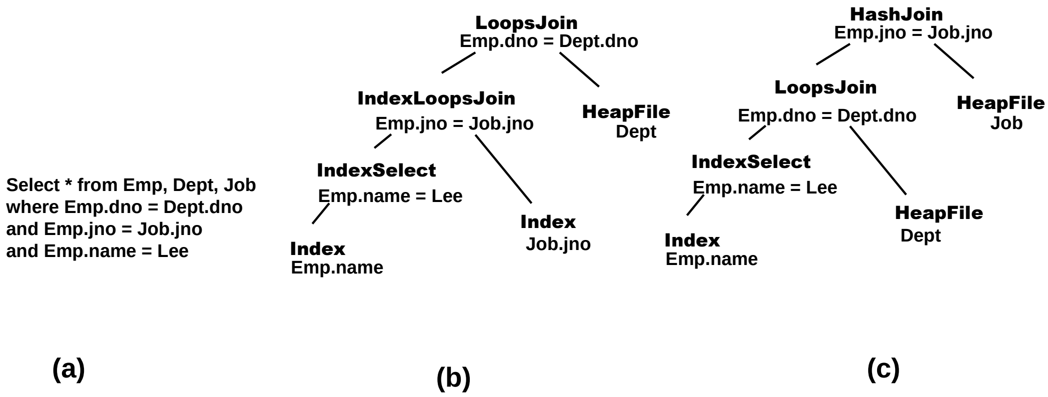


Figure 55: A Sub-Optimal Query Execution Plan

programming search strategy can be easily modified to give an Optimizer-Implementor useful We will describe how the Optimizer-Implementor uses hints from the optimizer to track down the bug that causes the optimizer to produce sub-optimal plans. In the rest of this section, will use the query shown in Figure 55(a) and an example. Suppose Figure 55(b) is the plan produced by the optimizer for this query. The Optimizer-Implementor believes that this plan is sub-optimal, and the plan shown in Figure 55(c) is the real optimal plan.

The Optimizer-Implementor specifies this “expected” plan as an input to the optimizer and tries to determine why this plan was not produced. We now describe how the System-R style dynamic programming search strategy is modified to use information about the “expected” plan and guide the Optimizer-Implementor in tracking down the source of the bug. While the query is being optimized, the optimizer constantly compares the various plans (sub-plans) being generated with the “expected plan” that is specified to it. The main possible outcomes of this analysis are:

- A sub-tree of the “expected” plan got produced, but got pruned out. This could have two causes:
  - A bug in the cost estimations for these plans causes the “expected” plan to

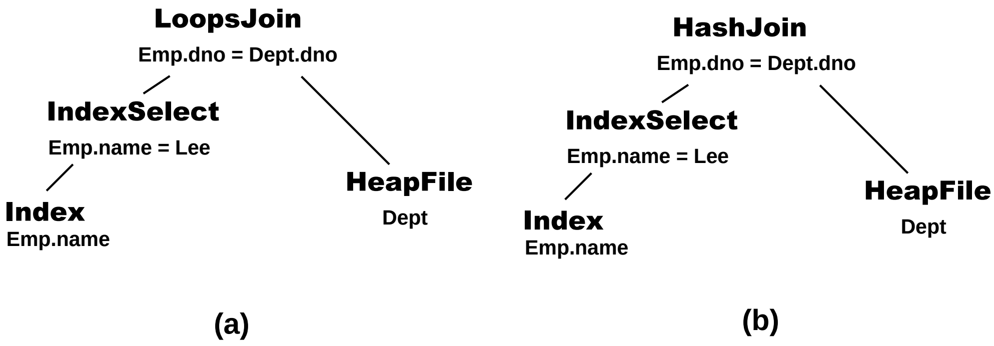


Figure 56: A bug in cost estimation

seem more expensive (*cost-estimation anomaly*), or

- There is no bug in the optimizer, and the plan produced by the optimizer is indeed cheaper than the "expected" plan (*no anomaly*).

- Due to a bug in the enumeration logic, the "expected" plan never got produced. This could happen if

  - An operator tree corresponding to a sub-tree of the "expected" plan did not get produced. This indicates a bug in some `TREETOTREEGENERATOR` class (*tree generation anomaly*).

  - An access plan corresponding to a sub-tree of the "expected" plan did not get produced, even though the corresponding operator tree got produced. This indicates a bug in some `TREETOPLANGENERATOR` class (*plan generation anomaly*).

In the rest of this section, we consider these anomalies in greater detail.

4.1.1 The Cost-Estimation Anomaly

Consider the sub-plan shown in Figure 56(a). If this sub-plan is pruned out by the optimizer during the optimization process in favor of another plan (for example, Figure 56(b)), then the "expected" plan cannot be produced by the optimizer. In this case, there is some bug in the cost estimation of these plans which causes the optimal one to seem more expensive and get pruned out. The optimizer can now output this information to the Optimizer-Implementor. The Optimizer-Implementor can use a debugger to step through the cost estimation functions while they are being used to estimate the cost of these two sub-plans and can then quickly determine what part of the code caused the error in cost estimation. The error could either be a mistake in the code for some cost function, or it could be the result of some statistics from the system catalogs being incorrect or out-of-date.

Another possibility that we have frequently encountered in such cases is that there is no bug, and the optimizer was right after all! Sometimes, "intuitively" the plan produced by the optimizer seems "obviously" sub-optimal and the "expected" execution plan "seems" to be a better plan. However, after stepping through the cost functions for the two sub-plans indicated by the optimizer, and looking at the estimates in details, it turns out that due to the consequences of some details that were missed by the Optimizer-Implementor, the plan produced by the optimizer is indeed better than the "obvious" choice.

4.1.2 The Plan Generation Anomaly

Consider the sub-plan shown in Figure 57(a). If the "expected" plan needs to be produced by the optimizer as the optimal plan, then it is necessary that the sub-plan shown here should be produced by the optimizer while it explores the search

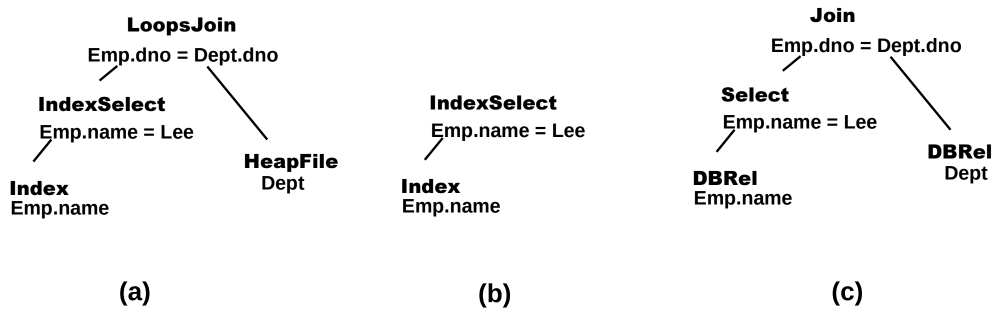


Figure 57: An access plan does not get produced

space. If, during the course of optimization, the optimizer discovers that the sub-plan shown never gets produced, this clearly indicates a bug in the enumeration logic of the optimizer.

Let Figure 57(b) be a sub-plan and Figure 57(c) be an operator tree that was produced by the optimizer during optimization. It should be clear that if both of these were produced by the optimizer, then the sub-plan in Figure 57(a) should have been produced by the application of some `TREETOPLANGENERATOR::APPLY` method. If sub-plan 57(a) was never produced by the optimizer, the bug is clearly in the `TREETO-PLANGENERATOR::APPLY` method.

Now it would be very useful if, at the end of optimization, the optimizer can output a diagnostic indicating that sub-plan 57(a) was never produced during optimization even though 57(b) and (c) were produced. The Optimizer-Implementor knows that the access plan should have been produced by the `HASHJOINGENERATOR::APPLY` method. From this the Optimizer-Implementor can easily determine that the bug is either in the `HASHJOINGENERATOR::CANBEAPPLIED` method or the `HASHJOINGENERATOR::APPLY` method.

This information can significantly simplifies the job of debugging the optimizer.

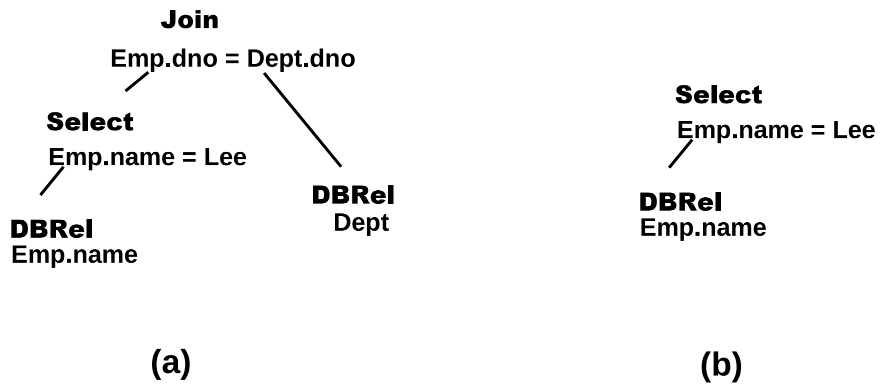


Figure 58: An operator tree does not get produced

The Optimizer-Implementor can now re-run the optimizer with the same input query and use a standard source code debugger to set a breakpoint at the `CANBEAPPLIED` and the `APPLY` methods of the `HASHJOINGENERATOR` class and suspend execution when either of them is being applied to the operator tree of Figure 57(c). Then he can step through the execution of these functions and find out why the sub-plan of Figure 57(a) was not produced.

Detection of plan generation anomalies in the general case is described in Section 4.2.

4.1.3 A Tree Generation Anomaly

There is another problem that could occur during the optimization of the query considered in the previous sub-section. It is possible that the operator tree shown in Figure 57(c) never gets produced. Obviously, if the operator tree does not get produced, the access plan 57(a) cannot be produced.

Consider the operator trees shown in Figure 58. Note that Figure 58(a) is just a reproduction of Figure 57(c). If during the course of optimization, the optimizer discovers that the operator tree shown in Figure 58(b) gets produced, but 58(a) never gets produced, it is clear that when the `TREETOTREEGENERATOR::APPLY` method was invoked on operator tree 58(b), it never produced operator tree 58(a). Once the optimizer determines this, it can indicate this to the Optimizer-Implementor. The Optimizer-Implementor knows that the operator tree should have been produced by the `JOINEXPAND::APPLY` method. From this the Optimizer-Implementor can easily determine that the bug is either in the `JOINEXPAND::CANBEAPPLIED` method or the `JOINEXPAND::APPLY` method.

Armed with this information, the Optimizer-Implementor can re-run the optimizer with the same input query and use a standard source code debugger to set a breakpoint at the `CANBEAPPLIED` and the `APPLY` methods of the `JOINEXPAND` class and suspend execution when either of them is being applied to the operator tree of Figure 58(b). He can then step through the execution of these functions and find out why the operator tree of Figure 58(a) was not produced.

4.2 Detecting Anomalies

In this section, we describe an algorithm that can be used to efficiently detect cost-estimation, and tree and plan generation anomalies.

In the beginning, the optimizer is provided with an "expected" plan that it is expected to produce. While optimization is in progress, the following three types of events are monitored.

- Every time some algorithm instance (that represents an access plan) is produced, it is compared against the given "expected" access plan. If it matches some node of the "expected" access plan, then that node is marked as "plan-produced".

- Every time some operator instance (representing an operator tree) is produced, it is compared against the "expected" access plan. If it matches some node of the "expected" access plan, then that node is marked as "tree-produced".

- Every time an algorithm instance is pruned, it is compared against the "expected" access plan. If it matches some node of the "expected" access plan, that node is marked "pruned".

If at the end of optimization, there is a node of the "expected" access plan that is marked "pruned", it represents a sub-plan that was produced by the optimizer but deleted as sub-optimal. The lowest such node in the tree[1] indicates the location of a cost-estimation anomaly. If there is no cost-estimation anomaly, then the lowest node in the tree that is marked "tree-produced" but is not marked "plan-produced" is the location of a plan generation anomaly. This is because it represents a node for which an operator tree was produced, but an access plan wasn't. Finally, in the absense of plan generation anomalies, the lowest node in the tree that is neither marked "tree-produced" nor marked 'plan-produced" is the location of a tree generation anomaly. If the "expected" plan is not produced by the optimizer as the optimal plan, then at least one of the above anomalies will necessarily have occured.

4.3 Debugging based on Incomplete Specifications

In the previous section, we assumed that when an optimizer produced a sub-optimal query execution plan, the Optimizer-Implementor was aware of a better plan. This is not always desirable. First, fully specifying an "expected" plan to the optimizer is a cumbersome and time-consuming task. More importantly, the Optimizer-Implementor often does not know any better plan. Most of the time, the Optimizer-Implementor

[1]Specifically, the first such node that is encountered while doing a pre-order traversal of the tree.

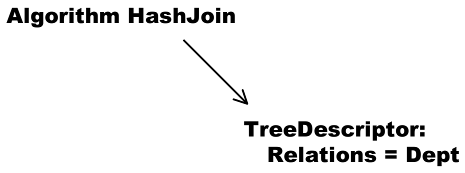


Figure 59: A *Partial Plan Specification*

only has an instinctive feeling that the plan produced by the optimizer is sub-optimal.

Under these circumstances, we can simplify the job of the Optimizer-Implementor by allowing him to specify only certain "expected" characteristics of the "expected" plan, instead of specifying the complete plan. Using a similar procedure as described in the previous section, it can be determined why a plan with the "expected" characteristics was not produced by the optimizer.

Next, we describe *Partial Plan Specifications*, a flexible facility for partially specifying "expected" characteristics of a plan. We then describe how a *Partial Plan Specification* can be used instead of the "expected" plan described in the previous section.

4.3.1 *Partial Plan Specifications*

A *Partial Plan Specification* is something that can be used to partially describe a query execution plan. A *Partial Plan Specification* can be thought of as a regular expression pattern such that any given query execution plan will either "match" the pattern, or not match it. Specifically, a *Partial Plan Specification* is a tree that represents a part of a query execution plan. Consider Figure 59. This represents a partial specification of a query execution plan and specifies *"any plan that contains a hash-join with the relation* `Dept` *as its right input"*.

In general, a *Partial Plan Specification* is a tree of *Plan Specification Nodes*. Each

Plan Specification Node consists of a constraint like `relations = Dept` or `operator = hash-join`. A query execution plan "matches" a *Partial Plan Specification* if the *Partial Plan Specification* can be mapped on to a sub-tree if the query execution plan in such a way that the constraint in each *Plan Specification Node* of the *Partial Plan Specification* is satisfied by the corresponding node in the query execution plan[2].

The constraint in a *Plan Specification Node* can either be of the form `operator = NAME`, or `algorithm = NAME`, or they can refer to properties of the `TREEDESCRIPTOR`, or the `PLANDESCRIPTOR` of that node. `relations = Emp, Dept` is an example of a constraint based on properties of a `TREEDESCRIPTOR`. This constraint is matched by any node that contains exactly the relations `Emp` and `Dept`. Such a constraint may also use the `<` or `>` operators instead of the '=' operator. For example, it could be `relations > Emp, Dept`, in which case it is satisfied by any node that contains both the `Emp` and `Dept` relations, and possibly more. Conversely the constraint `relations < Emp, Dept` is satisfied by any node that might or might not contain the `Emp` and `Dept` relations, but contains no other relations. An example of a constraint based on the `PLANDESCRIPTOR` is `sortorder = Emp.name`.

A complex constraint can also be arbitrarily composed from simple constraints by using boolean connectives like `and`, `or` and `not`. Thus, `algorithm = indexscan and relations = Emp` matches any plan that uses an indexed scan on the Emp relation.

Figure 60 shows some examples of *Partial Plan Specifications*. Figure 60(a) shows a constraint that is matched by any plan that scans the `Dept` relation using an index. Figure 60(b) specifies a join order for a query. Finally, Figure 60(c) shows a more

[2]We note that it is possible to conceive of more general *Partial Plan Specifications*. One generalization is to allow specification of some condition that involves to remote parts of a query execution plan (that are not necessarily part of the same sub-tree). Another generalization is to allow the specification of tree patterns based on some sort of regular expression grammer. These generalizations significantly increase the complexity of the code, as well as the space requirement of the optimizer. Due to this, we felt that generalizing the *Partial Plan Specification* was not worth the increase in complexity.

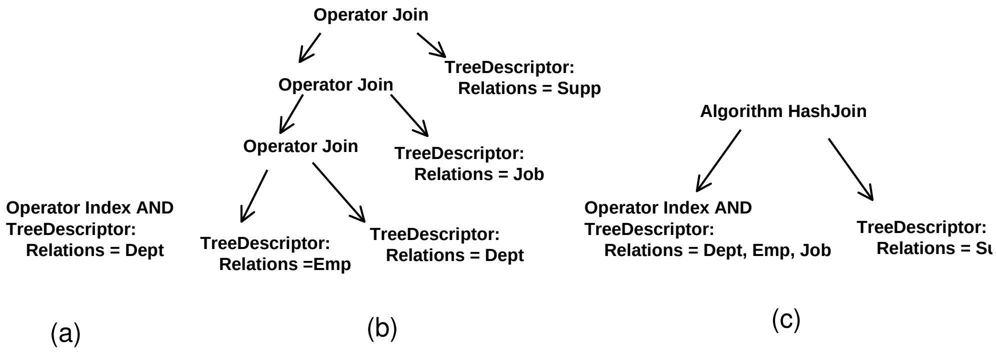


Figure 60: Examples of *Partial Plan Specifications*

general constraint that indicates that the `Supp` relation must be joined to the composite of the `Dept`, `Emp` and `Job` relations using a hash-join.

We note that this scheme is easily extensible as far as the Logical and Physical Query Algebra are concerned. The `PLANDESCRIPTOR` class and all its member functions and methods are implemented by the Optimizer-Implementor. If the Optimizer-Implementor wishes to use constraints based on the `PLANDESCRIPTOR` of some node in an access plan, he is required to implement a `MATCH` method on the `PLANDESCRIPTOR` class. This method takes a constraint and returns `TRUE` if the `PLANDESCRIPTOR` satisfies that constraint, and false otherwise. A constraint based on the `PLANDESCRIPTOR` are not interpreted by OPT++ at all. It is directly passed on to the `MATCH` method, and the return value is used to determine whether the plan matches that constraint. In a similar way, constraints based on the `TREEDESCRIPTOR` are handled by the `TREEDESCRIPTOR::MATCH` method.

4.3.2 Optimizer Aided Debugging Using *Partial Plan Specifications*

It is quite easy to extend the optimizer-aided debugging technique of Section 4.1 to use *Partial Plan Specifications* instead of a complete plan. Whenever a new operator or algorithm instance is created during the optimization it is compared with the given *Partial Plan Specification.* Whenever there is a match, the corresponding nodes are marked as "tree-produced" or "plan-produced". Whenever an algorithm instance that matches the *Partial Plan Specification* is pruned out in favor of a plan that does not match, the corresponding node is marked "pruned". Detection of anomalies based on these markings is done exactly as before.

Thus the Optimizer-Implementor can use any *Partial Plan Specification* to narrow down the search for the source of bugs in the optimizer. Consider the case where the Optimizer-Implementor does not know what the correct optimal plan for a particular query is, but only has a vague idea of what characteristics the optimal plan might have, (for example, "it probably uses an index for `Dept.name` and a `hash-join` for joining `Emp` and `Dept`"). He can then construct a *Partial Plan Specification* based on those characteristics and use that to debug the optimizer. It might turn out that the best plan that satisfies the given characteristics is worse than the plan produced by the optimizer. In that case, the Optimizer-Implementor will detect this while comparing the costs of the plans produced, and can repeat this process after modifying the *Partial Plan Specification.* This process can be repeated until the bug is found, or the Optimizer-Implementor is convinced that the optimizer is producing the optimal plan.

4.4 Over-riding an Optimizer

Even if an optimizer is "bug-free", there are a number of situations in which there is still a need to disregard the plan produced by the optimizer and force it to generate a plan having some specified characteristics.

The circumstances under which this might be desirable are

- Sometimes, due to incomplete information, the optimizer will produce a bad plan for some query. If the user (*i.e.*, the end user, not the Optimizer-Implementor) happens to have more information about the data in the database then a hint could be given to the optimizer, helping it to produce the right plan. A number of commercial systems have extended their query languages to allow the user to specify such hints. This indicates that there is a demand in the real world for optimizers that take hints.

- For validation of the cost-model during the development of the optimizer, it is desirable to be able to execute different plans for a given query and compare costs. In other words, if the optimizer estimates that plan B is more expensive than plan A, then we would like to actually run both plan A and plan B and compare their execution times.

- Hints are also helpful for debugging and tuning the execution engine. Often, during the development of the database, it is desirable to be able to execute specific plans so as to check certain functionality in the back-end, to debug parts of the execution engine code, and to profile and tune the performance of the database.

All these can be easily achieved using the *Partial Plan Specifications* described in the previous section. All a user needs to do is supply the optimizer with a *Partial Plan*

Specification indicating to it the characteristics of the "expected" plan. For example, the user might specify that a certain relation be scanned using a particular index; or that a certain join be implemented using nested-loops join. Or he could specify the join order to be used for a subset of the relations in the query.

The search strategy of the optimizer has to be modified to incorporate this functionality into it. Specifically the pruning mechanism would have to be modified so that it tries to produce plans that match the given *Partial Plan Specification*. Every plan that is produced by the optimizer during the course of optimization is marked with a boolean flag indicating whether or not it matches the given *Partial Plan Specification*. When comparing two equivalent plans, the pruning mechanism considers this flag as follows:

- if one of the plans matches the *Partial Plan Specification* and the other does not, then the latter is considered more expensive and is pruned out in favor of the former.

- if both plans match or do not match the *Partial Plan Specification*, then they are compared in the usual fashion, based on their estimated costs.

This ensures that if there exists one or more plans in the search space that match the given *Partial Plan Specification*, the cheapest of these plans will be produced as the result of optimization. If no such plan exists, then the optimizer just functions like a regular optimizer.

4.5 Debugging other Search Strategies

Unfortunately, the task of debugging an optimizer is intimately tied to the search strategy used by the optimizer. Due to this, the techniques used as a debugging aid have to be tailored to a specific search strategy. Since the System-R style Dynamic Programming search strategy is the most widely used search strategy, we have concentrated mainly on this strategy. The techniques described in this chapter do not work with any other search strategy. It is unclear whether these or any other techniques can be extended for debugging optimizers built using any other other search strategies. We believe that the simplicity of the bottom-up dynamic programming strategy made it easy for us to come up with this algorithm for debugging it.

Debugging an optimizer built using other search strategies remains a problem. However, the problem is slightly alleviated due to the following observation. When OPT++ is used to implement two different optimizers, one based on the bottom-up dynamic programming strategy, and one based on another strategy, a lot of the code is shared between them. All code other than the search strategy code and the `TREETOTREEGENERATORs` is shared (the entire the Algebra component, the `TREETOPLANGENERATORs` and the `PLANTOPLANGENERATORs`). In case the search strategy is not a transformative strategy like the Volcano style strategy, then most of the `TREETOTREEGENERATORs` are also shared.

In this case, a lot of the debugging burden can be reduced by first debugging the optimizer using the bottom-up dynamic programming strategy. This allows the Optimizer-Implementor to find all the bugs in the code using the optimizer-aided technique described in this chapter. Then, the Optimizer-Implementor switches over to the other search strategy. At this point the only code that is different is the `TREETOTREEGENERATORs` and the search strategy code. Further, the search strategy is written only once and after it has been tested and debugged, it is not changed when an optimizer for a specific system is implemented. Hence, the Optimizer-Implementor can be fairly confident that if the optimizer does not behave as expected, the bug lies somewhere in the `TREETOTREEGENERATORs` code. This considerably reduces the time required for debugging the optimizer.

Chapter 5

Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans

5.1 The Need for Dynamic Re-Optimization

In this chapter, we describe the *Dynamic Re-Optimization* algorithm that can detect the sub-optimality of a query plan while executing the query in order to re-optimize the query and improve its performance. We describe the *Dynamic Re-Optimization* algorithm that deals with sub-optimal plans produced by a query optimizer. Our approach essentially consists of detecting sub-optimality of a query execution plan during query execution in order to find ways in which the execution of such a query can be speeded up. During query optimization, the plan produced by the query optimizer is annotated with the various estimates and statistics used by the optimizer. Actual statistics are collected at query execution time. These observed statistics are compared against the estimated statistics and the difference is taken as an indicator of whether the query-execution plan is sub-optimal. The new statistics (much more accurate than the initial optimizer estimates) can now be used to optimize the execution of the remainder of the query.

Collection of statistics at run-time can significantly slow down the execution of a query. Further, re-optimizing part of the query and modifying the query execution plan at run-time also incurs overheads. This can actually cause the performance of a query to deteriorate instead of improving. To prevent such problems, we use hints from the optimizer to determine the most strategic places in the query where statistics should be collected, and to determine the conditions under which to re-optimize a query.

Our approach is quite different from the competition model proposed by Antoshenkov [Ant93, Ant96], the dynamic query plans of [GW89] and [GC94], or the parametric query optimization algorithms proposed in [INSS92]. The differences between these algorithms and our approach are further described in Section 6 when we discuss related work.

We first describe the details of this algorithm in Section 5.2. Then, in Section 5.3, we describe an implementation of the algorithm in the Paradise database system, and report the results of a performance study that validates our algorithm.

5.2 The *Dynamic Re-Optimization* Algorithm

The *Dynamic Re-Optimization* algorithm tries to detect sub-optimality of a query execution plan while the query is being executed. If a query execution plan is believed to be sub-optimal, it dynamically changes the execution plan of the remainder of the query (the part that hasn't been executed yet) leading to an improvement in performance.

These are the salient features of the algorithm:

1. **(Annotated) Query Execution Plans**: We assume that a conventional query optimizer is used to produce a query execution plan for a given query. The

only requirement on the plan generated by the query optimizer is that the plan produced by the optimizer should include information about the optimizer's estimates of the sizes of all the intermediate results in the query, and the execution cost/time for each operator in the query. We refer to such a plan as an *annotated query execution plan* in the remainder of this chapter.

2. **Runtime Collection of Statistics**: At specific intermediate points in the query, various statistics are collected during query execution. These statistics are used to obtain improved estimates of the sizes of intermediate results and execution costs. These improved estimates can be compared against the optimizer's estimates to detect sub-optimality of the query execution plan.

3. **Dynamic Resource Re-allocation**: The improved estimates are used to improve the allocation of shared resources (like memory) to the various operators of the query, leading to improved performance.

4. **Query Plan Modification**: The improved estimates are also used to determine whether the remainder of the query execution plan would benefit from re-optimization. If so, then the remainder of the query is re-optimized.

5. **Keeping Overheads Low**: Collection of statistics at query execution time can result in a significant overhead if used indiscriminately. To prevent this from happening, at query optimization time, the most effective points to collect statistics are determined, and statistic collection operators are inserted into the query execution plan at only those points.

In the remainder of this section, we describe each of the above items in detail. We end the section with an overview of the whole dynamic re-optimization process, and how it all fits together.

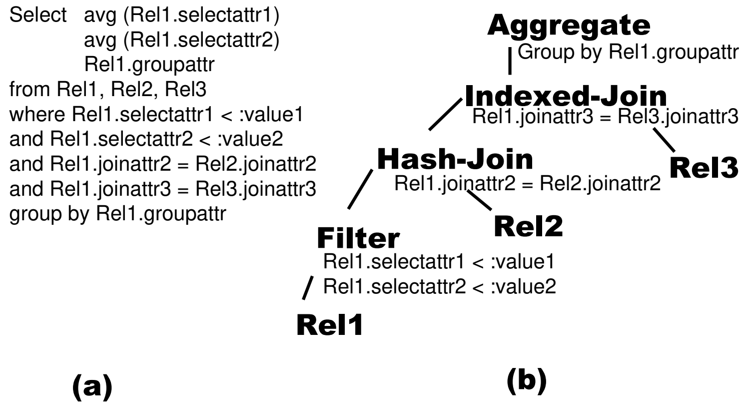


Figure 61: A query and its query execution plan

5.2.1 Query Execution Plans

The job of a query optimizer in a database system is to take as input a query (which is declarative) and produce an execution plan for that query. Figure 61(a) shows an example SQL query. We will use this query as a running example throughout this section for illustrative purposes. Figure 61(b) shows a possible execution plan for this query that might be produced by a query optimizer. An execution plan is essentially a tree in which each node represents some database operator (like *hash-join*, *index-scan*) being applied to its inputs.

During the course of optimization, the query optimizer estimates the sizes of various intermediate results that might be produced, and the cost/time taken by each operator. As part of the *Dynamic Re-Optimization* algorithm, we modify the query optimizer so that these estimates are included in the query evaluation plan that it produces, and are sent to the database execution engine. In the remainder of this chapter, we refer to such a plan as an *annotated query execution plan*. The kind of estimates we expect the plan to be annotated with are selectivities of selection and join predicates, sizes of

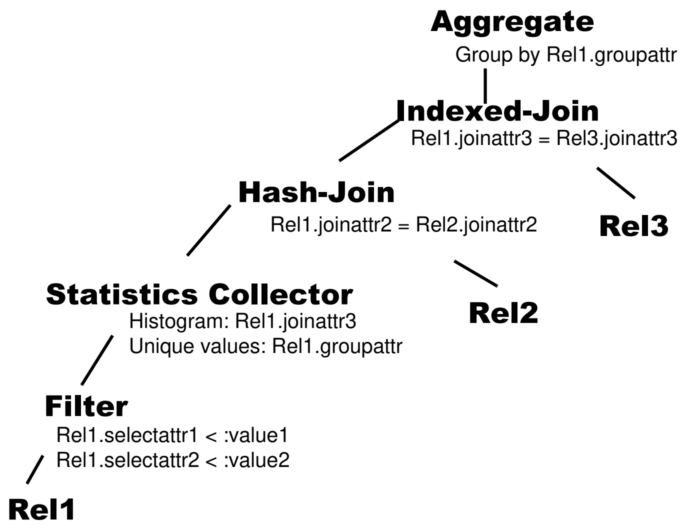


Figure 62: Collection of Statistics at run-time

the intermediate results, and estimates of the number of groups in case of aggregate operators.

5.2.2 Run-time Collection of Statistics

In this sub-section, we describe how statistics can be collected at specific points during the execution of a query plan. We describe the kinds of statistics that we can collect, and how this can be done without any I/O overhead. These statistics can then be used to get improved estimates for intermediate result sizes and operator execution costs. In this section, we deal only with the method of collecting the statistics. The question of determining what statistics to collect and at what points in the query execution plan is deferred to a later section.

We now describe how statistics can be collected for an intermediate result of a query without any I/O overhead. Consider Figure 62. There is a *filter* operation that applies selection predicates to the Rel1 relation. Just after the *filter* operation, a *statistics collector* operator is inserted into the query execution plan. As the tuples are being produced by the *filter* operator, they can be examined by a statistics collection routine, and the required statistics can be gathered without interrupting the normal execution of the query. Thus, for example, the cardinality of the result of the *filter* operation can be computed by keeping a running count of the number of tuples that stream past the statistics collection routine, and the average tuple size can be computed by keeping a running average.

There are two limitations of this approach. First, this approach cannot be used to collect any statistics that cannot be gathered in just one pass of the input. This is not a severe limitation because, the statistics that we need to gather for *Dynamic Re-Optimization* can be computed with reasonable accuracy using this approach. To compute cardinality and average tuple-size of a relation, a single pass is obviously enough. Using reservoir sampling [Vit85], histograms can also be computed with reasonable accuracy [PI95]. The number of unique values of a particular attribute (or a set of attributes) can be computed using the bitmap approach of [FM85] or reservoir sampling ([Vit85] as described in [PIHS96]).

The second limitation of this approach has to do with the pipelining of operators in a query execution. If statistics collection is being done in the middle of a pipelined execution of a row of operators, then none of the operators in the pipeline can benefit from those statistics. This is because all the operators in the pipeline are executing concurrently with the statistics collection routine. Hence, the statistics will not be ready until all the operators in the pipeline have completed a significant portion of their execution[1]. This problem is inherent in our approach, but we will see that, in spite of this limitation, *Dynamic Re-Optimization* performs well in practice.

[1] It should be noted that a blocking operator, like *hash-join*, acts as a natural break in a pipeline, because it consumes all of its first input before producing any tuples of output.

An alternative to this would be to actually break the pipeline, and force materialization of intermediate results at points where statistics need to be collected. This, however, can significantly slow down the execution of a query, and we consider this to be too high a price to pay.

It should be noted that there is a significant difference between conventional statistics that are computed and stored in system catalogs, and the statistics gathered for the *Dynamic Re-Optimization* algorithm. Conventional statistics need to be rather general in the sense that they are computed once and then used for estimations in various different types of queries. Consider a histogram built on an attribute a of relation R. This same histogram might be used to estimate the selectivity of an equality predicate of the form 'R.a = 10', a range predicate of the form 'R.a between 10 and 100', a join operation such as 'R.a = S.b' and to estimate the number of unique values of R.a (for aggregation). By contrast, histograms constructed for the *Dynamic Re-Optimization* algorithm can be very specific because the exact purpose for which the statistics are being computed is known. This can be exploited to increase the accuracy of the estimates. [PI95] indicates that different histograms are suited for different purposes. Hence, the type of histogram and method of computation can be adapted to the problem at hand.

After statistics are gathered in this fashion during query execution, they can be used to obtain new estimates for intermediate result sizes and operator execution costs for the remainder of the query. We note that the statistics collected at run-time are actually observed statistics, as opposed to estimates (which the optimizer uses). Further, as described in the previous paragraph, these statistics can be "specific" to the query being executed. Due to this, the new estimates can be a significant improvement over the *optimizer's estimates* that are included in the annotated query execution plan.

We refer to these estimates as the *improved estimates* in the remainder of this chapter. In the next two sub-sections, we describe exactly how these *improved estimates* can be used to improve the execution of the query.

5.2.3 Dynamic Resource Re-allocation

In this sub-section, we describe how *improved estimates* can be used to improve the allocation of shared resources to a query, leading to an improvement in performance. We first briefly comment upon resource allocation algorithms, and then discuss how they can benefit from *improved estimates*.

Most of the state-of-the-art algorithms for basic relational operators like sort, join and aggregate require a large amount of main memory to perform well with large datasets. The performance of these algorithms depends critically upon the actual amount of memory allocated. Assuming a workload of complex queries consisting of a number of memory-consuming operations, it is unrealistic to expect that the memory requirements of all the queries can be satisfied. This gives rise to the problem of deciding how to divide available memory among different queries in the system, and different operators in the query.

Memory allocation strategies for complex queries can be classified into two categories. The memory allocation is either decided at query optimization time by the optimizer [SAC$^+$79, C$^+$92], or it is determined at execution time based upon estimated memory characteristics of the query [MD93, YC93] (or individual operators of the query [Nag99]). In either case, these algorithms estimate the memory requirements using statistics, and decide upon an allocation of memory based on the trade-offs involved. Allocating too little memory to a particular query or operation implies that it has to do more I/0 to make up for the lack of memory, and its performance suffers. On the other hand, allocating too much memory results in under-utilization of memory (which could have been better used by another operator), again leading to sub-optimal performance. We do discuss the actual algorithms for memory management and allocation, but we note that any memory management algorithm that intelligently allocates memory based on estimated memory requirements runs into the same problems that face a conventional query optimizer: *i.e.* inaccurate estimates.

As discussed in the previous sub-section, during the course of query execution, statistics about intermediate query results can be gathered and used to improve upon the estimates of the query optimizer. These improved estimates can be used to improve the allocation of memory to the various operators of the query. Specifically, when improved estimates are available, the memory management module can be re-invoked and supplied with the new estimates. The memory management module uses these new estimates to produce a new memory allocation for the remainder of the query. Overall performance is expected to improve since the new memory allocation is based on improved estimates.

Consider for example the query execution plan in Figure 63. We now describe how this actually works in a specific database system (such as Paradise [P$^+$97]). In this plan the *filter* operator produces 15000 tuples that require 3MB of memory. Based on this estimate, the maximum memory requirement for each join is estimated at 4.2 MB (size of left input plus overhead), and the minimum requirement is 250KB. Let us assume that at run-time only 8MB of memory is available for this query. In this case the Memory Manager believes that the maximum memory requirement for both joins cannot be satisfied. Hence, it allocates 4.2 MB to the first *hash-join* (its maximum memory requirement), allocates only 250KB to the second *hash-join* (its minimum memory requirement), and allocates the left over memory to the *aggregate* operator.

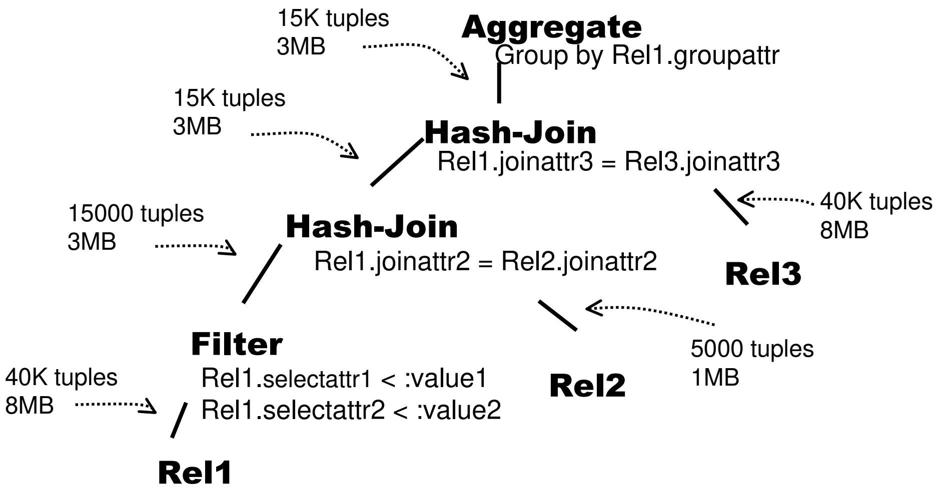


Figure 63: Use of improved statistics to improve memory allocation

This causes the second *hash-join* to execute in two passes.

If a statistics collector operator is now inserted into the query execution plan just after the *filter* operator, (as shown in Figure 62), the exact number of tuples resulting from the *filter* operation can be observed. Let us assume that the actual number of tuples satisfying the selection predicate is 7500, and not 15000. Now, the maximum memory requirement for the second *hash-join* is re-computed and is found to be 2.05MB. The Memory Manager can satisfy this requirement. Using the new memory allocation, the hash- join of *Rel3* can be completed in one pass, resulting in a significant improvement of performance.

In this chapter, we assume that once an operator starts executing, its memory allocation cannot be changed. In other words, improved statistics can only be used to improve the memory allocation for operators that have not begun executing. If, however, the operators in the database system have been implemented in such a manner that they can respond to changes in memory allocation in mid-execution, our algorithm

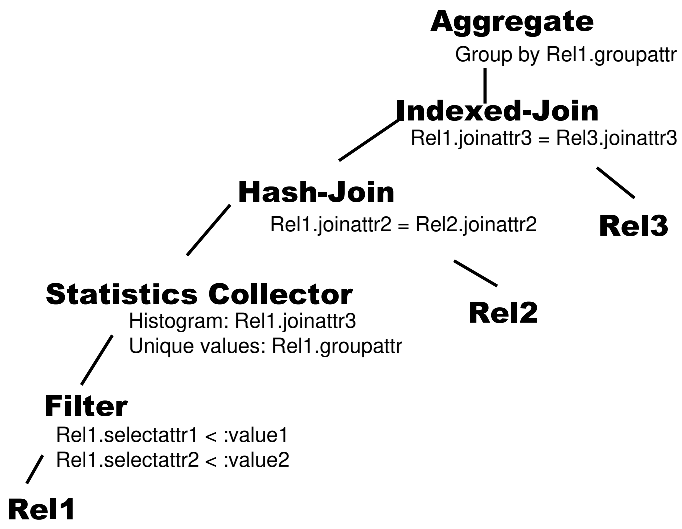


Figure 64: A potentially sub-optimal query plan

can be extended to take advantage of this.

Throughout this chapter, we have concentrated only on dynamically improving the memory allocation for a query. However, similar techniques can be applied to handle the allocation of any shared resources (e.g. processors in an SMP).

5.2.4 Query Plan Modification

In the previous sub-section, we described a relatively simple change to improve the execution of a query. The allocation of memory to the various operators in the query was modified without actually modifying the query execution plan. While that can result in significant savings in some cases, a much more serious problem with query execution is that the query execution plan itself might be sub-optimal. For example, the join order might be sub-optimal, or the choice of algorithms (*e.g. hash-join vs. indexed nested-loops join*) could be improved. In this case, tremendous savings can be achieved by modifying the query execution plan.

Consider the query execution plan shown in Figure 64. Let us assume that the query optimizer's estimate for the number of tuples resulting from the *filter* operation has a significant error[2] . Since the remainder of the query plan is based on this estimate, it is quite possible that the plan might be sub-optimal. At this point it is possible to use the new statistics to re-invoke the query optimizer and generate a better execution plan for the query.

We note that at the time the new statistics for the result of the *filter* become available, the *filter* operation has already completed execution, and the build phase of the *hash-join* algorithm is also complete. However, the probe phase of the *hash-join* has not yet started, and none of the other operators have even started execution. Under the circumstances, there are three options that the re-optimization algorithm might consider.

The simplest course of action is to completely discard the current execution, generate a completely fresh new execution plan for the query, and execute it from the beginning. This approach has the major disadvantage that it completely discards a significant amount of work that has been already performed and starts out afresh. For this approach to succeed, the combined amount of work done by the new query execution plan and the work that was discarded should be less than the work that would have been done by the previous plan. It is conceivable that this could be the case for some query plans, especially if the sub-optimality is detected early. However, we believe that this approach is too risky, and we do not consider it further in the remainder of this chapter.

The second option is to suspend execution of the query, and only re-optimize those parts of the query that have not started executing. In the example above, the *filter*

[2]There are a number of reasons why this can happen even if there are state-of-the-art histograms on the base relation. The histograms might be out-of-date. The *filter* might involve two different correlated attributes of the relation (*e.g.* 'R1.a = 10 and R1.b = 20') and the histograms do not capture the correlation. Or, the selection predicate might have a user-defined function in a external language, in which case there is no way for the database system to estimate the selectivity of the *filter*.

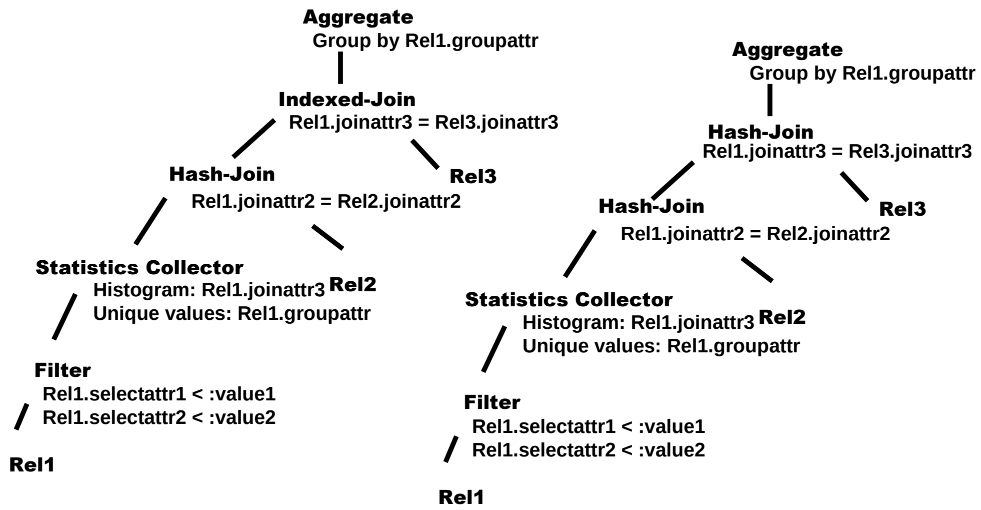


Figure 65: Re-optimization of a plan without discarding any work

operation is already complete and *hash-join* is also partially done. However, the *indexed nested-loops join* and *aggregate* have not yet begun execution. Hence, a plan involving these two operators can be modified without having to discard any work. Specifically, the query optimizer is re-invoked with new statistics. It is also given the information that the *filter* and the build phase of the *hash-join* is done. The query optimizer then produces a new plan in which the *filter* and the *hash-join* are left as they are, but the remainder of the plan is re-optimized. This situation is pictured in Figure 65.

While we believe that this approach is the best under the circumstances, it does require significant modifications to the query scheduler of the database system to make it work. Specifically it requires the ability to suspend a query in mid-execution, and to modify the query execution plan of the remainder of the query (without the knowledge of the operators that are already halfway through their execution) and to resume execution using the new plan. While this concept is easy to grasp, actually implementing it in a real system can be a problem, especially if the scheduler was not initially designed to handle situations like this.

To tackle this problem, we modified the algorithm slightly to get a new algorithm

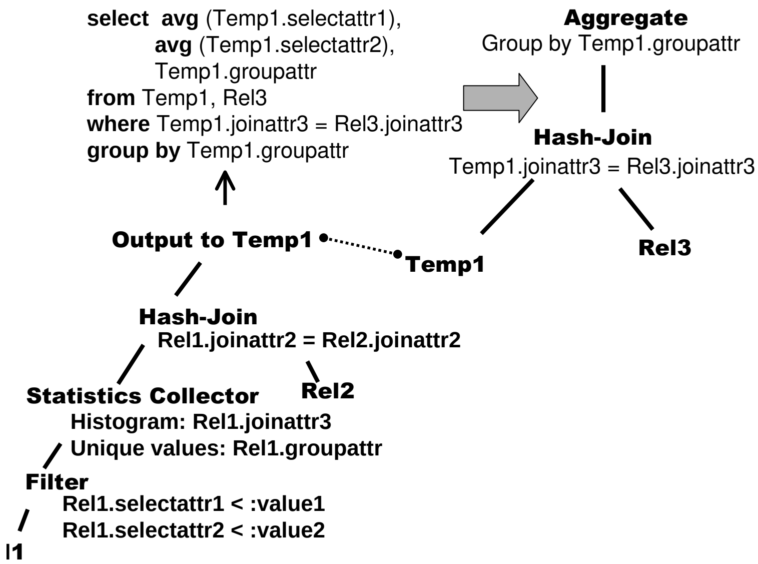


Figure 66: Re-optimization of a plan by materializing intermediate results

that is less efficient, but is much easier to implement. Figure 66 shows how this works. In this approach, we do not suspend the execution of the query, but let the currently executing operators (*i.e.* the *hash-join* involving *Rel2*) run to completion. However, instead of piping output to the next operator in the query execution plan, it is re-directed to a temporary file on disk. Now, SQL corresponding to the remainder of the query is generated in terms of this temporary file. This modified query is then re-submitted to the parser/optimizer like a regular query[3].

When to re-optimize: Re-optimizing a query has a significant overhead associated with it. First, there is a non-trivial cost associated with re-parsing and re-optimizing a query. Second, if the re-optimization forces an extra materialization of an intermediate result, the cost of writing and reading that result is incurred. For this reason, re-optimization of a query is not triggered every time the statistics of an intermediate result are observed to be different from the optimizer's estimates. Instead,

[3]Of course, care has to be taken to ensure that the new query executes in the same transaction context as the previous one.

this decision is made using some heuristics based on the (estimated) costs involved.

Let $T_{cur-plan,optimizer}$ be the optimizer estimate for the time required to execute the current plan. Let $T_{cur-plan,improved}$ be the *improved estimate* for the same. Let $T_{materialize,estimated}$ be the estimated overhead for materializing (writing and reading) the intermediate relation. Let us assume that the optimizer is actually re-invoked and it produces a new plan for the remainder of the query. In this case, let $T_{opt,actual}$ be the time that would be required to re-parse and re-optimize (the remainder of) the query. Let $T_{new-plan,total}$ be the total estimated time for executing the new plan (including the time for work already completed, the time for optimization, the time for materialization, and the time to execute the remainder of the query using the new plan).

Obviously, re-optimization should be considered only if $T_{cur-plan,improved} > T_{new-plan,total}$. Unfortunately, is not known until the optimizer is actually re-invoked. Let us, for the moment, assume that $T_{opt,actual}$ is always negligibly small. In that case, the solution is easy. When observed statistics are found to be different from the estimated statistics, the optimizer is invoked to produce a new plan (since this step is considered negligibly cheap) and an estimated $T_{new-plan,total}$. Now, if $T_{new-plan,total} < T_{cur-plan,improved}$, the new plan is accepted and we take the steps required for dynamic modification of the plan (*i.e.* materializing the intermediate result, and then executing the new plan using the materialized result). If, however, this is not the case, then we reject the new plan, and continue execution as before. In this case, no materialization of the intermediate result needs to be done, and the only overhead incurred is the $T_{opt,actual}$ required to getting an estimate for $T_{new-plan,total}$.

Unfortunately, $T_{opt,actual}$ is not always negligibly small and the overhead can be significant. We note that it is not too difficult to get a conservative estimate for

$T_{opt,actual}$. Let us call this estimate $T_{opt,estimated}$. The time taken to optimize a query does not depend upon the sizes of the datasets involved. Rather, it depends upon the number of operators in the query. Mainly, the cost is dominated by the cost of enumerating the various join orders for the query. Assuming the worst case, a query containing n joins requires the most time for optimization if it is a star-join query [OL90]. The time taken to optimize a star-join query containing n joins is usually rather stable for a given optimizer and database system. Hence, an optimizer for a particular database system can be calibrated to obtain these estimates.

Now, we use a couple of heuristics to determine whether it is worth spending $T_{opt,estimated}$ time to re-invoke the optimizer. First, we note that re-optimizing is probably not worth the trouble unless the query execution time is much higher than the optimization time. Specifically, we use the heuristic,

$$\frac{T_{opt,estimated}}{T_{cur-plan,improved}} > \theta_1 \tag{2}$$

In this equation θ_1 is a parameter for the Dynamic Re-Optimization algorithm, and should be a small quantity like 0.05. The optimizer is *not* re-invoked if equation 2 holds.

Another point to be noted is that re-optimization is probably not worthwhile unless there is reason to believe that the current plan might be sub-optimal. To model this, we use the difference between $T_{cur-plan,optimizer}$ and $T_{cur-plan,improved}$ as an indicator of whether the current plan is likely to be sub-optimal. Specifically we re-optimize only if

$$\left| \frac{T_{cur-plan,improved} - T_{cur-plan,optimizer}}{T_{cur-plan,optimizer}} \right| > \theta_2 \tag{3}$$

In this equation θ_2 is another parameter for the algorithm, and is set at approximately 0.2.

5.2.5 Keeping overheads low

So far in this section, we have described the *Dynamic Re-Optimization* algorithm, based on the assumption that "statistics" are collected at "key" points during the execution of a query. In this sub-section, we describe exactly what "statistics" are collected, and what are the "key" points in the query.

Obviously, the decision about what statistics to collect needs to be made at query optimization time. After a conventional optimizer has produced a query execution plan, we process this plan and insert statistics-collection operators at various points in the query execution plan. We will refer to this algorithm as the *statistics-collectors insertion algorithm*. This algorithm determines what are the "most effective" statistics to collect, and produces a plan containing the appropriate statistics collection operators. A simple solution would be to measure cardinalities, sizes, and histograms at all intermediate points during the execution of the query. As described in Section 5.2.2, collection of statistics at query execution time is relatively cheap since there is no I/O overhead. Nevertheless, the CPU overhead itself can be significant in some cases. For the queries that benefit from *Dynamic Re-Optimization*, the savings achieved by re-optimization out-weigh the overheads associated with statistics collection, but for queries that do not get re-optimized, the overhead actually results in an increase in the query execution time.

The *Dynamic Re-Optimization* algorithm is useful for detecting certain kinds of sub-optimalities in complex queries. However, there are a number of queries for which

Dynamic Re-Optimization does not help. Obviously, if the plan produced for a particular query is already optimal, or close to optimal, re-optimization does not help. Another possibility is that the query might be too simple (for example, consisting of just one join). In this case, even if the query plan produced by the optimizer is sub-optimal, *Dynamic Re-Optimization* is not useful, because by the time collection of statistics is complete, most of the query is also done executing. Thus, even though the new statistics may indicate that the query plan was sub-optimal, it is too late to do anything about it.

The *Dynamic Re-Optimization* algorithm is not targeted towards these queries. However, it is important that their performance does not suffer if the *Dynamic Re-Optimization* algorithm is used. If possible, statistics collection should be entirely avoided for such queries. If not, steps should be taken to ensure that the overhead introduced is kept acceptably low.

Due to these considerations, it becomes important to carefully choose what statistics are collected at query execution time. There is an important trade-off to be considered here. Collecting statistics at too many points in the query can lead to an unacceptably high overhead. On the other hand, if statistics are collected at too few points in the query, some of the sub-optimalities in the query execution plan might not get detected, leading to the loss of some optimization opportunities.

We now describe the *statistics-collectors insertion algorithm* that is used to determine what statistics to collect during query execution. For the remainder of this chapter, we assume that the time required for measurement of cardinality and size (in pages) of a table, and the minimum and maximum values for its attributes, is negligible. Hence, we assume that these statistics are measured for all intermediate results in a query. The *statistics-collectors insertion algorithm* will be restricted only to computations of histograms and estimations of number of unique values of a particular attribute (or set of attributes). If, however, in a particular database system, measuring cardinality/size has a significant overhead associated with it, the same techniques can be applied to them as well.

The *statistics-collectors insertion algorithm* starts by making a list of all the potentially useful statistics that can be computed. For a given intermediate table, a histogram on a particular attribute is potentially useful if that attribute is part of a join predicate or a selection predicate later on in the query execution plan. Similarly, computing the number of unique values of an attribute (or set of attributes) is potentially useful if that attribute (or set) is part of a group-by clause of an *aggregate* operation later in the query execution plan. Given this list of potentially useful statistics, we need to determine which ones should be discarded, and which ones computed.

The *maximum acceptable overhead*, μ (specified as a fraction of the total execution time of the query), is an external parameter supplied to the algorithm. Thus, if $T_{cur-plan,optimizer}$ is the optimizer's estimate of the query execution time, then $\mu \times T_{cur-plan,optimizer}$ is the maximum time that can be allocated to the collection of statistics. Now, we need to determine a subset of the potentially useful statistics that take less than $\mu \times T_{cur-plan,optimizer}$ time to compute, and which are "most effective" in detecting the sub-optimality of a plan. To be able to do this, we need to estimate two things. First, we need to estimate the cost of computing each of the statistics. This can be easily estimated using the optimizer's estimates of the sizes of intermediate results. Second, we need some measure of the "effectiveness" of a particular statistic in detecting sub-optimality of a plan.

Two key factors are considered while deciding the effectiveness of statistics in detecting sub-optimality of a query execution plan. The first factor is the probability that the corresponding optimizer estimates are inaccurate. If there is a high probability that the optimizer's estimates are accurate, then there is not much reason to actually observe the statistics at run-time. The second factor is the fraction of the query execution plan that is affected by that particular statistic. The larger the fraction of the query that might be affected by a statistic, the more effective is the statistic at detecting sub-optimality of a plan.

The first question that we ask is what are the chances that the optimizer's estimates corresponding to that attribute are inaccurate? For example, if there is an equality selection on a particular attribute of a base table, and there exists a serial histogram on that attribute, then chances are very high that the optimizer's estimates for the result of the selection operator are very accurate [PI95]. On the other hand, if there is neither a histogram nor an index on that attribute, chances are very high that the optimizer's estimates are rather inaccurate. In that case, computing a histogram on the result at run-time is likely to be very useful.

We chose to use a heuristic value called an *inaccuracy potential* to capture the likelyhood of an optimizer estimate being inaccurate. The *statistics-collectors insertion algorithm* assigns an *inaccuracy potential* level of *low*, *medium* or *high* to the various *optimizer estimates* in a query execution plan using the following rules. An *inaccuracy potential* of *high* for a particular statistic indicates that there is a high possibility of the corresponding optimizer estimate being inaccurate. We first assign *inaccuracy potential* levels to the statistics on base tables found in catalogs. Then the *inaccuracy potential* levels are propagated upwards in the query execution plan.

It might have been possible to formulate an algorithm that uses numeric values for *inaccuracy potential* (as opposed to the qualitative *low*, *medium* and *high*). This could be based on various error formulae for the optimizer estimation functions and histograms. However, given the number of variables and the number of approximations in this algorithm, we felt that the extra effort would not necessarily result in any significant improvements. Due to this, we chose to use a set of heuristic rules for determining the *inaccuracy potentials* based on our experiences:

- The *inaccuracy potential* for a histogram on an attribute of a base table is *low* if it has a serial histogram, *medium* for equi-width and equi-depth histograms, and *high* if there is no histogram.

- If the system catalogs contain estimates for the number of unique values of a particular attribute of a base table, the inaccuracy potential for this estimate is *low*. The inaccuracy potential for the number of unique values of an attribute (or set) at any intermediate point in a query execution plan is always *high*. (In other words, the inaccuracy potential for number of unique values is *low* only for attributes in a base table, and is *high* in all other cases.)

- Some database systems have information available about the update activity on a table since the last time statistics were updated. In this case, the inaccuracy potential level for all statistics is increased one level if there has been significant update activity since the last time statistics were collected.

- The *inaccuracy potential* for the output of a selection operator involving a simple predicate is the same as the *inaccuracy potential* of its input. In other words, *inaccuracy potential* is *low* if there exists a serial histogram on the input, *medium* for equi-width and equi-depth histograms and *high* when there are no histograms.

- If a selection predicate involves two or more attributes of the relation, then *inaccuracy potential* of the output is one level higher than the *inaccuracy potential* of the input. In other words, if the inaccuracy potential for input is *low*, then *inaccuracy potential* for output is *medium*, and if the input is *medium* or *high*, *inaccuracy potential* is *high*. This increase in *inaccuracy potential* is due to the possibility of correlations that are not captured by the histograms.

- If a selection predicate involves user-defined methods, the *inaccuracy potential* of output is always *high*.

- Consider an equi-join where the join attributes are keys for the corresponding tables. In this case, the output can be estimated rather accurately if the input is known. Due to this, the *inaccuracy potential* for the output of an equi-join on key attributes is the same as the maximum *inaccuracy potential* of its inputs. If the equi-join is on a non-key attribute, then the *inaccuracy potential* is one level higher than the *inaccuracy potential* of its inputs.

- The *inaccuracy potential* for non-equi-joins is always *high*.

- The *inaccuracy potential* for the output of an aggregate operator is the same as the *inaccuracy potential* with which the number of unique values for the grouping columns is known in the input.

The other factor in determining effectiveness of computing a particular statistic is the fraction of a query execution plan that is affected by that statistic and has not yet executed. Consider Figure 67. This figure shows two statistics being collected at query execution time. One is a histogram on the attribute *Rel1.joinattr3* in the output of the *filter* operation, and the other is the number of unique values of the *Rel1.groupattr* attribute. Now the *joinattr3* attribute is part of the join predicate in the *indexed nested-loops join* pictured in the figure, and hence the corresponding histogram is useful in estimating the cost of that join and the size of its output. Hence, the portion of the query execution plan affected by the histogram on the *joinattr3* attribute consists of all

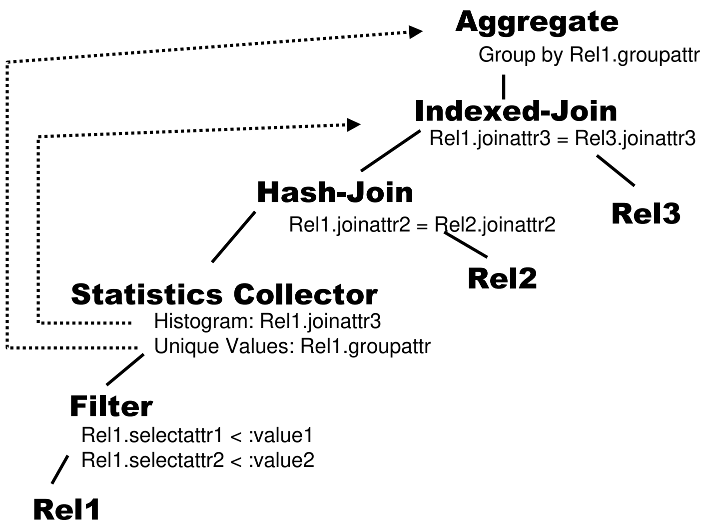


Figure 67: Fraction of a query affected by statistics

the operators after that join. On the other hand, the number of unique values for the *groupattr* attribute is only useful for the *aggregate* operation. Hence the portion of the query execution plan affected by this statistic consists only of the *aggregate* operation.

Now the relative effectiveness of two different statistics is compared as follows. If one statistic has a higher *inaccuracy potential*, then that statistic is considered to be more effective in detecting sub-optimality of a plan. If the *inaccuracy potentials* for two statistics are the same, then the statistic that affects a larger portion of the query execution plan is considered more effective. Using these rules, the list of all potentially useful statistics is ordered according to increasing effectiveness. Now, we begin deleting the least effective statistics from this list one by one until the total estimated time for computing all the statistics drops below the maximum acceptable overhead ($T_{cur-plan,optimizer}$).

5.2.6 Summary

To summarize, this is how the entire *Dynamic Re-Optimization* algorithm works. First a conventional optimizer is used to generate a conventional query execution plan for a query. Then the *statistics-collectors insertion algorithm* is invoked to insert statistics-collection operators into the query execution plan. The *statistics-collectors insertion algorithm* ensures that the statistics-collection operators inserted into the query plan do not slow down the query by more than a fraction μ. The output of the *statistics-collectors insertion algorithm* is the final static plan for the query that can be stored in the database system. We note that this plan contains all the optimizer's estimates for the sizes of various intermediate results and the execution times for the operators in the query.

At query execution time, the statistics-collector operators that have been inserted into the query gather statistics on the intermediate results of the query execution. These statistics are then used to obtain improved estimates for the execution times for the remaining operators of the query. These estimates are compared with the optimizer estimates that are stored as a part of the query plan. If the estimates are significantly different, and the query is expensive enough to warrant re-optimization, then the query optimizer is re-invoked to obtain a new plan for the remainder of the query. If the estimated total execution time for the new plan (including overhead of re-optimization and materialization of intermediate results) is less than the estimated execution time of the old plan, then the execution plan for the remainder of the query is replaced with the new plan.

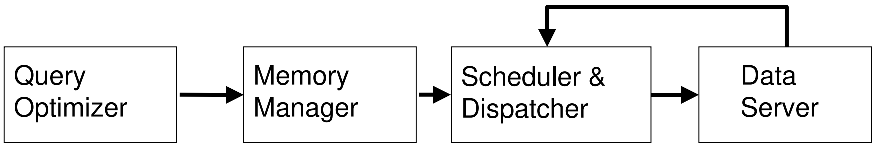


Figure 68: Query Execution in Paradise

5.3 Implementation and Performance

As an experimental validation of the *Dynamic Re-Optimization* algorithm we implemented it in the Paradise database system [P+97]. In this section, we report some of the results of our experiments. First we describe the details of the actual implementation of the *Dynamic Re-Optimization* algorithm in the context of Paradise, and its interactions with the memory management module of Paradise. Then we study the performance of the *Dynamic Re-Optimization* algorithm using datasets and queries based upon the TPCD benchmark specification [Raa95]. We also performed some experiments using skewed datasets to measure the effect of skew on performance.

5.3.1 Implementation in Paradise

Paradise is a database system designed to handle rich data-types through the use of Abstract Data Types (ADTs) and provides scalability through the use of parallelism. In our experiments, we concentrated mainly on the relational features of Paradise.

Figure 68 shows some of the components of the Paradise system that are involved in optimizing and executing a query. The query optimizer is built using the OPT++ architecture [KD98], and uses a conventional dynamic programming algorithm based on the System-R optimizer [SAC+79]. The cost estimates in the optimizer are based on histograms stored in the system catalogs. The system uses MaxDiff histograms as described in [PI95]. This produces a static plan that contains the query execution strategy as well as the optimizer's estimates of the sizes of intermediate query results.

This annotated plan is submitted to the database engine for query execution.

At query execution time, the Memory Manager of the database engine determines the allocation of memory to the various operators of the query. It determines the memory requirements (minimum and maximum memory demands) of each operator using the estimates provided by the optimizer. Based on the memory requirements of each operator, and by considering the trade-offs involved, it allocates some amount of memory to each operator. The amount of memory thus allocated to an operator represents the maximum memory that the operator is allowed to use during execution. If all the data required by the operator does not fit into the allocated amount of memory, it has to spill some of the data to disk. Details of the Memory Management module of Paradise are described in [Nag99].

The Memory Manager annotates a query execution plan with memory allocation values, and hands over the plan to the query scheduler and dispatcher for execution. The query scheduler and dispatcher executes a complex query execution plan in phases by partitioning it into a number of *segments*. Each segment is a subset of the operators in the query execution plan that can be executed concurrently. Typically, a segment consists of a set of consecutive operators that can be executed in a pipelined fashion. The different segments of a query execution plan are executed one after another in sequence. The dispatcher dispatches a segment of operators to the data-servers and waits for them to complete execution. When all the operators of a segment complete execution, a message is sent to the dispatcher, and it advances to the next segment in the execution plan.

Figure 69 shows how we modified Paradise to incorporate *Dynamic Re-Optimization*. First, the *statistics-collectors insertion algorithm* (SCIA) was added as a post-processing phase after the query optimizer. This takes the query execution plan produced by the

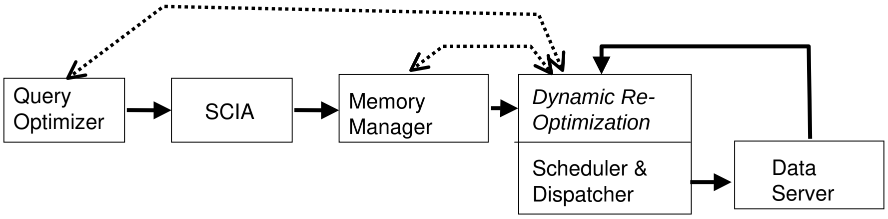


Figure 69: Query Execution with *Dynamic Re-Optimization*

optimizer and inserts statistics collection operators in it as described in Section 5.2.5. The scheduler-dispatcher is modified to take into account the *Dynamic Re-Optimization* algorithm. As in the previous design, after the Memory Manager is done with memory allocation, it hands over the plan to the scheduler and dispatcher. This partitions the plan into segments and begins dispatching each segment in sequence.

In the new scheme, when a segment is dispatched to the data-servers to be executed, it might contain *statistics-collector* operators. As far as the data-servers are concerned, these are regular operators similar to *hash-join* or *index-scan*. The only difference is that when a *statistics-collector* completes execution, it sends back to the dispatcher a message containing the statistics collected. At this point, the *Dynamic Re-Optimization* algorithm in the dispatcher is invoked. This can do one of three things at this point. First, it uses Equation 2 and Equation 3 (discussed in Section 5.2.4) to determine whether to consider re-optimizing the query. If the answer is yes, it invokes the query optimizer and obtains a new plan for the remainder of the query, using the new statistics. Then it uses the optimizer estimate of the cost of execution of the new plan to determine whether the cost of the new plan is actually less than the estimate for the old plan in spite of the re-optimization overhead. If this is true, the *Dynamic Re-Optimization* algorithm instructs the data-server to finish execution of the last operator and write the result to a temporary file. It deletes all the state information for the old plan from the dispatcher data-structures and then submits the new query plan for execution. If the new plan is not cheaper than the old plan, then the dynamic re-optimizer continues working with the old plan. However, it uses the new estimates to invoke the Memory Manager again to obtain an improved memory allocation for the plan based on the improved statistics. This process continues until the query completes execution.

In addition to implementing the *Dynamic Re-Optimization* and the *statistics-collectors insertion algorithms* in the system, we had to add the *statistics-collector* operator to the data-server. The *statistics-collector* operator was added as a regular streamed operator (similar to the *filter* operator). It took a stream of tuples as its input and produced exactly the same stream of tuples as its output. Since this operator just needs to examine the tuples without modifying or discarding any of them, it can be implemented without requiring an extra copy. To compute the size of the relation, the number of tuples, and the minimum and maximum value for an attribute, we maintain a single value that is updated after each tuple is examined. For computing a histogram, one database page is allocated to hold a reservoir sample [Vit85] for the histogram. As each tuple is examined, the value of the corresponding attribute is copied into the reservoir according to the sampling technique described in [PI95]. When all the tuples from the input are exhausted, the reservoir is examined to build the histogram.

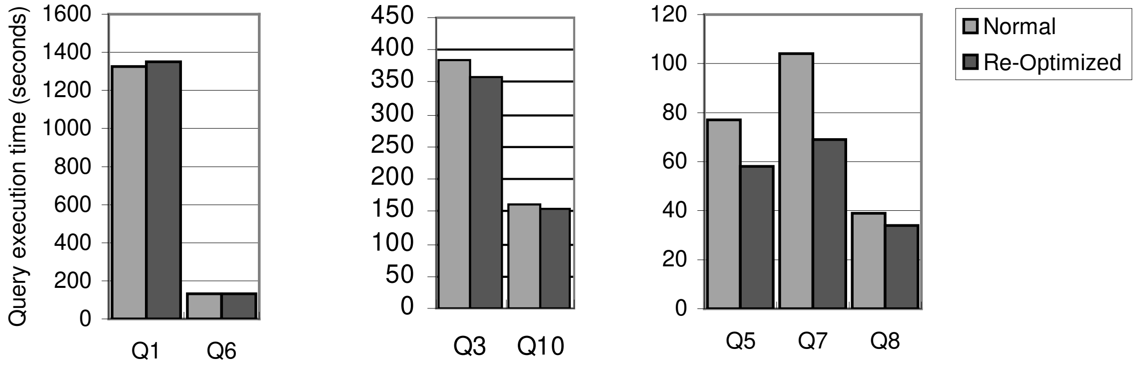
5.3.2 Experimental Results using TPC-D queries

To study the effect of *Dynamic Re-Optimization* on real queries, we performed experiments using some TPC-D queries. The TPC-D dataset generator was used with a scale factor of 3 to generate a 3GB database. Using this database, we ran queries Q1, Q3, Q5, Q6, Q7, Q8, Q10 described in the TPC-D specification [Raa95][4]. All the experiments were run on a cluster of 4 PCs each configured with dual 133 Mhz Pentium processors, 128 MB of memory, dual fast and wide SCSI-2 adapters (Adaptec 7870P), and one Seagate Barracuda 2.1 GB disk drive (ST32500WC). Solaris 2.5 was used as the operating system. The processors were connected using 100Mbit/second ethernet and a Cisco Catalyst 5000 switch that has an internal bandwidth of 1.2 GB/second. The buffer pool was kept at 32MB at each node of the system. We purposely chose not to have a larger buffer pool since we wanted to study the effect of memory management techniques on query optimization. Refer to [Raa95] for the specifications of the queries. We ran each query with and without the use of *Dynamic Re-Optimization*. Each query was executed 5 times and the average execution time was reported.

In all these queries, we set the value of μ (maximum allowable overhead) to 0.05 ensuring that none of the queries ever performed 5% worse than normal. The parameters θ_1 and θ_2 were kept at 0.05 and 0.2 respectively. An analysis of the sensitivity of the *Dynamic Re-Optimization* algorithm to the values of μ, θ_1, and θ_2 is contained in [Kab98].

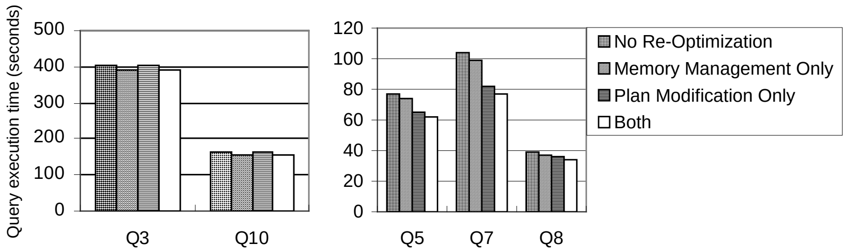
Based on the expected effects of *Dynamic Re-Optimization* on different types of queries, we can classify all queries into three categories. Queries that contain zero or one joins will never get re-optimized. We refer to such queries as *simple* queries. Queries containing two or three joins will usually not benefit much from plan modification, but might see some benefits from improved memory management. We refer to this category of queries as *medium* queries. Finally, all queries containing four or more joins are the primary targets for which *Dynamic Re-Optimization* is designed. We will refer to them

[4]The other queries in the TPC-D benchmark specification were not included in our experiments because some of the necessary features were not supported by Paradise. For the same reason, minor modifications were made to the queries that where included. In all cases where a query contained aggregates over expressions (*e.g.* SUM (L_EXTENDEDPRICE*(1-L_DISCOUNT))) we replaced it with a simpler aggregate expression (*e.g.* SUM (L_EXTENDEDPRICE)).



Simple **Medium** **Complex**

Figure 70: Performance of *Dynamic Re-Optimization*



Medium **Complex**

Figure 71: Isolating the effect of improvements due to memory management and plan modification

as *complex* queries. In the query set that we used, Q1 and Q6 are simple, Q3 and Q10 are *medium*, while Q5, Q7 and Q8 are complex.

Figure 70 shows the results of our experiments. We see that queries Q1 and Q6 do not benefit at all from *Dynamic Re-Optimization*. This is an expected result, since these are *simple* queries. We see a small increase in the execution time for Q1, indicating the overhead of statistics collection. Q3 and Q10 show modest improvements (upto 5%) in performance, while the *complex* queries show larger improvements (10 to 30%).

From the previous experiment, it is unclear how much of the performance improvement is due to improvements in the memory allocation for the query and how much is due to plan modification. To isolate these effects we performed another experiment

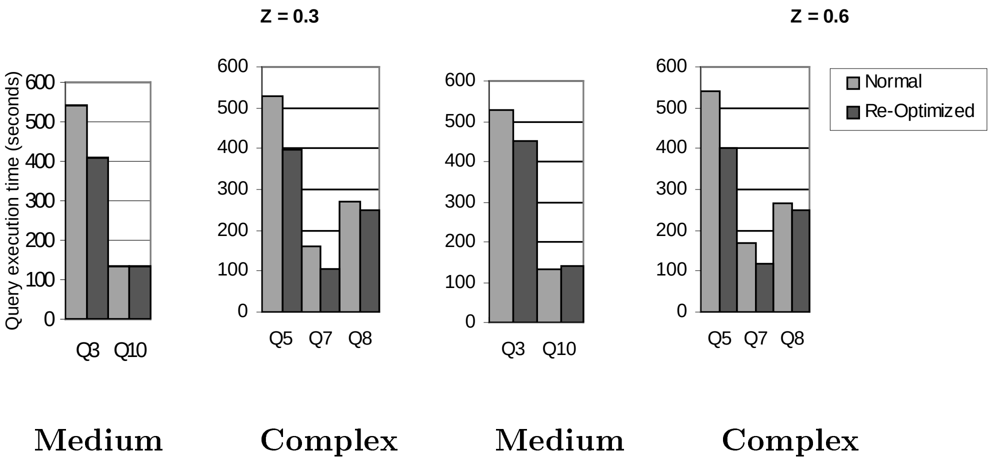


Figure 72: Effect of skew

in which the *Dynamic Re-Optimization* was run in two different modes. In one mode, the improvements in statistics were used solely for improving the memory management of the query, and plan modification was turned off. In the second mode, dynamic re-allocation of memory was turned off and only plan modification was used to improve the performance of the query. The results of this experiment are shown in Figure 71. A couple of interesting observations can be made about these results. First, we see that all the *medium* queries benefit only from improved memory management. Second, the *complex* queries benefit from both, improved memory management, as well as plan modification. They see a small improvement (5 to 10%) due to memory management and a larger one (10 to 20%) due to plan modification. Since the *simple* queries are not really affected by *Dynamic Re-Optimization* we have not included them in this or later experiments.

We also ran some experiments to study the effect of skew on the performance of *Dynamic Re-Optimization*. For this, we used the same queries with skewed data. Instead of generating TPC-D data with uniform distributions, we modified the data generator to skew all non-key attributes using generalized Zipfian distribution ([Zip49] as described in [Poo95]). We ran two sets of experiments with values for the Zipf factor

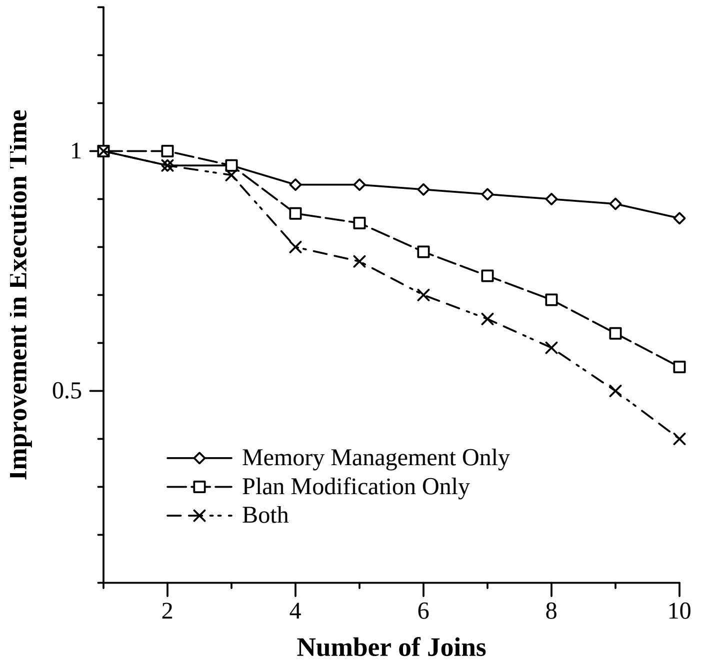


Figure 73: Dynamic Re-Optimization: Random Queries

(z) value set at 0.3 and 0.6. The results of these experiments are plotted in Figure 72. Comparing these charts with the charts for the uniform data (Figure 70) we see that the relative performance of *Dynamic Re-Optimization* improves slightly as more skew is introduced in the system. In some cases the benefit from re-optimization actually decreases when skew increases (for example Q10). This can be attributed to the fact that in some cases, the accuracy of serial histograms actually increases when skew is increased.

5.3.3 Experimental Results Using Randomly Generated Queries

In the previous section, we conducted a performance study based on a few TPC-D queries. Since the number of queries were very limited, there is a chance that the results might not be statistically skewed. To overcome this problem, we repeated similar experiments with randomly generated queries over a synthetic database.

We used a database which had ten relations of different sizes from containing about

10000 to a 100000 tuples. The sizes ranged from 10MB to 100MB. The total size of the database was about 2.5GB. We randomly generated queries containing 1 to 10 joins. Each relation in a query had a select predicate defined on it. The selectivity of the predicate had a uniform probability of being either 1%, 5%, 10%, 20% or 50%. Each join had a 90% chance of being a foreign key join, 10% chance having a non-key attribute in the join predicate. Each query had a 90% chance of having an aggregate in the project list. The aggregate function, the aggregating columns and the grouping columns were picked randomly. Each data-point in the graphs represents three randomly generated queries. Each query was run 5 times and the average time was reported. We used the same hardware platform as in the previous section.

Figure 73 shows the results. Since the execution times of two different randomly generated queries can vary wildly even though they have the same number of joins, we have only reported the ratio of the time taken for executing the re-optimized plan to the time taken for execution of the original plan. We see that the improvement due to *Dynamic Re-Optimization* increases steadily as the number of joins in the query increases. There is an improvement of about 25% for queries containing 5 joins, and it is about 60% for queries containing 10 joins. We see that most queries show about a 5 to 15% improvement due to memory management. The rest of the improvement comes from plan modification.

Chapter 6

Related Work

6.1 Extensible Query Optimization

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 coming up with 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.

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

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

The optimizer developed as a part of the Starburst project [LFL88, HP88] uses a two step process to optimize queries. The first phase uses a set of production rules to 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. Each "non-terminal" in the grammar can have multiple production rules (suggesting execution alternatives) and conditions of applicability. These rules are used to construct an optimal execution plan in a bottom up fashion similar to the System-R optimizer. Cost estimates are used for choosing between alternatives.

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

The optimizers generated by the Exodus Optimizer Generator [GD87], 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 equivalent to the input query. The implementation rules are used to generate access plans corresponding to the operator trees.

Like the Volcano Optimizer Generator and the Starburst optimizer, OPT++ incorporates extensible specification of logical algebra operators, execution algorithms, logical and physical properties, and selectivity and cost estimation functions. Interesting physical properties, input constraints for execution algorithms and enforcers ("glue" operators) are also supported. OPT++ can be used to emulate both, the Starburst as well as the Exodus/Volcano based optimizers. The search strategies that are used in those optimizer generators are both built into OPT++. In fact, OPT++ can also be used to implement the transformation rules and implementation rules of Volcano and the rewrite 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 [SJ90], the blackboard architecture of Kemper, Moerkotte and Peithner [KMP93], are all based on the concept of dividing an optimizer into regions that carry out different parts of the optimization. A query then has to pass through these various regions to be optimized. These architectures differ in the methods used to pass control between the various regions.

In [SJ90], control passes from one region to another in a fixed sequence. [MDZ93] uses a hierarchy of regions in which the parent region dynamically controls the sequence of regions through which the query passes while being optimized. In the blackboard approach [KMP93], knowledge sources are responsible for moving the queries between regions.

All these architectures describe very general frameworks for extensible query optimization that support multiple optimizer control strategies and allow the addition of new control strategies. By making very specific assumptions about the kinds of manipulations that are allowed on the operator trees and access plans, OPT++ is able to put a 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 assumptions, a number of different search strategies can easily be implemented in OPT++ quite easily.

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

The EROC toolkit for building optimizers [MBHT96] was developed concurrently with OPT++ and 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[1], 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

[1] Different classes derived from the `TREETOPLANGENERATOR` classes, as explained in Section 2.4

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 addressed in OPT++ currently.

6.2 Dynamic Re-Optimization

One of the earliest query optimizers [WY76] was, in some sense, a dynamic query optimizer. However, after the publication of [SAC+79], most of the work on query optimization has focussed on optimization of a query at compile time. Since the late 80s, however, the limitations of this approach have begun to be felt, and there has been an emergence of a number of different query optimization schemes in which some of the optimization decisions are postponed to query execution time.

[DMP93] describes a scheme in which query execution plans generated by an optimizer are re-optimized just before query execution time if they are believed to be sub-optimal. At query optimization time, the statistics used by the optimizer to generate the optimal plan are stored with the plan in the database system. At query execution time, the actual statistics from the system catalogs are compared against the statistics stored in the plan. If they are found to differ significantly the query is re-optimized before execution. This differs significantly from our approach. First, the query is only re-optimized before execution begins. In their approach, there is no collection of statistics, or modification of the plan in the middle of query execution.

The competition model of Antoshenkov [Ant93, Ant96] represents another way of dynamically determining the plan of a query. In his approach, competing executions start executing using different plans. After a while, it becomes clear that one of the plans is better than the others, and the execution of the sub- optimal plan is stopped.

While this approach might work well for determining which access method to use for a particular table-scan, or which join algorithm to use for executing a particular join, it cannot be extended easily to the case where the join order for a complex query might be sub-optimal. Further, the competition model cannot be used for dynamically improving the resource allocation of a query.

One important reason for sub-optimality of query execution plans is that a lot of information about the run-time system (availability of memory, bindings of host language variables, existence of indices) is not available at query compile time. The dynamic execution plans of [GW89, GC94], and the parametric query optimization algorithm of [INSS92] try to tackle this problem. Their approach is to produce a composite plan that is in effect a combination of a number of different plans, each of which is optimal for a given set of values of run-time parameters. One of the problems with this approach is that as the number of things that are unknown at query optimization time increases, the space/time complexity of the optimization algorithm, and the complexity of the parameterized/dynamic plan produced by the algorithm increases. Given the limited amount of time that is available for query optimization, these approaches either have to resort to the use of randomization for exploring the vast search space [INSS92], or to make simplifying assumptions [GC94]. Another shortcoming of these approaches is that they do not address the issue of statistical and propagational errors in estimates. Thus, if a histogram-based estimate of the selectivity of a predicate is inaccurate, the corresponding sub-optimality in a plan cannot be detected using these approaches. However, they do have an advantage over Dynamic Re-Optimization in that they do not impose any overheads on query execution at run-time.

A hybrid algorithm that combines the parametric/dynamic query plans approach and the *Dynamic Re-Optimization* algorithm could possibly combine the best features of both approaches. The query optimizer can try to anticipate the most common cases that might arise at run-time and produce a parameterized plan that covers these possibilities. At query execution time, statistics can be observed/collected to determine which plan to choose for query execution. If a situation arises at run-time that is not covered by the common cases anticipated by the query optimizer, dynamic re-optimization can be used. This approach suggests a possible direction of future research.

Chapter 7

Conclusions

Modern database systems expose a number of problems with traditional query optimizers. As new applications for database systems emerge, query optimization is becoming more and more difficult, and at the same time more critical to the success of the system. This thesis presents our attempts at solving some of the problems faced by query optimizers.

To tackle the problems of extensibility and maintainability of query optimizers we describe OPT++ a new architecture for building extensible optimizers. It uses an object-oriented design to provide extensibility through the use of inheritance and late binding. The design makes it easy to implement a new optimizer as well as to modify existing optimizers implemented using OPT++. Extensibility is provided in the form of the ability to easily extend the logical or physical query algebra, to easily modify the search space explored by the search strategy, and to even change the search strategy.

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.

Debugging an optimizer is a complex and time-consuming task. In particular, determining the source of a bug in an optimizer that produces sub-optimal plans is difficult. We describe support for debugging that is incorporated into OPT++, including visual optimizer execution tracing, and automated detection of potential sources of errors using hints from the Optimizer-Implementor. This support is very valuable for an Optimizer-Implementor and greatly reduces the development time required to implement or modify an optimizer.

We have described the implementation and comparative performance study of various relational and object-relational optimization techniques and search strategies. This has been possible through the use of OPT++ and demonstrates the usefulness of the OPT++ architecture as a tool for experimental study of query optimization. The performance results we describe and the resulting recommendations would be valuable inputs to an Optimizer-Implementor who wishes to determine what optimization techniques to implement in an optimizer for a specific database system.

The *Dynamic Re-Optimization* algorithm we described, can detect sub-optimalities in query execution plans for complex queries, and improve the performance of such queries by dynamically re-optimizing the execution plan. Strategically placed statistics collectors are inserted into query execution plans to observe sizes and data distributions of intermediate query result sizes at run-time. These run-time statistics are used to improve the allocation of shared resources (memory) to the query, and to modify the query execution plan if need be. We also describe how this can be done efficiently without placing too much of an overhead on the execution of the query. We have demonstrated experimental results to support our claim that Dynamic Re-Optimization can significantly improve the performance of complex queries if their query execution plans are sub-optimal without significantly slowing down the queries whose plans do not benefit from re-optimization.

As emerging new applications force databases to support complex decision support queries, complex data-types and user-defined methods, it will become more and more difficult for query optimizers to statically produce good query execution plans. Some form of re-optimization of query execution plans at run-time will become necessary in such cases. We believe that the techniques we have presented, possibly in combination with parameterized plans will form the basis for the future evolution of query optimizers to meet this challenge.

Declarative query languages and automatic query optimization were an important reason for the success of relational database systems. Lack of good query optimizers could very well lead to the downfall of the next wave of innovations in database system technology. In this thesis, we have examined the inadequacies of traditional query optimizers in dealing with issues raised by modern database systems and demonstrated ways to overcome them. We believe that the ideas contained in this thesis represent an important step in ensuring that query optimizers keep up with the other advances in database systems.

Bibliography

[Ant93] Gennady Antoshenkov. "Dynamic Query Optimization in Rdb/VMS". In *In Proceedings of the IEEE Conference on Data Engineering*, pages 538–547, 1993.

[Ant96] Gennady Antoshenkov. "Dynamic Optimization of Index Scan Restricted by Booleans". In *In Proceedings of the IEEE Conference on Data Engineering*, pages 430–440, 1996.

[BG92] Ludger Becker and Ralf Harmut Guting. "Rule-Based Optimization and Query Processing in an Extensible Geometric Database System". In *ACM Transactions on Database Systems*, volume 17:2, June 1992.

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

[C$^+$92] M. S. Chen et al. "Using Segmented Right-Deep Trees for Execution of Pipelined Hash Joins". In *Proc. of the 18th VLDB Conf.*, 1992.

[CS96] Surajit Chaudhari and Kyuseok Shim. Optimizing queries with aggregate views. In *Proceedings of the 1996 ACM-SIGMOD Conference*, Montreal, Canada, June 1996.

[DMP93] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. "Adaptive Query Optimization in a Deductive Database System". In *In Proceedings of the Proceedings of the Second International Conference on Information and Knowledge Management*, Washington D. C., USA, 1993.

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

[FM85] P. Flajolet and G. N. Martin. "Probabilistic Counting Algorithms for Database Applications". In *Journal of Computer and System Sciences*, volume 31(2), pages 182–209, 1985.

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

[GC94] Goetz Graefe and Richard Cole. “Optimization of Dynamic Query Evaluation Plans”. In *Proceedings of the 1994 ACM-SIGMOD Conference*, 1994.

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

[GLPK94] César Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. “Fast, Randomized, Join-Order Selection - Why Use Transformations”. In *Proc. of the 20th VLDB Conf.*, Santiago de Chile, Chile, 1994.

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

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

[Gra95] Goetz Graefe. “The Cascades Framework for Query Optimization”. In *Bulletin of the Technical Committee on Data Engineering*, volume 18-3, pages 19–29, September 1995.

[GW89] Goetz Graefe and Karen Ward. “Dynamic Query Evaluation Plans. In *SIGMOD Proceedings*, pages 377–388. ACM, June 1989.

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

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

[IC91] Yannis Ioannidis and S. Christodoulakis. “On the Propogation of Errors in the Size of Join Results”. In *Proceedings of the 1991 ACM-SIGMOD Conference*, Denver, Colorado, May 1991.

[IK84] Toshihide Ibaraki and Tiko Kameda. “Optimal Nesting for Computing N-relational Joins”. In *ACM Transactions on Database Systems*, volume 9 of *3*, pages 482–502, October 1984.

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

[INSS92] Yannis Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos Sellis. “Parametric Query Optimization”. In *Proc. of the 18th VLDB Conf.*, 1992.

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

[Kab98] Navin Kabra. *“Query Optimization for Relational and Object-Relational Database Systems”*. PhD thesis, University of Wisconsin, Madison, 1998.

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

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. “Optimization of Nonrecursive Queries”. In *Proc. of the 12th VLDB Conf.*, Kyoto, August 1986.

[KD98] Navin Kabra and David J. DeWitt. “Opt++: An Object Oriented Implementation for Extensible Database Query Optimization”. In *to appear in The VLDB Journal*, 1998.

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

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

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

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

[MBHT96] William J. McKenna, Louis Burger, Chi Hoang, and Melissa Truong. "EROC: A Toolkit for Building NEATO Query Optimizers". In *Proc. of the 22nd VLDB Conf.*, Mumbai (Bombay), India, 1996.

[MD93] Manish Mehta and David J. DeWitt. "Dynamic Memory Allocation for Multiple Query Workloads". In *Proc. of the 19th VLDB Conf.*, Dublin, Ireland, 1993.

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

[MS79] C. L. Monma and J. B. Sidney. "Sequencing with Series-Parallel Precedence Constraints". In *Mathematics of Operations Research*, volume 4, pages 215–224, 1979.

[Nag99] Biswadeep Nag. *"Memory Allocation Strategies for Decision Support Systems"*. PhD thesis, University of Wisconsin–Madison, August 1999.

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

[P+97] Jignesh M. Patel et al. "Building a Scalable Geo-Spatial DMBS: Technology, Implementation, and Evaluation". In *Proceedings of the 1997 ACM-SIGMOD Conference*, Tuscon, Arizona, May 1997.

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

[PI95] Viswanath Poosala and Yannis Ioannidis. "Histogram-Based Solutions to Diverse Database Estimation Problems". In *Data Engineering Bulletin*, volume 18(3), pages 10–18, 1995.

[PIHS96] Viswanath Poosala, Yannis Ioannidis, Peter J. Haas, and Eugene Shekita. "Improved Histograms for Selectivity Estimation of Range Predicates". In

Proceedings of the 1996 ACM-SIGMOD Conference, Montreal, Canada, June 1996.

[Poo95] Viswanath Poosala. "Zipf's Law". Technical report, University of Wisconsin, Madison, 1995.

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

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. "Access Path Selection in a Relational Database Management System". In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1979.

[SAH87] Michael Stonebraker, Jeff Anton, and Michale Hirohama. "Extendability in POSTGRES". In *Data Engineering Bulletin*, volume 10(2), pages 16–23, 1987.

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

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

[SI92] Arun Swami and Balakrishna R. Iyer. "A Polynomial Time Algorithm for Optimizing Join Queries". Research Report RJ 8812, IBM Almaden Research Center, 1992.

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

[Vit85] J. S. Vitter. "Random Sampling with a Reservoir". In *ACM Transactions on Mathematical Software*, volume 11, pages 37–57, 1985.

[VM96] Bennet Vance and David Maier. "Rapid Bushy Join-Order Optimization with Cartesian Products". In *Proceedings of the 1996 ACM-SIGMOD Conference*, Montreal, Canada, 1996.

[WY76] Eugene Wong and Karel Youssefi. "Decomposition – A Strategy for Query Processing". In *ACM Transactions on Database Systems*, September 1976.

[YC93] Philip S. Yu and D. W. Cornell. "Buffer Management Based on Return on Consumption in a Multi-Query Environment". In *VLDB Journal*, volume 2(1), January 1993.

[YL95] Weipeng P. Yan and Per-Ake Larson. Eager and lazy aggregation. In *Proc. of the 21st VLDB Conf.*, Zurich, Switzerland, September 1995.

[Zip49] G. K. Zipf. *"Human Behavior and the Principle of Least Resistance"*. Addison-Wesley, Reading, MA, 1949.