INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

IMPLEMENTING THE POSTGRESQL QUERY OPTIMIZER WITHIN THE OPT++ FRAMEWORK

JU WANG

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

DECEMBER 2002
© JU WANG, 2003

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-77723-5

Canada

Abstract

Implementing the PostgreSQL Query Optimizer within the OPT++ Framework

Ju Wang

As a promising object-oriented reuse technology, frameworks have been attracting enough attention. However, much less work has been done on framework-based development, compared with on framework development.

In this thesis we described our work, framework-based implementation of an optimizer for a relational database. We implemented our target optimizer based on a query optimization framework, OPT++. In order to borrow expertise from a mature optimizer, we studied a model optimizer, the PostgreSQL optimizer. The operators and algorithms supported by it are summarized. The transformative rules applied are extracted. Its search strategy is analyzed.

In our application, typical operators for relational algebra are supported. Main transformative rules extracted from the PostgreSQL optimizer are applied. A PostgreSQL-like search strategy is implemented. Constrained dynamic programming and genetic algorithm are incorporated to optimize joins. Additionally, we modified the framework to fit sub-queries and explicit joins. We describe our implementation by following the framework recipes. Problems and implementation considerations are presented in detail. Furthermore, more general issues in framework-based development are discussed.

Acknowledgements

I would like to thank my supervisor, Dr Gregory Butler, for his guidance and finance support. Also I would like to acknowledge Navin Kabra and Jinmiao Li. My thesis is based on their work. I got prompt replies from them when asking them for helps. I had helpful discussions with Bin Nie when doing the implementation. Finally, I would like to thank the Quebec government for the financial support I received during my studies at Concordia University.

Contents

List of Tables VIII

List of Figures IV

Chapter 1 Introduction ... 1

1.1 The Problem and Related Work ... 1

1.1.1 The Problem ... 1

1.1.2 Related Work ... 2

1.2 Our Work ... 4

1.3 Contribution of the Thesis ... 5

1.4 The Layout of the Thesis ... 5

Chapter 2 Background ... 7

2.1 Frameworks ... 7

2.1.1 What is a Framework? ... 7

2.1.2 Benefits of Frameworks ... 8

2.1.3 White-box Frameworks and Black-box Frameworks ... 9

2.1.4 Frameworks and Design Patterns ... 10

2.2 Query Optimization of Databases ... 11

2.2.1 Overview ... 11

2.2.2 Cost-based Plan Selection ... 13

2.2.3 Search Strategies ... 14

Chapter 3 OPT++: An Object-Oriented Framework for Query Optimization ... 16

3.1 Overview ... 16

3.1.1 Basic Idea to Achieve Extensibility and Reusability ... 17

3.1.2 Architecture ... 18

3.2 The Algebra Component ... 21

3.2.1 Classes and Hooks ... 21

3.2.2 Equivalence of Two Logical Operator Trees ... 24

3.3 The Search Space Component ... 25

3.3.1 Design Patterns Used 26
3.3.2 Classes and Hooks 28
3.4 The Search Strategy Component 33
3.4.1 Design Patterns Used 34
3.4.2 Classes and Hooks 35
3.5 Cost Evaluation and Dynamic Pruning 41
3.5.1 Cost Evaluation 41
3.5.2 Dynamic Pruning 42
Chapter 4 The PostgreSQL Optimizer 46
4.1 Overview 46
4.2 Main Data Structures of the PostgreSQL Optimizer 47
4.2.1 Internal Representation of a Query 47
4.2.2 Internal Representation of a Plan 50
4.3 Transformative Rules Used in the PostgreSQL Optimizer 51
4.4 Search Strategies Used in the PostgreSQL Optimizer 55
4.4.1 Overall Search Strategy 55
4.4.2 The Constrained Dynamic Programming Search Strategy 58
4.4.3 The Genetic Algorithm Query Optimization in PostgreSQL 63
Chapter 5 The Implementation in OPT++ 68
5.1 Overview 68
5.2 Internal Representation of a Query 68
5.3 Application of Transformative Rules 72
5.3.1 Expression-normalization 72
5.3.2 Select-push-down 73
5.3.3 Sub-query-pull-up 74
5.4 Implementation of the Algebra Component 76
5.4.1 Define Logical Algebra 77
5.4.2 Define Physical Algebra 80

5.4.3 Set up Relationships between the Logical and Physical Algebras 83
5.5 Implementation of the Search Space Component 85
5.5.1 Implement the Visitor Hierarchy 85
5.5.2 Implement the Generator Hierarchy 86
5.6 Implementation of the Search Strategy Component 88
5.6.1 Implement Property Calculation 88
5.6.2 Implement the PostgreSQL Search Strategy 91
5.7 Problems and Implementation Considerations 95
5.7.1 Generality and Efficiency 95
5.7.2 Framework Mismatch and Solution 96
5.7.3 Special Treatment for GroupBy and OrderBy 98
5.7.4 Constrained Dynamic Programming 99
5.7.5 Genetic Algorithm 99
5.8 Testing the Target Optimizer 101
5.8.1 Correctness Test 101
5.8.2 Efficiency Test 104
5.9 Reuse Summary 106
5.10 Limitations of Our Implementation 107
Chapter 6 Issues in Framework-Based Development 109
6.1 Development Process 109
6.1.1 Process for General Software Development 109
6.1.2 Process for Framework-Based Development 110
6.2 Understanding Frameworks 112
6.3 Unplanned Customization 115
6.4 Need for Tools 117
Chapter 7 Conclusion 119
Bibliography 121

List of Tables

Table 3.1 Expression Sets of Some Trees 24

Table 3.2 The Main Attributes of Class OperatorTree 38

Table 3.3 The Main Attributes of Class OperatorTreeProperty 38

Table 3.4 The Main Methods of Class OperatorTreeProperty 39

Table 3.5 The Main Attributes in Class AlgorithmTree 40

Table 3.6 The Main Attributes in Class AlgorithmTreeProperty 40

Table 4.1 Logical and Physical Operators Supported by PostgreSQL 46

Table 4.2 The Possible Combinations 59

Table 5.1 Operators and Associated Algorithms in Our Algebra 84

Table 5.2 Optimization Times of the Two Optimizers 105

List of Figures

Figure 2.1 Schematic Diagram of Query Processing 11

Figure 3.1 Basic System Design 18

Figure 3.2 System Architecture 19

Figure 3.3 Overall Class Diagram 20

Figure 3.4 The Algebra Component 21

Figure 3.5 Some Logical Trees 25

Figure 3.6 The Sequence Diagram for Visitor Pattern in the Search Space Component . 27

Figure 3.7 The Search Space Component 27

Figure 3.8 The Activity Diagram for the Method *VisitDBOperators* 29

Figure 3.9 The Activity Diagram for the Method UnaryOperatorExpand::apply 31

Figure 3.10 The Activity Diagram for UnaryAlgorithmTreeGenerator::MakePhyNodes 32

Figure 3.11 The Activity Diagram for UnaryAlgorithmTreeGenerator::Apply 32

Figure 3.12 The Search Strategy Component 34

Figure 3.13 The Activity Diagram for Deleting an Algorithm Tree 43

Figure 3.14 The Activity Diagram for Deleting an Operator Tree 44

Figure 4.1 A Query Tree 49

Figure 4.2 A Plan 51

Figure 4.3 The Activity Diagram for PostgreSQL Optimization 57

Figure 4.4 Constrained Dynamic Programming 61

Figure 4.5 The Activity Diagram for Genetic Algorithm 64

Figure 4.6 A Join Tree 65

Figure 4.7 The Activity Diagram for GEQO 65

Figure 5.1 The Class Diagram for a Query Tree. 69

Figure 5.2 Select-push-down 73

Figure 5.3 The Activity Diagram for Sub-query-pull-up 75

Figure 5.4 A Join Tree after Sub-query-pull-up 76

Figure 5.5 The Customized Logical Algebra 77

Figure 5.6 The Customized Physical Algebra 81

Figure 5.7 The Customized Generator Hierarchy 86

Figure 5.8 The Implementation of the Search Strategy 93

Figure 5.9 The Implementation of Bottom-up Strategy 95

Figure 5.10 The Implementation of the Method JoinExpand::Apply 98

Figure 5.11 The Classes for Genetic Algorithm 100

Chapter 1 Introduction

As an object-oriented reuse technology, frameworks have been studied for over a decade. A lot of work on framework development, evolution, documentation and application has been done, and the technology still remains a hot research issue. The Know-It-All Project [But02] is investigating methodologies for the development, application, and evolution of frameworks. A concrete framework for database management systems is being developed as a case study for the methodology research. To investigate issues in framework-based application development, and evaluate and verify the query optimization sub-framework, we built a query optimizer for a relational database based on the sub-framework.

1.1 The Problem and Related Work

1.1.1 The Problem

Performance of computer hardware has increased dramatically in the past decades. However, development of complex software is still expensive and error-prone. One of the reasons for the situation is that much of cost and effort is wasted on re-discovery, re-design and re-implementation across software industry. Effective software reuse helps to attack the problem. Therefore, software reuse has been one of the main goals of software engineering [Fay99].

Frameworks are an object-oriented reuse technology. They provide reuse at the level of domain knowledge, requirements, architecture, micro-architecture, design, and code, in the context of a product line. As an active research field, frameworks have been studied for over a decade in aspects of development, documentation, evolution and application. Also, in practice, many frameworks are developed and applied successfully in a variety of domains. Some of the famous examples are MVC, ET++, and IBM San Francisco framework. MVC (Model/View/Controller) [Gol84] is the Smalltalk user interface framework. ET++ [Wei88] is a portable application framework, used to make platform-independent GUI-based application programs. IBM San Francisco [IBM02] is a business framework for commercial application assembly.

Although a lot of work has been done, many issues, such as reducing framework development cost, domain-specific enterprise framework development, black-box

frameworks, framework development management, framework economics, framework standards [Fay97] need further study. Therefore, framework technology remains a hot research field.

Within the field, much work has been done on the design, documentation, evolution of object-oriented frameworks, but few studies have been conducted on framework-based software development. The main concern of the thesis is framework-based software development.

Know-It-All is a framework methodology study project. Its goals are to investigate methodologies for the development, application, and evolution of frameworks and to develop a framework for database management systems as a case study for the methodology research. Some sub-frameworks have been developed. The query optimization framework is one of the sub-frameworks.

The thesis work is a subproject of the Know-It-All project. The aims of the subproject are:

- to investigate issues in framework-based application development;
- to evaluate and verify the framework for optimizers; and
- to build an optimizer for a relational database based on the framework.

Framework-based application development is also called adaptation or instantiation. Our work is basically the process of instantiation. Here, only issues in white-box frameworks are considered because of the scope of our project.

1.1.2 Related Work

There is quite a lot of published literature on development, documentation and evolution of object-oriented frameworks, but there are few papers talking about using frameworks. Relatively systematic studies have been done by Garry Froehlich et al. [Fro97, For98, Fro99, Fro00] in the University of Alberta. They summarize the issues in using frameworks, propose some guidelines for choosing a framework for a domain application, propose a hook model to help understand and instantiate a framework, also specify requirements a tool called HookMaster that helps develop and instantiate frameworks. Their method is on the basis of specifying all hooks in a framework properly. Hooks, as it will be explained in section 2.1.1, are specific ways in which a

framework can be customized. They believe that hooks can help understand and instantiate frameworks by being characterized in two dimensions. In his thesis, Mattsson [Mat96] outlines general activities involved in framework-based development, and discusses some potential problems in the field.

From the perspective of query optimization, because writing, debugging an optimizer and evaluating different search strategies are difficult and time-consuming, a lot of attempts have been made to build extensible optimizers. Some early optimizers, for example, ones in System-R [Fre87] and Starburst [Lee88] allow the algebra to be extended but with fixed search strategies. In contrast, others allow extensible search strategies. OPT++ [Kab99], which is an object-oriented optimization framework, tried to achieve both goals.

Concretely, our work builds on two software artifacts: OPT++, and the PostgreSQL optimizer.

OPT++

The query optimization framework we instantiated is called OPT++, which was developed by Navin Kabra et al. in the University of Wisconsin. Although its developers did not call it a framework, it is a framework from the perspective of software engineering. It designs frozen spots that contain expert knowledge. The expertise knowledge can be used without any changes. Also it defines hot spots that allow the optimizer-implementer to extend it easily. OPT++ uses the object-oriented features of the C++ programming language to simplify the task of implementing, extending and modifying an optimizer. It incorporates all the features of an extensible optimization framework including specification of a logical algebra, execution algorithms, logical, and physical query processing alternatives, and selectivity and cost estimation.

Jinmiao [Li01] refined and re-documented the OPT++ framework. Some design patterns were applied explicitly, and some components were re-decomposed to make the framework more flexible and understandable.

Our work was actually based on the revised version of OPT++.

The PostgreSQL Optimizer

To build a new optimizer, it is a good idea to study a mature query optimizer. Therefore, we studied a model optimizer, the PostgreSQL optimizer. The reasons why we chose it are:

- The PostgreSQL optimizer is typical. PostgreSQL stems from Postgres [Pos02a], which is viewed as the ancestor of relational databases. Although, it experienced a lot of critical changes, and now it is an object-relational database management system, its optimizer is still a typical one covering most common optimization techniques.
- PostgreSQL is an open-source database management system. The source code is available to public. Hence, we can study the optimizer in detail.

Unfortunately, there is very little literature talking about the PostgreSQL optimizer, so we mainly focused on its source code and comments.

1.2 Our Work

To instantiate the framework, our work involved studying the model optimizer, understanding the framework, modifying the framework for unplanned customization, and customization.

- Studying the PostgreSQL optimizer
  - Its relational algebra implemented was summarized;
  - Key data structures were analyzed;
  - Main transformative rules were extracted; and
  - Search strategies adopted in the optimizer were studied.
- Understanding the framework

  Understanding the framework is a nontrivial work in framework instantiation. OPT++ is a white-box frame, so it needs the user to understand its internals to use it effectively. We studied its overall architecture, design patterns used, main data structures and algorithms, example applications to gain suitable understanding before we started our implementations. In fact, the understanding and the customization overlap for a rather long time. Therefore, understanding the framework accounts for a considerable proportion of the time of instantiation.

- Modifying the framework for unplanned customization

  Because frameworks are generalized from applications, no matter how elaborately they are designed, there can be some customization that is unplanned. In some cases, custom requirements cannot be fitted only by extending the framework. We modified such things to fit the unplanned customization.

  - The parser;
  - Internal representation of query and expression; and
  - The control flow in part (to be recursive to fit sub-query processing).

- Customization

  - Implementing abstract hook methods;
  - Overriding some hook methods with default implementation;
  - Overloading a hook method; and
  - Adding new application specific classes when necessary.

1.3 Contribution of the Thesis

In this thesis, we describe the implementation of a query optimizer based on a query optimization framework, OPT++. The PostgreSQL optimizer is analyzed deeply. The operators and algorithms supported by the optimizer are summarized. The transformative rules applied are extracted. Its search strategy is studied. OPT++ was evaluated and verified. A PostgreSQL-like search strategy is implemented within the framework. Constrained dynamic programming and genetic algorithm are incorporated to optimize joins. In addition, we modified the framework to fit sub-queries and explicit joins. Problems and implementation considerations are presented in detail. Furthermore, on the basis of our experience, more general issues in framework-based development are discussed, and some guidelines for the issues are proposed.

1.4 The Layout of the Thesis

Chapter 2 gives brief background information about the two fields our work involves: frameworks and query optimization. Chapter 3 introduces OPT++ in a reasonable detail so that the reader can understand our work. Complete documentation is available in Kabra's papers [Kab99] and Jinmiao's thesis [Li01]. In chapter 4, we will analyze the

PostgreSQL optimizer. The relational algebra, internal representation, transformative rules and search strategies are extracted so that they can be implemented or applied in our optimizer. Actually, the chapter can be viewed as requirement analysis and system specification of our optimizer. Chapter 5 describes our implementation based on the framework. Problems and implementation considerations are discussed in detail at the end of the chapter. In chapter 6, general issues in framework instantiation are discussed on the basis of related literature and our experience.

Unless otherwise indicated, all diagrams in the thesis are UML [Boo99] diagrams, and we discuss data types and programming mechanisms in terms of C++.

Chapter 2 Background

Basically, our work involved two fields: frameworks and query optimization. In our work, we applied framework technology to build an optimizer. Therefore, it is necessary to give brief background information about the two fields.

2.1 Frameworks

As a technique promising maximum reuse at many levels, object-oriented frameworks are a very active issue for both the software industry and academia.

2.1.1 What is a Framework?

According to Johnson [Joh88], a framework is a reusable, "semi-complete" application that can be specialized to produce custom applications. Framework technology has been studied for more than a decade. Frameworks have proven to improve productivity owing to their reusability. Frameworks can be reused in many levels such as domain knowledge, analysis, architecture, design and code. Unlike earlier reuse techniques based on class libraries, frameworks are domain specific. Frameworks are an object-oriented reuse technology, so by default, we refer to an object-oriented framework, when we talk about a framework.

The following are three common terminologies in frameworks.

- Hot spots [Pre94] are the general areas of variability within a framework where placing hooks is beneficial. A hot spot may have many hooks within it.
- Hooks are the places in a framework that can be adapted or extended to provide application specific functionality. A hook can be adapted in some way such as by filling in parameters or creating subclasses. Each hook description documents a problem or requirement that the framework builder anticipates an application developer will have, and provides guidance about how to use the hook and fulfill the requirement. They are the means by which frameworks provide the extensibility to build many different applications within a domain [Fro97].
- Frozen spots [Pre94] within the framework capture the commonalities across applications, in contrast to hot spots. They are fully implemented within the

framework and leave no work to the framework user. Typically there are no hooks associated with them.

2.1.2 Benefits of Frameworks

According to Fayad [Fay99], the primary benefits of frameworks stem from their modularity, reusability, extensibility, and inversion of control.

- Modularity — Using object-oriented technology, frameworks encapsulate volatile implementation details behind stable interfaces, and hence modularity is enhanced. When design and implementation changes within a module occur, the impact is localized without affecting other modules. The property stems from object-oriented technology.
- Reusability — The stable interfaces provided by frameworks guarantee that they can be reapplied to create new applications. Framework reusability makes it possible to use the domain knowledge and prior effort of experienced developers; therefore re-creating and re-validating common solutions to recurring problems are avoided. Reuse of framework components can improve productivity of programmers, as well as enhance the quality, performance, reliability and interoperability of software.
- Extensibility — Through inheritance and polymorphism of object-oriented languages, a framework can be extended easily by providing explicit hook methods to be overridden. That allows applications to extend its stable interfaces. Extending a framework is much easier than developing an application from scratch, so timely customization of new application services and features can be achieved.
- Inversion of control — Unlike a library, a framework calls custom code. At run-time, a framework calls custom code through polymorphism. In this way, the control flow of system is defined in the framework instead of applications. When events occur, the framework invokes corresponding hook methods on pre-registered handler objects, which are instances of custom subclasses of abstract classes defined in a framework. The objects of subclasses perform application-specific processing on the events.

Because these features of frameworks, framework-based applications need reduced time to market, provide improved maintainability and reliability, conform to industry and domain standards better, and embody domain expertise. In other words, framework-based development is faster, better and cheaper.

2.1.3 White-box Frameworks and Black-box Frameworks

A framework originates from a single application. Towards maturity, after many iterations, it will experience white-box stage and black-box stage, and finally reach an application generator. Both white-box and black-box frameworks can be reused, but the ways to reuse are different. White-box frameworks are reused by sub-classing, while black-box frameworks are reused by composition.

White-box Frameworks

When a framework is new, it tends to be white-box. A white-box framework requires the framework user to understand the internals of the framework to use it effectively.

A white-box framework is reused by sub-classing. Its behaviors can be extended by deriving subclasses from the abstract classes defined by the framework and implementing the hook methods. This is achieved by taking advantage of inheritance mechanism of object-oriented languages. A white-box framework often is provided in format of source code.

To reuse a white-box framework, we need to:

- understand how the subclass and super-class work together;
- have access to both the protected and the public parts of the class; and
- provide application specific functionality, by overriding existing methods, and implementing abstract methods.

Also, we can make use of the parent's methods.

Black-box Frameworks

As it evolves, a framework becomes more black-box. Now, it is reused by composition, instead of inheritance. Its behaviors are extended by composing components together, and delegating behavior between components. A black-box framework does not require a deep understanding of the framework's internal implementation in order for users to use it.

In black-box frameworks, objects tend to be smaller, and there tend to be more of them. The intelligence of the system comes as much from how these objects are connected together as much as what they do in themselves.

Composition tends to be more flexible than inheritance. Consider an object that uses inheritance versus one that delegates. With inheritance, the object basically has two choices: it can do the work itself, or it can call on the parent class to do it. Delegation is the idea that instead of an object doing something itself, it gives another object the task. With delegation, an object can do the work itself (or perhaps in its parent), or it can give the work to another object.

Application generators and even black-box frameworks sound so much easier to use than white-box frameworks. However, they cannot be reached without creating white-box frameworks first. To develop a black-box framework, we need to recognize components that provide application behavior, and define protocols among the components. Discovering this costs time. Relatively, white-box frameworks are easier to create.

The framework we used in our project is a white-box framework.

2.1.4 Frameworks and Design Patterns

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful to create a reusable object-oriented design. They capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities. [Gam95]

Frameworks are different from design patterns in such ways:

- Design patterns describe micro-architectures, while frameworks have concrete architectures.
- Design patterns are abstract, while frameworks are semi-implemented.
- Frameworks are domain specific, while design patterns are more general.

However, there are some connections between design patterns and frameworks

- Framework design may incorporate design patterns at the micro-architectural level.
- Flexibility for a framework is often achieved by applying design patterns.
- Design patterns help document a framework.

2.2 Query Optimization of Databases

Query optimization is key technology for database systems. An optimizer selects the optimal physical access plan for a declarative query. A user just tells a DBMS "what to do", and an optimizer will decide "how to do". To find an optimal physical access path, an optimizer enumerates (probably not all) possibilities of physical paths, and chooses the best one based on their estimated execution cost.

2.2.1 Overview

Both algebra and calculus can be used to model operation on relations in relational databases. SQL queries are represented with a few operators that form relational algebra.

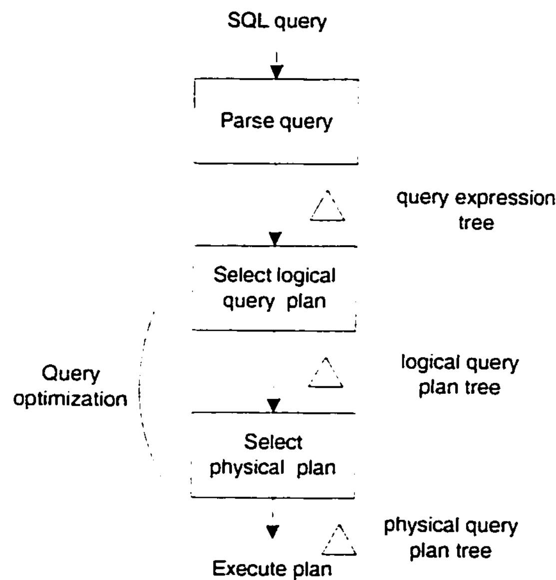

Figure 2.1 Schematic Diagram of Query Processing

The process of answering a query can be divided into three major steps, as sketched in Figure 2.1. [Gar00] A parse tree representing the grammar structure of the query is first generated. A logical query plan that is a tree composed of logical operators is generated. The tree may be reformatted into an equivalent based on some algebraic rules to achieve a better logical query plan. Physical plans are enumerated and the best one with the cheapest cost is selected. The last two steps are called query optimization. In practice, there may be no clear boundary between the two steps.

To discuss query optimization, it is necessary to make clear the concepts mentioned in above process.

Operator

An operator (also called a logical operator) is an operation in relational algebra. A query can be represented by a tree of operators. In an algebra that is capable of representing SQL queries, there are typically ten operators, union, intersection, difference (except), selection, projection, product, join, duplicate elimination, grouping, and sorting. Product can be viewed as a kind of special join, where there are no joinable tuples in the two input relations.

Algorithm

An algorithm (also called a physical operator) is a particular implementation of a logical operator. An operator may have more than one algorithm that can implement it. For example, the operator join, can be implemented with nested-loop, merge join or hash join algorithm. For the same input, the costs of different algorithms vary with the system statistics of a database. Given an algorithm on an input, we cannot know the exact execution cost of the algorithm without executing it, but we can estimate the costs based on the system statistics, and choose the best one.

A Logical Query Plan

A logical query plan is a tree of logical operators that represent the particular operations on data and the order of the operations. A logical query plan is also called a logical plan or an operator tree.

A Physical Query Plan

A physical query plan is built from algorithms. It represents a real physical access path to answer a query. It is also called an access plan, an algorithm tree, or a physical plan. It should be kept in mind that although, for the same query, different physical plans produce the same result, the execution costs are different. That is why optimization is needed.

According to Garcia-Molina [Gar00], to optimize a query, such decisions should be made:

- Selecting better logical trees. We can transform a logical plan into its equivalent with cheaper cost using algebraic rules. Note that no system statistics information is needed to do the transformation.
- Selecting an algorithm for each logical operator. We can execute only physical algorithms, so this must be done. Moreover, for the same operator, costs of different execution algorithms are different. Note that system statistics information is needed to estimate the costs in order for us to make a decision.
- Adding enforcers if necessary. Some algorithms require that its input have some kind of property, which must be obtained by executing an enforcer algorithm. For example, merge join requires that both its inputs have a common sort-order at the join attribute. This has to be achieved by adding a sort algorithm.
- Selecting the way in which arguments are passed from one algorithm to the next, by storing the intermediate result on disk or by using iterators and passing the intermediate result directly in memory.

2.2.2 Cost-based Plan Selection

To select an optimal plan with the cheapest cost, we have to know the costs of the possible plans. We cannot know the costs exactly without executing a physical plan, so we are forced to estimate the costs. In most cases, I/O is the principal cost of executing a query, so the cost can be estimated by I/O used to answer a query. That is:

$$\text{cost} \equiv \text{I/O cost}$$

I/Os are consumed to read in and write out the intermediate relations; therefore it is a function of the size of intermediate rel`ations. That can be represented as:

$$\text{cost}=f\,(\Sigma(\text{size of intermediate relations}))$$

The size of an intermediate relation equals the bytes per tuple times the number of its tuples. Namely:

$$\text{size} = \text{number of bytes per tuple} \times \text{number of tuples}$$

The former can be available in system statistics of a database, while the latter must be estimated on the basis of property of the logical operator and system statistics. Finally, we can estimate cost by the number of tuples of intermediate relations. That is:

$$\text{cost}=f\,(\text{number of tuples of intermediate relations})$$

There is no universally agreed-upon size-estimation method. Fortunately, the goal of size estimation is to help select a physical plan, instead of, to predict the exact size. Therefore, even an inaccurate size-estimation method will serve the purpose as long as it errs consistently. Some rules for estimation are suggested in Garcia-Molina's book [Gar00].

For a specific algorithm, the cost can be calculated on the basis of the estimated size of its input(s) and other parameters such as memory size available, which can be obtained from system statistics. The cost of a physical plan can be obtained by accumulating the costs of all its algorithms.

2.2.3 Search Strategies

A search strategy determines how the space of all physical plans is explored. Strategies can be classified into two categories, exhaustive and non-exhaustive, according to whether they explore all the space or not.

Exhaustive Strategies

When the search space is not too large to be managed, exhaustive strategies are preferred. Basically, exhaustive strategies check every possible physical plan to choose the best one; therefore, the optimum must be found. Dynamic programming is a typical exhaustive strategy, which will be explained further in latter chapters.

Non-exhaustive Strategies

When the search space is too large, the cost of optimization itself will become intolerable. In this case, non-exhaustive strategies will be adopted. The strategies cannot guarantee that the best plan is found, but it attempts to find a plan, which is almost as cheap as the best, in a limited time. One of the examples of such strategies is the randomized strategy. In our project, a randomized strategy with some heuristic information is implemented; it is called genetic algorithm,

It is worthwhile to point out that even an exhaustive strategy does not even necessarily explore all possible physical plans, because some of search space that must not contain the optimum can be excluded. For example, all logical trees in which there are selections but not pushed down should be excluded. Therefore, strictly speaking, exhaustive strategies search the space where the optimum must be.

In this chapter. background knowledge of framework and query optimization is introduced concisely. Our work was to apply the framework technology to build an optimizer. Because of suitable granularity of the query optimization problem, it can be regarded as a good example of framework-based application development.

Chapter 3 OPT++: An Object-Oriented Framework for Query Optimization

In this chapter, an object-oriented framework for query optimization, OPT++, is introduced. First, its architecture design is introduced, and then the three components in the architecture are presented in reasonable detail. Finally, the cost model and pruning mechanism used in the framework are explained.

3.1 Overview

Our optimizer is implemented within the OPT++ framework, so it is necessary to introduce the framework before we get to our implementation. For the framework user, understanding how to use a framework is a nontrivial task in the process of development.

OPT++ is a query optimization framework written in C++, developed by the University of Wisconsin. It is a white-box framework because it is reused by sub-classing. According to Kabra [Kab99], we can get such benefits to develop an optimizer based on it:

- Easy to develop

  Skeleton classes and control flow are well designed, and some search strategies are implemented in the design. An implementer needs only to subclass some abstract classes, and do domain specific design.

- Easy to extend/change

  Components (Algebra, Search Space and Search Strategy) are well decomposed. Therefore, each of its components can be changed with minimum impact on the other components.

- Easy to maintain

  With its modularity and clean program decomposition, OPT++ promotes sharing of code among different optimization schemes and implementations, leading, in turn, to improved maintainability.

- Efficiency

  Indirections are limited in the design level. At runtime, efficiency of OPT++-based optimizers is not affected by the late-binding mechanism offered by C++.

As we see, benefits claimed by OPT++ reflect the advantages of framework technology.

3.1.1 Basic Idea to Achieve Extensibility and Reusability

Like other white-box frameworks, OPT++ achieves extensibility and reusability chiefly through inheritance. 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. An abstract 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 in the search space. An optimizer for a specific database system can be written by sub-classing these abstract classes. Knowledge 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 and polymorphism mechanisms ensure that the search strategy of the optimizer will call methods of suitable subclasses at runtime. Figure 3.1 shows the basic system design of an OPT++-based optimizer[Kab99].

Furthermore, the search strategy itself is a class with virtual methods that can be overridden. Thus, new classes can be derived from this class to implement different search strategies. The optimizer-implementer can implement new search strategies by deriving new classes from the provided search strategy classes.

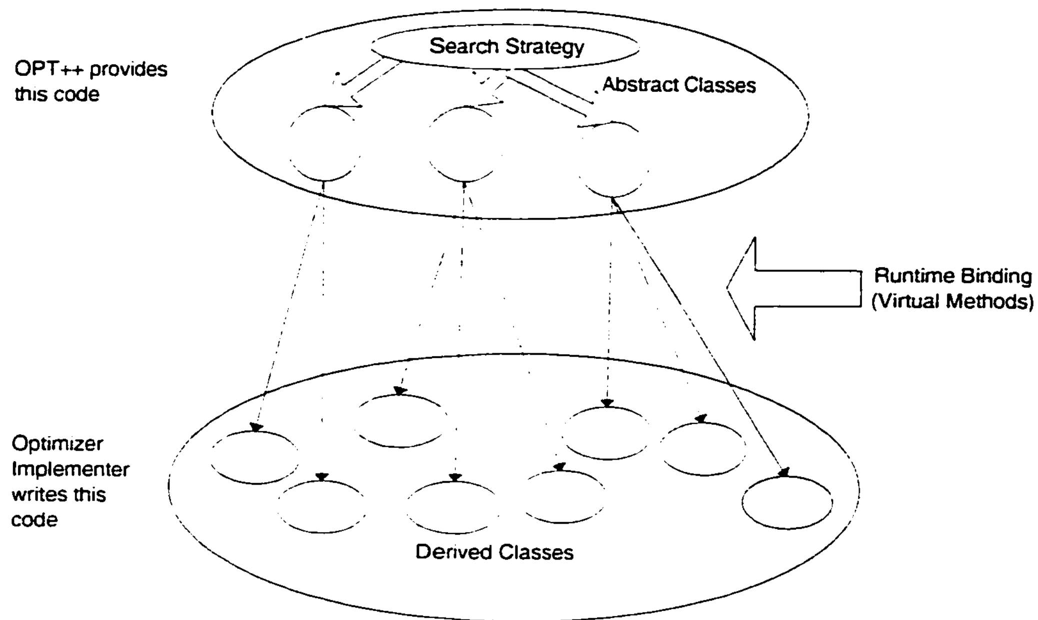

Figure 3.1 Basic System Design

3.1.2 Architecture

OPT++ is decomposed into 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 algebras for which the optimizer is written.

Figure 3.2 is a schematic diagram of the system architecture of an OPT++-based optimizer [Kab99].

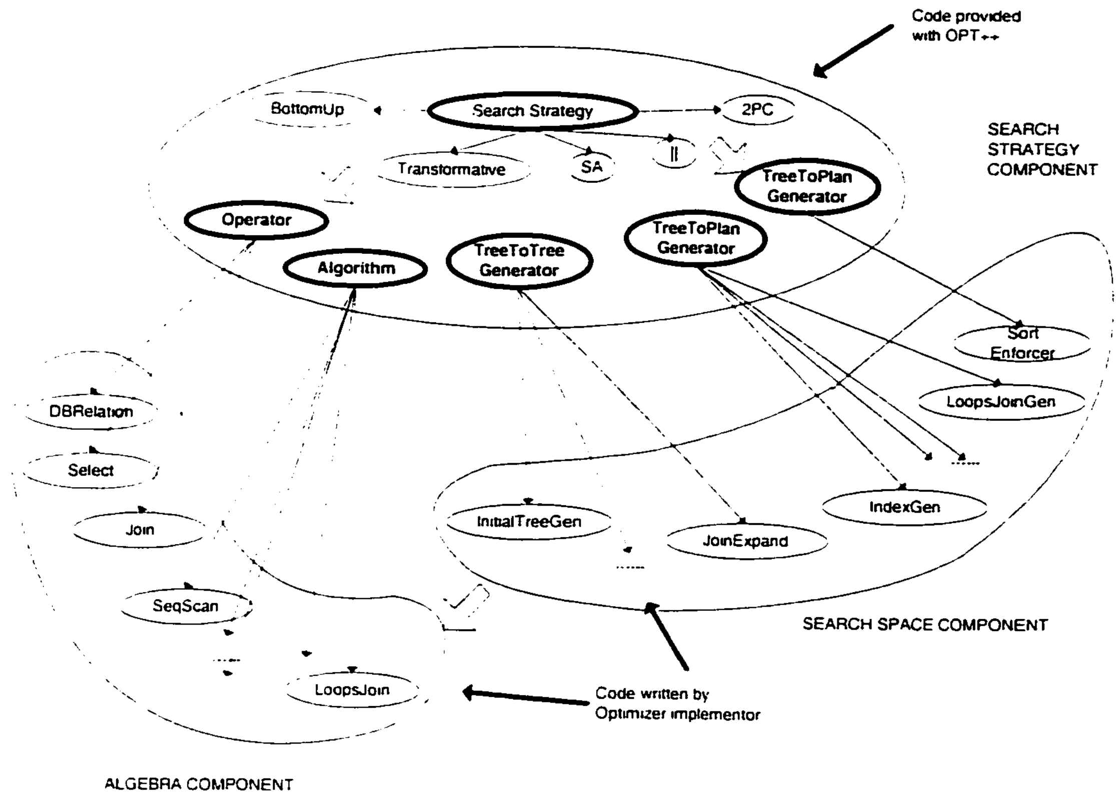

Figure 3.2 System Architecture

Jinmiao refined and re-documented the framework in an object-oriented style. Figure 3.3 is a UML class diagram [Li01] for the architecture of the framework.

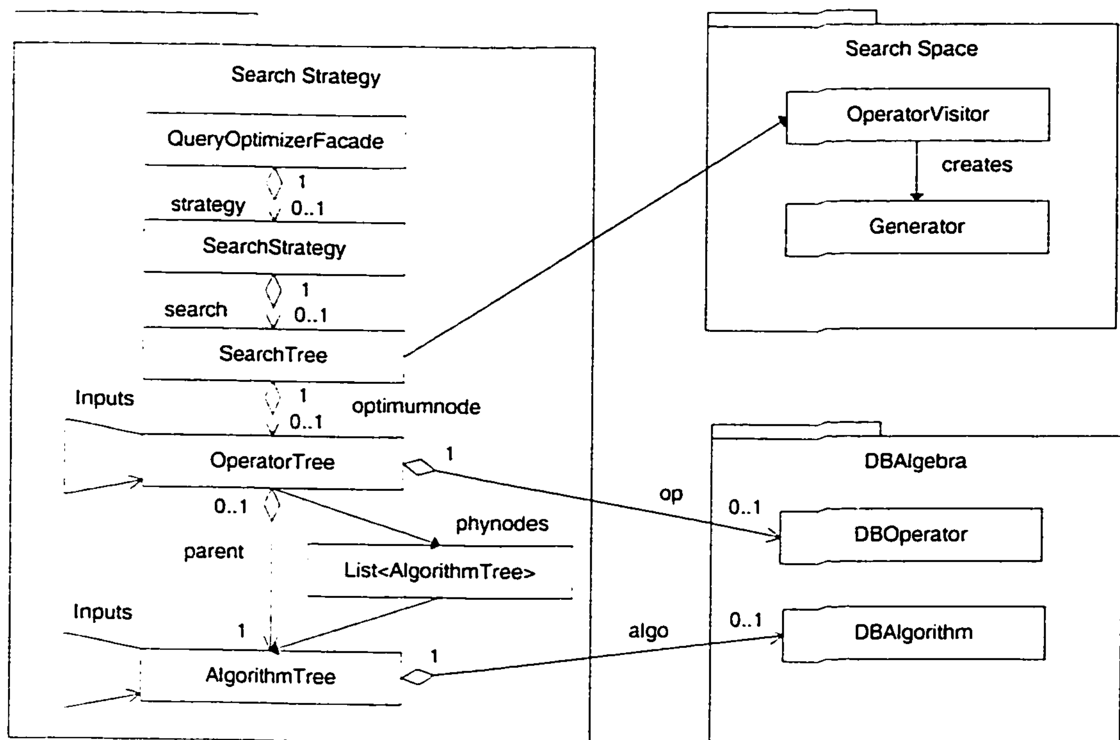

Figure 3.3 Overall Class Diagram

Note that some classes are renamed, and some classes reassigned among the components. For instance, the class **Operator** and **Algorithm** are renamed to be **DBOperator** and **DBAlgorithm**. Also they are classified into DBAlgebra component. Although the changes are not critical, we think the latter is more understandable.

The Algebra component defines the logical operators, the physical operators and their associations in a target database.

The Search Space component is used to decide what operator trees and access plans are generated, and hence plays a large part in controlling the search space that is explored by the search strategy.

The Search Strategy component encapsulates the system control. It implements different ways to perform search in the path space.

To our understanding, both the Search Space and the Search Strategy components are used to explore the algebra expression space, but Search Space is to control the space within a logical operator, while the Search Strategy component is used to control the inter-operator space.

3.2 The Algebra Component

The Algebra component defines logical and physical operators in a database. OPT++ defines some abstract classes and lets optimizer implementers define their own algebra by sub-classing them.

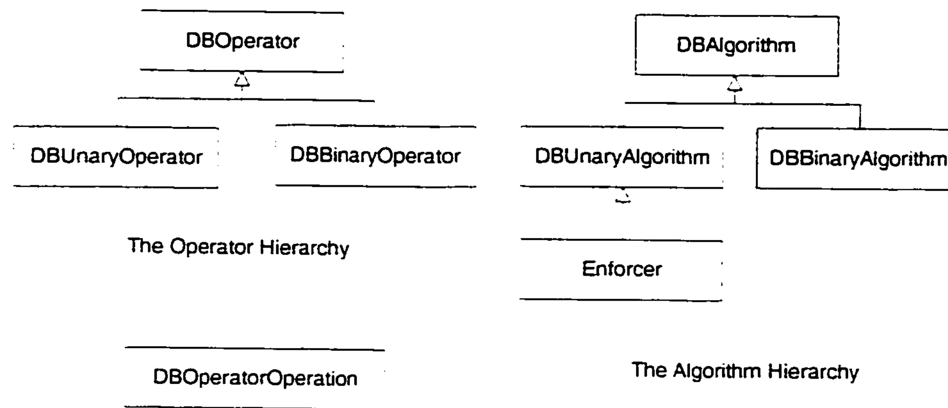

Figure 3.4 The Algebra Component

3.2.1 Classes and Hooks

The top abstract classes are class **DBOperator**, **DBAlgorithm**, **DBOperatorOperation**.

Class DBOperator

The abstract class represents logical operators in the query algebra. From the class, the optimizer-implementer is expected to derive one class for each operator in the actual query algebra. Besides real operators, such as SELECT and JOIN, database entities (for example relations and attributes) also can be operators. They will serve as leaf nodes in operator trees in the Search Strategy component.

According to the number of required inputs, operators can be classified into unary, binary and those that take no inputs. Because, with different arities, the behaviors of operators vary, two subclasses **DBUnaryOperator** and **DBBinaryOperator**, which are also abstract classes, are derived from class **DBOperator**. Actually, the optimizer-implementer needs to subclass the two classes in most cases.

Hooks:

Sub-classing: The optimizer-implementer is expected to derive one class for each operator in the actual query algebra.

Implementing abstract methods: Such methods must be implemented for their derived concrete subclasses:

- *Accept*—accepts a logical operator tree and tries to apply the operator to the tree. A logical tree to be accepted is encapsulated in a visitor class.
- *Duplicate*—copies itself. During the course of optimization, many instances of such operator classes are generated. For a class, all its instances are identical, so this method provides a convenient way to produce a new instance given an object.
- *MakeLogProps*—creates logical properties for the operator tree rooted at this operator. The tree is actually a sub-tree constructed so far. The method delegates the task to a constructor of class **OperatorTreeProperty** in SearchStrategy component.

The following method may be overridden:

- *Clones*—provides different ways to apply this operator to its input tree with different parameters. For example, for operator SELECT, given an operator tree, if there are restrictive qualifications on it, the method will return a list containing objects of class SELECT. Each of the objects encapsulates a restrictive qualification. If there are no restrictive qualifications on it, then an empty list is returned. The class **DBUnaryOperator** and **DBBinaryOperator** give the method a default implementation. However, according to our experience, the method needs to be overridden in most cases.

Class DBAlgoritm

The abstract class represents physical operators (i.e., execution algorithm) in the query algebra. From the class, the optimizer-implementer is expected to derive one class for each physical operator in the actual query algebra. For one logical operator, there may be more than one physical operator to implement it. For example, for logical operator DBRelation rerepsenting a relation, there can be FileScan (sequential scan) and IndexScan to implement it.

As in the logical operator hierarchy, physical operators can also be classified into unary, binary and those that take no inputs. Because, with different arities, the behaviors of physical operators vary, two subclasses **DBUnaryAlgorithm**, **DBBinaryAlgorithm**, which are also abstract classes, are derived from class **DBAlgorithm**. Particularly, in the algorithm hierarchy, there is a special class **Enforcer**, which represents the execution algorithms that do not correspond to any operator in the logical algebra. The purpose of these algorithms is to enforce physical properties in their outputs required by subsequent algorithms, instead of, to perform any logical data manipulation. For example, a Merge Join algorithm requires a SORT algorithm as an enforcer to make its input sorted on the join attribute.

Hooks:

Sub-classing: The optimizer-implementer is expected to derive one class for each algorithm in the actual query algebra.

Implementing abstract methods: The optimizer-implementer must implement such methods for their derived concrete subclasses in the algorithm hierarchy:

- *Duplicate*—the same meaning as in logical operator.
- *MakePhyProps*—creates physical properties for the algorithm tree rooted in at the algorithm. Like in class **DBOperator**, the method delegates its task to a constructor of class **AlgorithmTreeProperty** in the Search Strategy component.

Two another methods may be overridden:

- *MakePhyNodes*—builds the algorithm tree for an operator tree by delegating the task to the Search Space component.
- *Clones*—provides different ways to apply this operator to it inputs, with different parameters. According to our experience, we never override it.

For a concrete enforcer subclass, one has to implement such a method:

- *Enforce*—stipulates what to do when it is used to enforce an algorithm tree.

Class DBOperatorOperation

All concrete subclasses in the Operator hierarchy also must multi-inherit another class **DBOperatorOperation**. The class makes all its subclasses have a set containing expressions that are applied by the operator.

The class is the key to understanding how the framework judges equivalence of two logical trees and whether a logical tree is a complete tree, so it deserves more explanation. This will be described in the next section.

3.2.2 Equivalence of Two Logical Operator Trees

After a query is parsed, all expressions involved are put in a universal set. Note that besides normal expression such as arithmetic expression, Boolean expressions, both relations and attributes are viewed as expressions. Each logical tree keeps track of what expressions have been applied by keeping a set. Correspondingly, an operator must record what expressions are applied by it. When expression sets of two operator trees are equal, they are viewed to be equal, and normally, the one with expensive cost will be pruned if it does have some interesting property. If the expression set of an operator tree is equal to the universal set of the query, it is viewed to be a complete tree.

An example will make the idea more clear.

For a query: `SELECT * FROM employees AS e, departments AS d WHERE e.salary>1000 AND e.dept=d.id;`

Suppose the relation "employees" contains such attributes {name, dept, salary}, while "departments" contains {id, director, location}. The universal set will be {employees, departments, name, dept, salary, id, director, location, e.salary>25,e.dept=d.id}. The expression set of a SELECT object will contain {e.salary>25}, and the expression set of a JOIN object will contain {e.dept=d.id}.

Table 3.1 Expression Sets of Some Trees

Tree	Expression Set
t1	{employees, departments, name, dept }
t2	{employees, departments, name, dept, salary, id, director, location, e.salary>25}
t3	{employees, departments, name, dept, salary, id, director, location, e.salary>25, e.dept=d.id}
t4	{employees, departments, name, dept, salary, id, director, location, e.salary>25, e.dept=d.id}

Table 3.1 gives the corresponding expression sets of the operator trees in Figure 3.5. Note that by default implementation of OPT++, once a relation is introduced into an operator tree, all its attributes are introduced too. In the example, t3, and t4 will be viewed to be equal, and also they are complete trees.

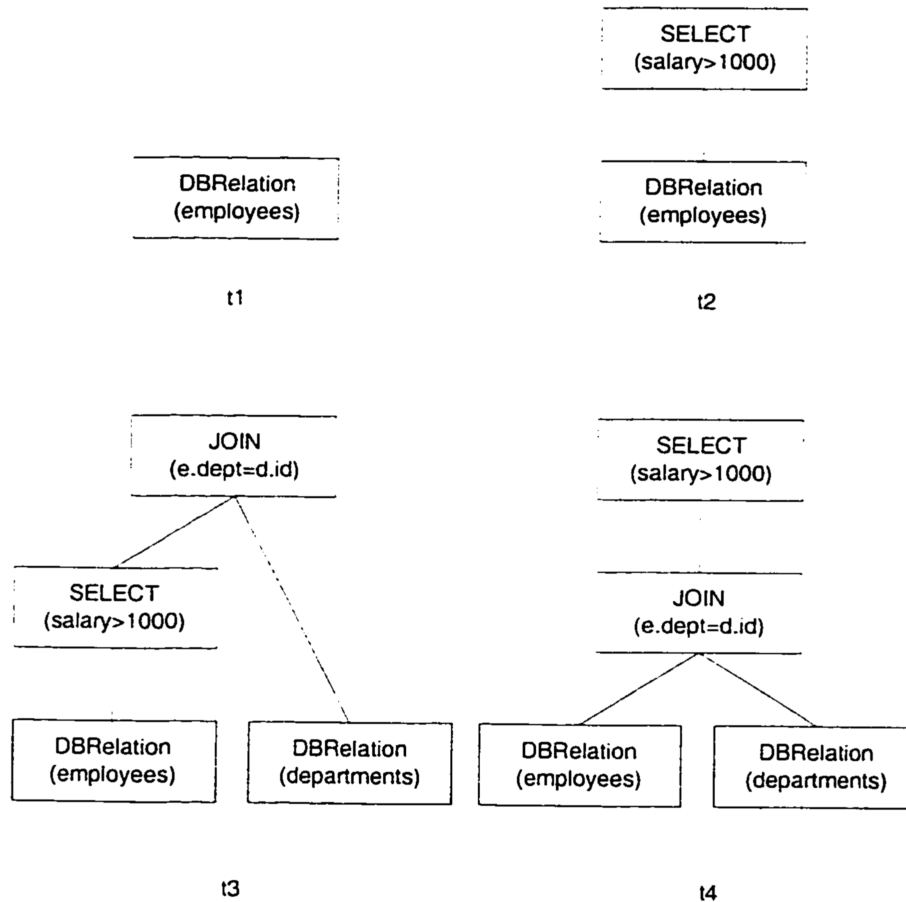

Figure 3.5 Some Logical Trees

3.3 The Search Space Component

The component defines that what should be done when a new operator is applied to an operator tree (a part of the complete tree). It controls the search space in one operator level. Actually, the component is associated with the search strategy component, which controls how to search the plan space in inter-operator level. As declared by Kabra [Kab99], the search space subclasses can be implemented without making assumptions

about the search strategy, but that is less efficient. Implementing the component based on a specific strategy will improve efficiency, but decrease the extensibility of the target system. As you will see in chapter 5, our instantiation has to balance extensibility and efficiency.

3.3.1 Design Patterns Used

Jinmiao [Li01] redesigned the Search Space component by applying visitor design pattern [Gam95]. There are two hierarchies in the components, the visitor hierarchy and generator hierarchy. The component performs operations on the logical algebra. The visitor hierarchy just dispatches the tasks to operator-specific generators in the generator hierarchy, where the real operations are done.

Visitor Design Pattern

As in the general visitor pattern, a logical operator object calls a visitor's corresponding method in its *Accept* method and passes itself as a parameter. The logical tree to be processed is encapsulated in the visitor when the visitor is initialized. Then the visitor performs operator-specific operation on the tree by delegating the task to corresponding generator classes.

Figure 3.6 shows the procedure when a Join object accepts an **ExpandTreeVisitor** in UML sequence diagram. When an object of class **Join** accepts a visitor, it just calls the corresponding method defined in the visitor class, *VisitJoin* in the diagram. Then the Join object is processed in the method, by delegating the task to the generator classes.

Note that Figure 3.6 does not show the complete process. The treatment of converting a logical tree to physical tree(s) is omitted.

The Visitor pattern makes the design more flexible, because it makes it easy to add a new operation. The framework provides hooks letting the user to add a new method to process a new operator easily.

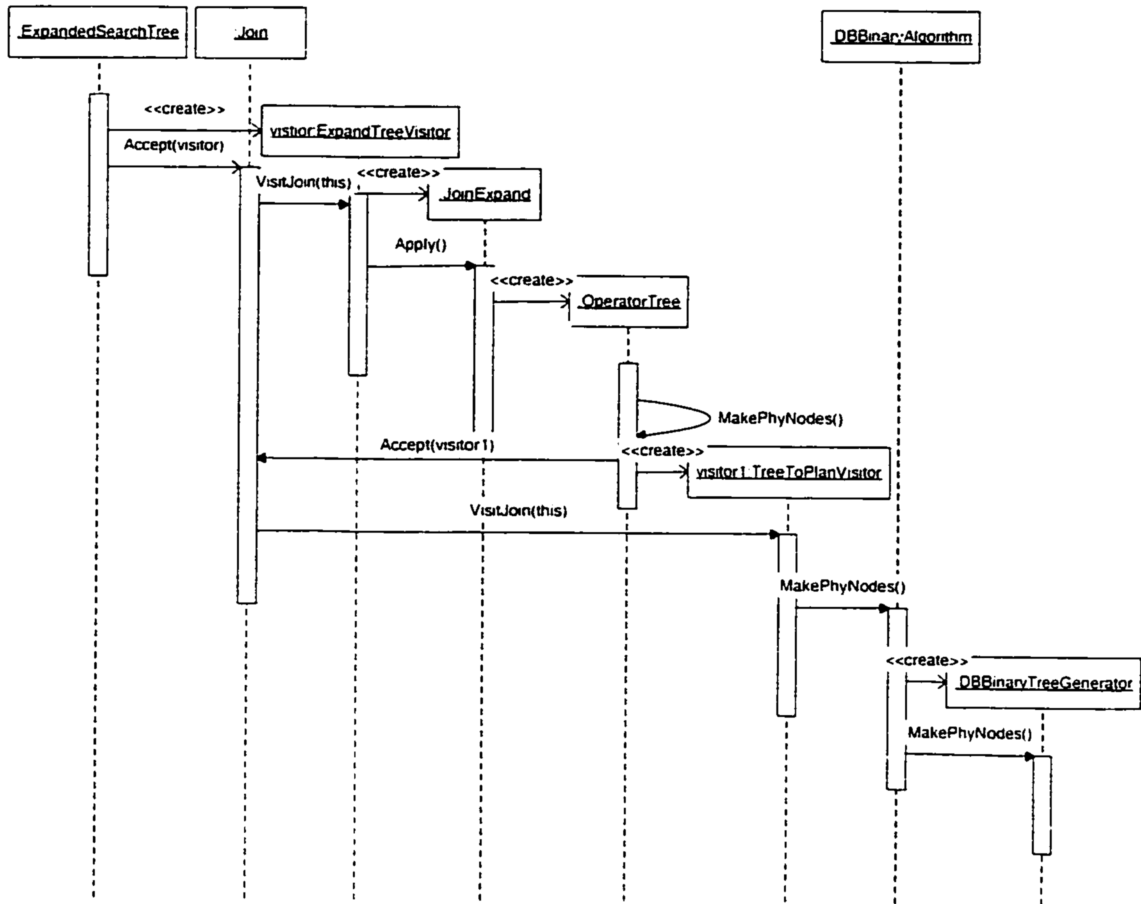

Figure 3.6 The Sequence Diagram for Visitor Pattern in the Search Space Component

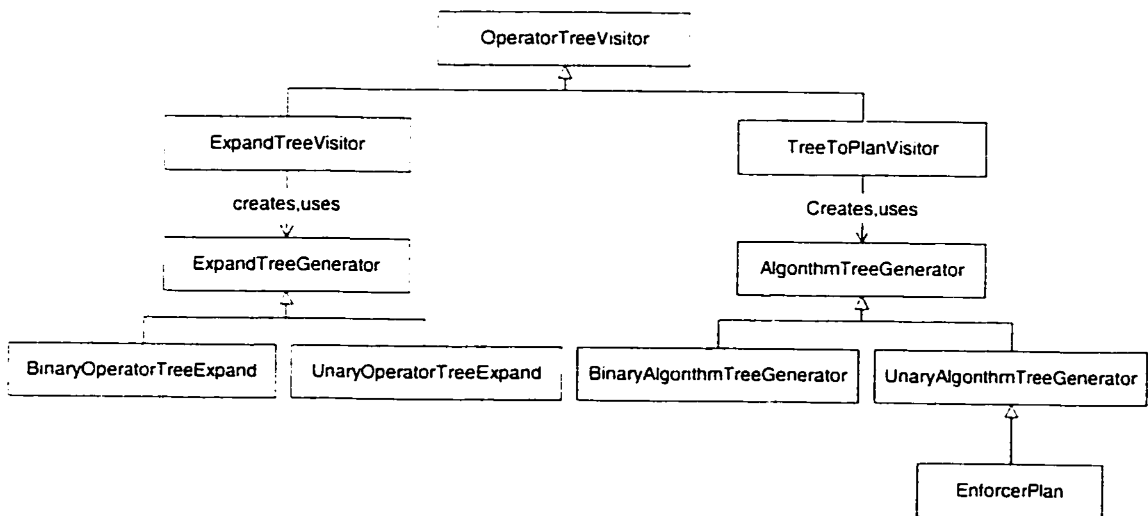

Figure 3.7 The Search Space Component

Figure 3.7 depicts main classes in the Search Space component provided by OPT++.

3.3.2 Classes and Hooks

3.3.2.1 The Visitor Hierarchy Classes

The visitor hierarchy defines three abstract classes. **OperatorTreeVisitor**, **ExpandTreeVisitor** and **ExpandTreeVisitor**. The generator hierarchy implements the major functionality in the Search Space component. The optimizer-implementer must derive concrete subclasses from class **ExpandTreeGenerator**, and **TreeToPlanGenerator**. The subclasses will be called by their corresponding **OperatorTreeVisitor** subclasses. The optimizer-implementer may define their own Visitor and Generator hierarchy parallel to **ExpandTreeVisitor** and **ExpandTreeGenerator** in different ways, but generally speaking, the optimizer-implementer does not need to change the **TreeToPlanVisitor** and **TreeToPlanGenerator** hierarchy.

The OperatorTreeVisitor Class

The **OperatorTreeVisitor** is the root class the visitor hierarchy. It is partially designed in the framework. It defines an attribute `currentTree` to represent the current logical operator tree to be processed. The class is an abstract class and leaves a series of pure virtual functions *VisitXX*. The functions must be implemented in its concrete subclasses by the optimizer-implementer. Here, XX refers to the name of a logical operator.

The ExpandTreeVisitor class

The **ExpandTreeVisitor** is a subclass of **OperatorTreeVistor**. It is designed to delegate the task to an operator's corresponding generator subclass. For example: We can define our own subclass of **OperatorTreeVisitor** and dispatch the tasks to our own generator classes. According to our experience, the **ExpandTreeVisitor** can be reused without changes.

The TreeToPlanVisitor class

The **TreeToPlanVisitor** subclass is designed to convert an operator tree to its corresponding algorithm trees. The visitor classes dispatch their responsibility to the

method, *MakePhyNodes*, of a concrete class. In turn, the method of a concrete generator class is called to perform the operation.

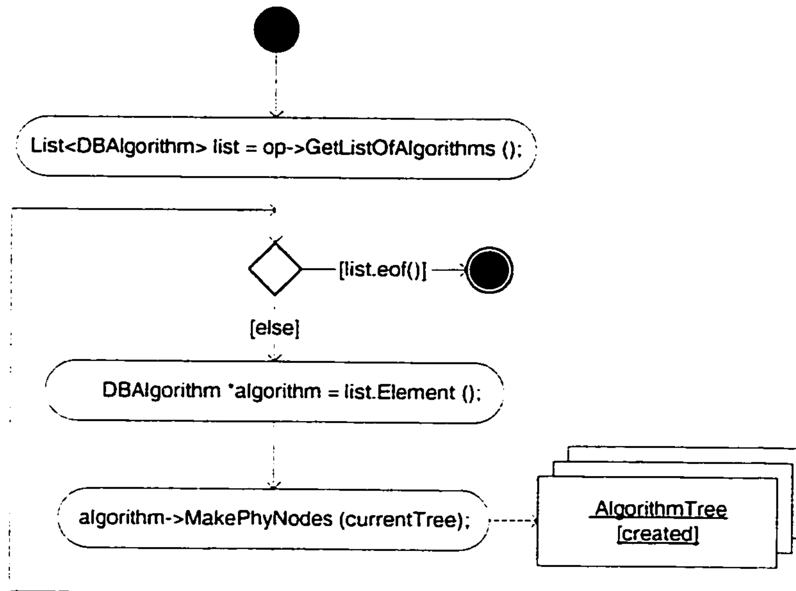

Figure 3.8 The Activity Diagram for the Method *VisitDBOperators*

All the methods like *VisitXX* in the **TreeToPlanVisitor** class are implemented in the same way. A private method, *VisitDBOperators*, is called to enumerate all physical trees according to algorithms associated with the logical operator. This can be explained in Figure 3.8.

First, the algorithms associated with the operator to apply are put in a list. Then, for each algorithm, its *MakePhyNodes* method is called to try to generate a physical tree. The concrete algorithm class calls the *MakePhyNodes* method of its corresponding generator class to do the job in turn. This is shown in Figure 3.6.

The hooks in the visitor hierarchy are:

Sub-classing: The optimizer-implementer can derive a new class from **OperatorTreeVisitor** to the search space in a different way. Nevertheless, because the visitor class in the framework just dispatches its tasks to corresponding generator classes, we can change the behavior of the Search Space component by changing the generator classes. Therefore, there is no need to derive a new class in practice.

Adding new methods: The optimizer-implementer must add a *VisitXX* method to each class in the hierarchy.

- The optimizer-implementer must add a *VisitXX* pure virtual method for each logical operator XX.
- The optimizer-implementer must implement a series of *VisitXX* methods in the subclasses to expand an operator tree and to convert an operator tree to algorithm tree(s).

According to our experience, for class **TreeToPlanVisitor**, we just implement the method *VisitXX* by calling the private method *VisitDBOperators*.

3.3.2.2 The Generator Hierarchy Classes

The generator hierarchy does not have a common root class, but for each subclass of class **OperatorTreeVisitor**, there should be a corresponding generator. There are two top generator classes, **ExpandTreeGenerator** and **AlgorithmTreeGenerator**.

Class ExpandTreeGenerator

It is to expand an operator tree. Class **UnaryOperatorExpand** and **BinaryOperatorExpand** are defined as subclasses of the class by the framework. Common behaviors are generalized in the two subclasses according to the number of their inputs. Normally, we subclass the two classes, instead of **ExpandTreeGenerator**, and override the method, *apply*.

The method, *apply*, of **UnaryOperatorExpand** is implemented as shown in Figure 3.9. First the *Clones* method of the operator to be applied is invoked to determine whether the operator can be applied to the given operator tree. Then the constructor of class **OperatorTree** is called to generate a new expanded logical tree.

The new created logical trees are passed to the search tree and preserved there.

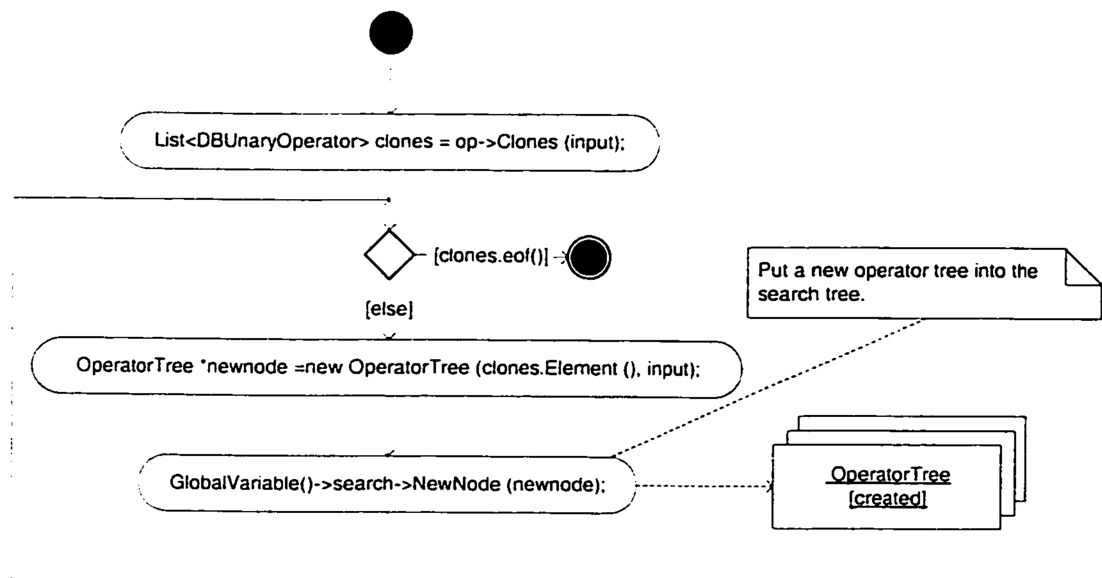

Figure 3.9 The Activity Diagram for the Method UnaryOperatorExpand::apply

Hooks:

Sub-classing: A subclass, such as **JoinExpand**, must be defined for each operator. It should be defined based on **UnaryOperatorExpand** or **BinaryOperatorExpand**. The method *apply* can be overridden if the optimizer-implementer does like the default behaviors of them.

Overriding: The method *Apply* is to apply an operator to a given operator tree. It is implemented in a default way. The optimizer-implementer can override it. Normally, we need to override it.

Class AlgorithmTreeGenerator

It is an abstract class to convert a logical tree to physical tree(s) according to the property of algorithm being applied. The two subclasses provided by the framework, **UnaryAlgorithmTreeGenerator** and **BinaryAlgorithmTreeGenerator**, define common behaviors when there are one or two inputs respectively. Like in class **ExpandTreeGenerator**, we subclass generally the two classes and override their methods, instead of **AlgorithmTreeGenerator**.

As we can see in Figure 3.6, a concrete algorithm class calls the method *MakePhyNodes* of an algorithm generator class to generate a physical plan. The class

then calls its method, *apply*, to do the job. The default implementation of method *MakePhyNodes* and *Apply* **UnaryAlgorithmTreeGenerator** is shown in Figure 3.10 and Figure 3.11. The two methods of class **BinaryAlgorithmTreeGenerator** are implemented in the similar way.

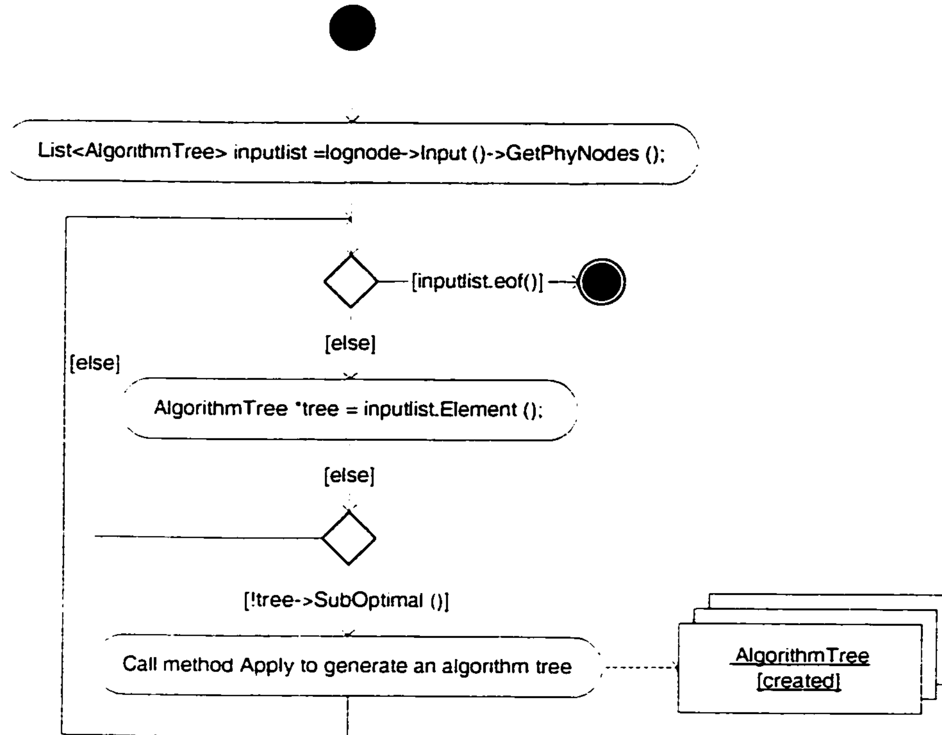

Figure 3.10 The Activity Diagram for UnaryAlgorithmTreeGenerator::MakePhyNodes

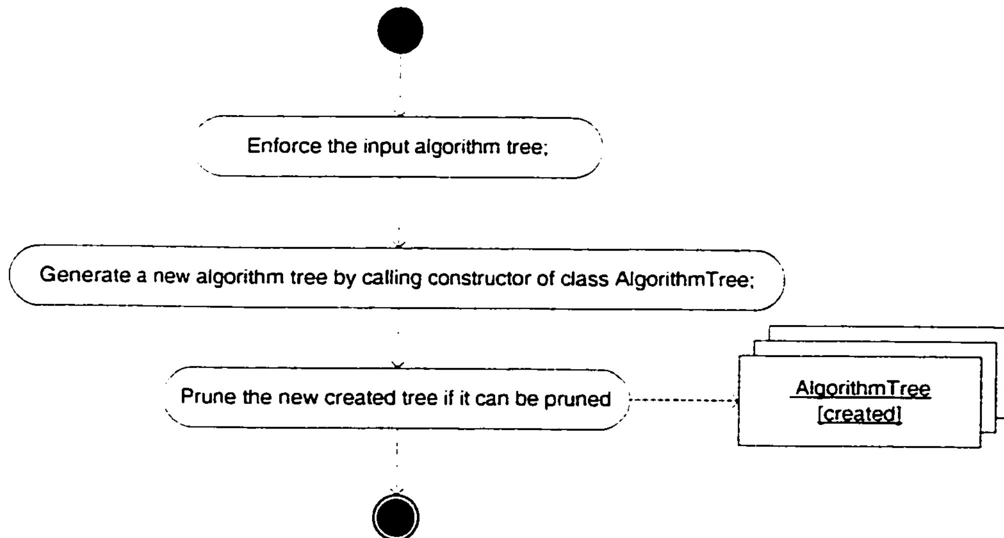

Figure 3.11 The Activity Diagram for UnaryAlgorithmTreeGenerator::Apply

As we see in Figure 3.10 and Figure 3.11, the method *MakePhyNodes* picks one algorithm tree from its input operator tree (there may be more one algorithm tree associated with an operator tree) at a time, and calls *Apply* to generate a physical plan using the input algorithm tree and the algorithm to be applied.

Hooks:

Sub-classing: A subclass, such as **JoinPlan**, must be defined for each operator. It should be defined based on **UnaryAlgorithmTreeGenerator** or **BinaryAlgorithmTreeGenerator**.

Overriding: The following methods can be overridden to change the way to generate an algorithm tree given an algorithm and input algorithm tree(s), if the optimizer-implementer does not like the default behaviors.

- *Apply*—apply an algorithm to its input algorithm trees.
- *MakePhyNodes*—an abstract method that is supposed to enumerate all possible combinations of inputs.
- *CanBeApply*—if the current algorithm can be applied to a given algorithm tree.

Our experience indicates we do not need to override them in most cases.

3.4 The Search Strategy Component

The search strategy Component encapsulates the system control flow. It implements different search approaches to find the optimal plan in the inter-operator level. At any time, there is only one search strategy dominating. The component also encapsulates the cost model and makes cost estimation a hot spot. It preserves all logical and physical plans generated up to the current time by maintaining an aggregation reference to the Algebra component, and dynamically prunes the sub-optimal trees that do not have interesting properties. Figure 3.12 is a class diagram for the Search Strategy component.

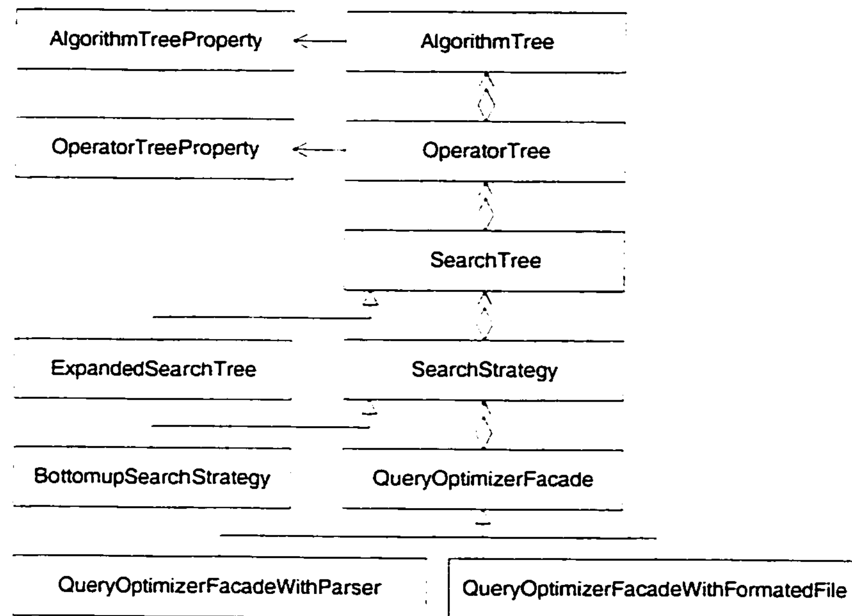

Figure 3.12 The Search Strategy Component

3.4.1 Design Patterns Used

There are three design patterns used in the components: Facade, Strategy and Factory Method.

The Facade pattern [Gam95] is used to provide a unified interface for the query optimization subsystem.

The motivation of the Strategy [Gam95] design pattern is to define a family of algorithms (called strategies), and to make them interchangeable. The idea conforms to the need for extensibility of changing the search strategy in query optimization. Therefore, it is natural to use the Strategy pattern. Typically, there will be only one strategy used when the framework is instantiated, but we do not know which one it is while designing the framework, so we cannot instantiate the concrete classes associated with the strategy. Therefore, the Factory Method pattern is adopted to defer the decision and let the concrete strategy decide what classes should be instantiated.

In Figure 3.12, **SearchStrategy** is an abstract class. We can define many concrete search strategies by sub-classing it. To make them interchangeable, a pointer to class **SearchStrategy** is defined as an attribute in the class **QueryOptimizerFacade** that uses the strategies. The code is:

```
SearchStrategy * strategy;
```

To change a strategy, we only need to initialize a concrete strategy class, for example, **BottomupSearchStrategy**, and assign its reference to the attribute. The code is:

```
strategy=new BottomupSearchStrategy;
```

In this way, Strategy pattern is implemented.

As it will be described in next section, the class **SearchStrategy** needs to preserve a reference to a concrete search tree class associated with a concrete search strategy class. However, because the class **SearchStrategy** is an abstract class, it does not know what concrete search tree class should be created while being initialized. As a result, there is only a protected attribute defined as an pointer to the abstract class **SearchTree**, like:

```
SearchTree* search;
```

An abstract method *CreateSearchTree* is defined in class **SearchStrategy**. The method is implemented in its subclass. For example, in its subclasss, **BottomupSearchStrategy**, the method is implemented as:

```
return new ExpandedSearchTree;
```

At runtime, a concrete search strategy class, for example, **BottomupSearchStrategy**, is instantiated, and then it will call the method *CreateSearchTree* to initialize the attribute, **search**. The code is:

```
search= CreateSearchTree();
```

This is how the Factory Method pattern is implemented.

The two patterns make it easy to change strategies.

3.4.2 Classes and Hooks

Class QueryOptimitztionFacade

It is an abstract class defining an entry point to the query optimization system. It has two concrete subclasses, **QueryOptimitztionFacadeWithFile** and **QueryOptimitztionFacadeWithParser**. By implementing the method, *PreprocessQuery*, in different ways, they behave differently when accepting queries in different formats.

Class SearchStrategy

It is abstract class.

Main methods:

Optimize—a public method to optimize the user query and return the optimal physical plan.

CreateSearchTree—the protected method that is supposed to be implemented in concrete subclasses to create their corresponding search trees.

Main attribute:

`search`—a private attribute representing the actual search tree (an instance of the **SearchTree** class)

Hooks:

Sub-classing: The optimizer-implementer must subclass the class **SearchStrategy** for a self-defined strategy, and implement the method *CreateSearchStrategy*. The class **BottomUpSearchStrategy** is just an example strategy.

Implementing the abstract method:

- *CreateSearchTree*—the protected method must be implemented in concrete subclasses to create their corresponding search trees.

The SearchTree class

It is an abstract tree representing a search tree that is used to explore the search space. **ExpandedSearchTree** is an example subclass of the **SearchTree** class, created by the class **BottomupSearchStrategy**.

The main attribute:

`listofunexpandednodes`—the important attribute is an array of lists, which are used to store the uncompleted operator trees generated during optimization.

The main method:

Prune—this is a critical static method used to prune the sub-optimal non-interesting trees dynamically based on the cost of physical trees. The method is critical to understand the efficiency of the framework, so it is explained in detail in 2.5.2.

Hooks:

Sub-classing: The optimizer-implementer must derive a subclass from the class **SearchTree** for the self-defined search strategies.

Implementing the following abstract methods:

- *MakeInitialTree*—a public abstract method. It is overridden to create initial operator trees. The initial tree depends on the concrete search tree derived from the class. For the Bottom-up strategy, an initial tree is just a leaf node representing a relation, while in the transformation strategy, it is a randomly created complete tree to be transformed. The optimizer-implementer must create initial tree(s) in a way that is suitable to later search.
- *DoSearch*—a public abstract method, which performs the search on the search tree. Different strategies implement it in quite different ways. For example, in the Bottom-up strategy, it expands a tree (it may be an initial tree or a resultant tree of one iteration of this process) by trying to apply all operators to it repeatedly until complete trees are created, while in the transformation strategy, the initial tree is transformed by applying transformative rule one at a time. The optimizer-implementer must build the search tree based on the initial operator trees(s) using a customized search strategy.

The OperatorTree class

The class is completely implemented, so no actions need to be taken.

This class is used to represent a logical operator tree. Actually, an object of the class is a node of such a tree. A node takes none, one or two inputs, which are also objects of the class, so a tree is constructed. The class, and **OperatorTreeProperty**, **AlgorithmTree** and **AlgorithmTreeProperty**, are the most successfully designed classes.

The class **OperatorTree** keeps a reference to an object of class **DBOperator**. It screens all operator specific details, but preserves general information required by optimization. It also keeps a reference to an object of class **OperatorTreeProperty**, which records the property of the current logical operator tree. Besides these, it holds a list containing all its corresponding algorithms.

Its main attributes are shown in table 3.2.

Table 3.2 The Main Attributes of Class OperatorTree

Name	Description
`op`	A pointer to an object of class **DBOperator**, which is current operator to be applied.
`inputs`	An array of pointers to its input(s), which are also objects of class OperatorTree.
`phynodes`	A list of objects of class **AlgorithmTree** associated with the operator tree.
`logprops`	An pointer to an object of class **OperatorTreeProperty**, which is used to represent logical property of the operator tree.

Class OperatorTreeProperty

The class is used to calculate and store logical properties of an operator tree. Each logical tree has an object of class **OperatorTreeProperty** associated with it. The properties are mainly used to determine equality of two logical trees and to calculate costs of its associated physical plans.

Its main attributes are shown in table 3.3.

Table 3.3 The Main Attributes of Class OperatorTreeProperty

Name	Description
`_numtuples`	The number of output tuples. This is used to calculate costs of its associated physical plans
`_operations`	A set containing operations (i.e. expressions) applied to the tree so far. It is accumulated when an operator tree grows. See 2.2.
`_index_path`	A string representing the index path name which can be used for an index scan.
`_is_interesting`	An integer indicating whether the associated operator tree has some interesting properties.

Table 3.4 lists its main methods.

Table 3.4 The Main Methods of Class OperatorTreeProperty

Name	Description
IsEqualTo	A public method that compares the logical properties of two operator trees.
IsCompleteQuery	A public method to judge whether a given tree is complete.

Hooks:

Adding a new method: The optimizer-implementer must add a constructor to the class **OperatorTreeProperty** for each logical operator XX with format:

```
OperatorTreeProperty (XX*, OperatorTree*);
```

In this way, the framework associates the specific operators with the general representation of operator trees.

In one constructor, at least such things must be done:

- Calculating `_numtuples`;
- Calculating `_operations`.

The optimizer-implementer may

- Redefine what is `_is_interesting`;
- Add some new logical properties.

Class AlgorithmTree

Similar to class **OperatorTree**, class **AlgorithmTree** is used to represent a physical algorithm tree. The main attributes are shown in table 3.5. Like class **OperatorTree**, the class is completely implemented, so no actions need to be taken.

Table 3.5 The Main Attributes in Class AlgorithmTree

Name	Description
`Algo`	A pointer to an object of class **DBAlgorithm**, which is current algorithm to be applied.
`Inputs`	An array of pointers to its input(s), which are also objects of class **AlgorithmTree**.
`Parent`	An pointer to an object of class **OperatorTree**, with which the algorithm tree is associated.
`phyprops`	An pointer to an object of class **AlgoriothmTreeProperty**, which is used to represent physical properties of the operator tree.
`suboptimal`	A Boolean attribute indicating whether the algorithm tree is sub-optimal. An algorithm tree is marked as sub-optimal when it has higher estimated execution cost, but used by other algorithm trees. In this case, we cannot delete it.

Class AlgorithmTreeProperty

The class contains the physical properties for an algorithm tree (physical query plan), such as estimated execution cost, sort-order, etc.

Table 3.6 The Main Attributes in Class AlgorithmTreeProperty

Name	Description
`_cost`	An object of class **Cost**. It contains the estimated execution cost for the current algorithm tree.
`_is_interesting`	An integer indicating whether the associated algorithm tree has some interesting properties. Whether an algorithm tree is interesting will be defined by the optimizer-implementer in a constructor of the class.

Hooks:

Adding a new method: The optimizer-implementer must add a constructor to the class **AlgorithmTreeProperty** for each logical operator XX with format:

```
AlgorithmTreeProperty (XX*, OperatorTree*)
```

In this way, the framework associates the specific operators with the general representation of operator trees.

In one constructor, at least such things must be done:

- Calculating `_cost`. This is implemented by delegating the task to class Cost;
- Calculating the size of an output tuple.

The optimizer-implementer may

- Redefine what is `_is_interesting`;
- Add some new physical properties.

Class Cost

The class is defined to compute the estimated execution cost, given an algorithm, and the algorithm tree to be constructed. The cost model of the optimizer is encapsulated in the class.

Hooks:

Overloading a method: The optimizer-implementer must overload the method, *compute*, for each physical algorithm. In such a method, the estimated execution cost is computed based on physical properties. The optimizer-implementer may change the cost model in the method.

3.5 Cost Evaluation and Dynamic Pruning

Cost evaluation and dynamic pruning are two of core functions of the framework. The cost model is a hot spot, which the optimizer-implementer must implement. Dynamic pruning mechanism deletes the sub-optimal operators and algorithm trees that are determined not to be parts of the complete optimal tree.

3.5.1 Cost Evaluation

There are different cost models for different situations. The most common used model is to use the disk I/O to estimate the execution cost of an algorithm [Gar00]. In the framework's sample model, both I/O cost and CPU cost are considered.

In the framework, the cost model is designed as a hot spot.

To calculate cost of an algorithm, the following information is needed:

- Estimated sizes of the inputs. This is calculated by constructors of class **OperatorTreePropperty**, which are provided by the optimizer-implementer.

- Selectivity of the predicates applied by the algorithm. This is computed by the method *Selectivity* of the class **Expression**, which can be re-implemented easily to change selectivity of predicates without affecting any other part of the framework.
- Estimated number of instructions required to execute the algorithm and other database statistics, such as the size of memory available, page-size, I/O cost per page. The data are available in system catalog.

3.5.2 Dynamic Pruning

In the course of optimization, many sub-trees and complete trees are generated. The optimizer compares a tree to its equivalents. The ones with higher cost and without interesting properties will be deleted as early as possible. The advantages of the dynamic pruning are:

- Saving time: For some operator trees, if the strategy can decide that they will not be the optimal tree, deleting them will save a lot of time, because we do not need to expand them further. The earlier such trees are deleted, the more time can be saved.
- Saving space: The unnecessary trees take up a lot memory. Especially, in C++, memory management is a critical problem. If we do not delete them properly, memory leakage will occur.

To prune the trees, the framework has to find equivalents for a given tree. The framework provides well-designed functionality to do that. It determines whether two trees are equal on the basis of equivalence of two expression sets of two trees using a hash mechanism. More details are described in Jinmiao's thesis.

Even though we find a sub-optimal equivalent, to delete a physical node is nontrivial. If it is deleted improperly, memory leakage may occur. Also, the system will be corrupted, because some other objects reference a non-existent node. Fortunately, the framework considers all situations.

Figure 3.13 shows the function to delete an algorithm tree. Deleting a physical node may cause its parent, an operator tree, to be deleted. The procedure is shown in Figure 3.14.

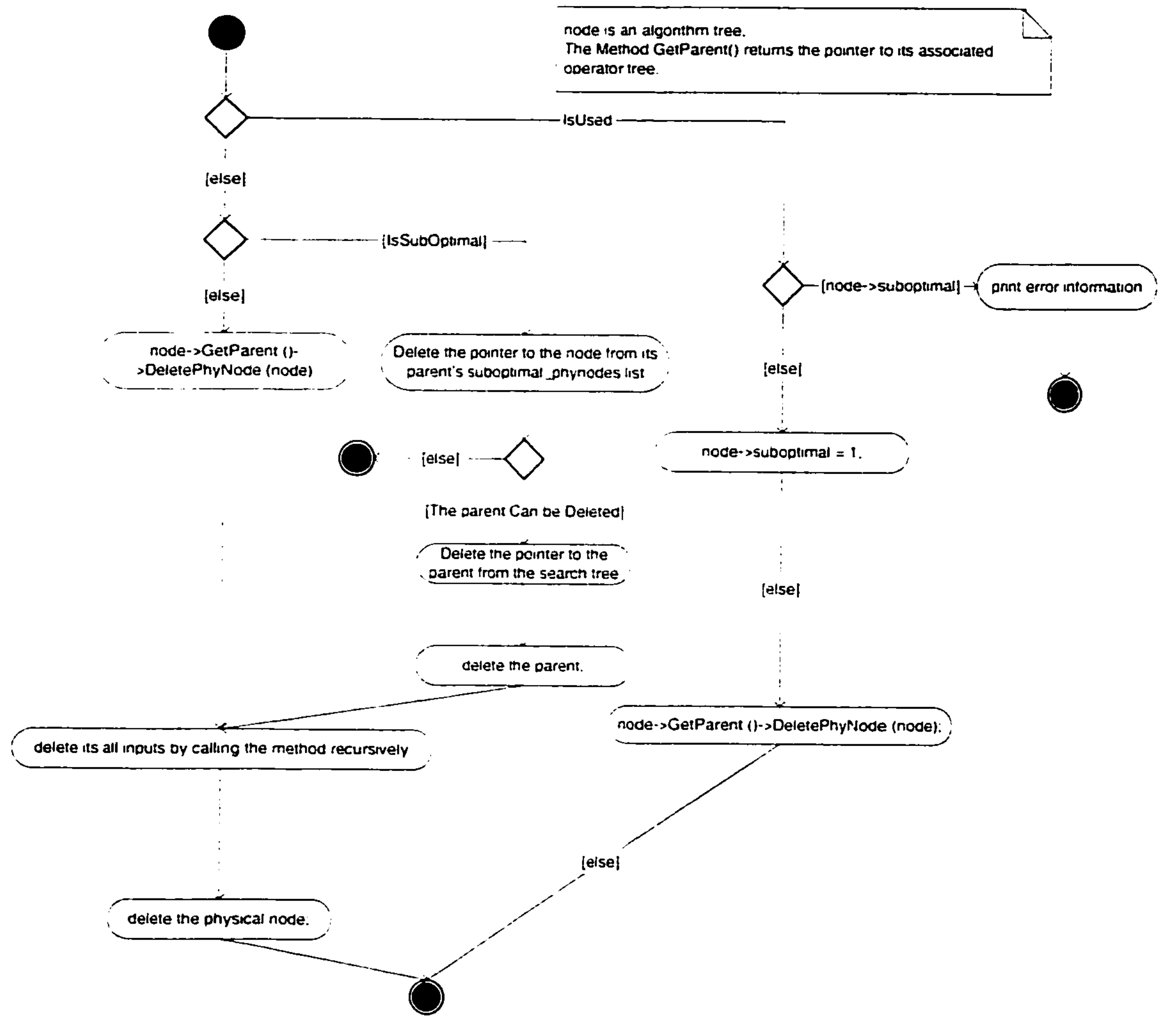

Figure 3.13 The Activity Diagram for Deleting an Algorithm Tree

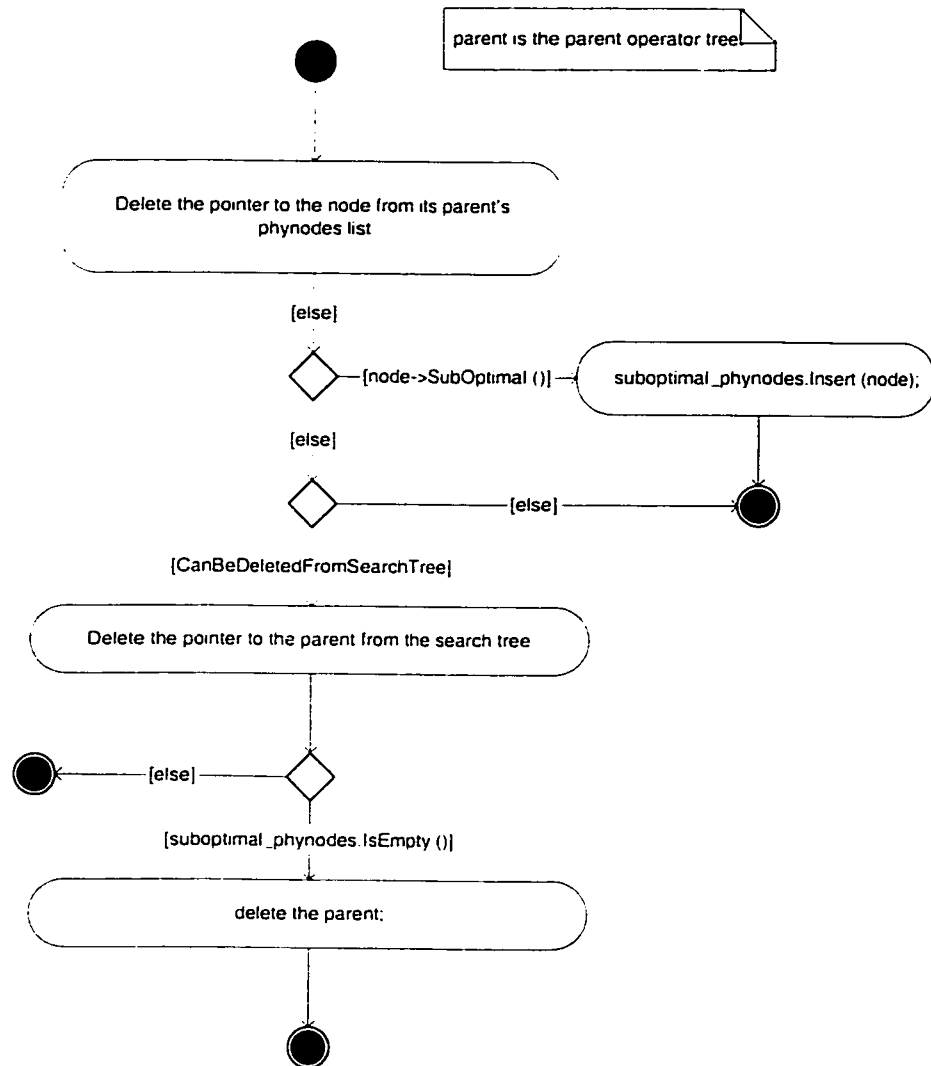

Figure 3.14 The Activity Diagram for Deleting an Operator Tree

Note that:

1. An algorithm tree can be used by more then one tree, so it has an attribute to record how many other nodes are referencing it. When an algorithm tree is used, we cannot delete it; instead, we set it to be sub-optimal and do not expand it further. For example: consider a node (n1), which has been used as an input node by another node (n2). Now, if a third node (n3) is created and it happens to be equivalent to node n1 and also happens to be cheaper than n1, then we would

delete n1. But if, in this situation, it happens that miraculously n2 turns out to be part of the ultimate optimal plan (unlikely, but possible in the current scheme) then we are in trouble, because n2 will try to access n1, but n1 has already been deleted.

2. The object of class **SearchTree** holds an array of lists containing pointers to operator trees, which may be expanded further. When a logical tree is no longer needed, we delete its pointer from the lists.

In this chapter, we described OPT++, the framework we use to build our own optimizer. We describe the framework based on our understanding gained from experience. Only the attributes and methods that help understand the framework are introduced. Complete description of OPT++ is available in the Kabra' paper [Kab99] and Jinmiao's thesis [Li01].

Chapter 4 The PostgreSQL Optimizer

This chapter will analyze the optimizer of PostgreSQL. After an overview, the main data structures of the optimizer are introduced. The transformative rules used in the optimizer are extracted and described. The rules are applied regardless of system statistics. The search strategies used in the optimizer are illustrated. Especially, dynamic programming and the genetic algorithm are described in detail.

4.1 Overview

The object-relational database management system now known as PostgreSQL (and briefly called Postgres95) is derived from the POSTGRES package written at the University of California at Berkeley [Pos02a]. Our target optimizer is for a relational database, so we only analyze its features for optimizing relational databases.

As a mature relational-object database, PostgreSQL is compatible with the SQL92/SQL99 standard. Although the optimizer does not generate logical trees while optimizing a query, we still can extract the logical operators and their corresponding physical operators abstractly on the basis of the SQL standard it supports. Table 4.1 shows the logical operators and physical operators implementing the logical operators. The relational-object database specific physical operators, such as materialization, append, and operators for non-SELECT queries, are not listed here.

Table 4.1 Logical and Physical Operators Supported by PostgreSQL

Logical Operator	Corresponding Physical Operators
RELATION	SEQSCAN, INDEXSCAN
UNION	UNION
INTERSECTION	INTERSECTION
EXCEPTION	EXCEPTION
SELECTION	SELECTION
PROJECTION	*
PRODUCT	NESTED LOOP
JOIN	NESTED LOOP, HASHJOIN, MERGEJOIN
DUPLICATE ELIMINATION	UNIQUE
GROUPBY	AGG
ORDERBY	SORT
SUBQUERY	SUBQUERYSCAN

Note: *There is no explicit projection operator in PostgreSQL. After each physical operator is executed, the result is calculated according to the target list. The calculation is actually a projection.*

All the logical and physical operators except the ones for set operation were implemented in our target optimizer.

The search strategy varies with the number of relations involved in a query. If the number of relations in a query does exceed a default value, the optimizer will do a near-exhaustive search through the join tree space; otherwise, the genetic algorithm will be employed to perform a semi-random search.

4.2 Main Data Structures of the PostgreSQL Optimizer

To understand how the optimizer works, it is necessary to study its main data structures. As with other parts of PostgreSQL, the optimizer was written in the C language. Therefore, the main data structures are *structs*. Here, the two data structures, Query and Plan are introduced. The *struct* Query represents a query internally, while the *struct* Plan stands for a physical plan generated for a query. Because there are no logical operator trees generated during the course of optimization, the two data structures are the most important ones in regard to understanding the optimizer.

4.2.1 Internal Representation of a Query

A query, after being parsed and rewritten, is represented with a query tree (a *struct* in C language) internally. The data structure contains all necessary information, which can be used conveniently in subsequent processes. In PostgreSQL, a query tree is composed of the following parts:

- The command type

  This is a simple value of *enum* type telling which kind of command (SELECT, INSERT, UPDATE or DELETE) produced the query tree. Because of the scope of our project, we focus only on the SELECT command.

- The range table

  The range table is a list of relations that are used in the query. In a SELECT statement these are the relations given after the FROM keyword.

Every range table entry stands for a table or view and tells by which name it is called in the other parts of the query. In the query tree, the range table entries are referenced by index rather than by name, so here it does not matter if there are duplicate names.

- The result relation

  This is an index (*integer*) into the range table that identifies the relation where the results of the query go. This is for SELECT INTO, INSERT, UPDATE and DELETE commands.

- The target list

  The target list is a list of expressions that define the result of the query. In the case of a SELECT, the expressions are what builds the final output of the query. They are the expressions between the SELECT and the FROM keywords. The symbol * is just an abbreviation for all the attribute names of a relation. It is expanded by the parser into the individual attributes, so the optimizer will never see it.

  Every entry in the target list contains an expression that can be a constant value, a variable pointing to an attribute of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators etc.

- The qualification

  The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean value that tells whether the operation (INSERT, UPDATE, DELETE or SELECT) for the final result row should be executed or not. It is the WHERE clause of an SQL statement.

  For a specific relation, a qualification may be a Boolean expression restricting the output of the relation, or a join qualification implying a join with another. For example, `table1.age>25` is a restriction qualification, while `table1.name=table2.name` is a join qualification implying a join between table1 and table2.

- The join tree

The query's join tree is a tree showing the structure of the FROM clause. For a simple query like `SELECT FROM a, b, c`, the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions—particularly outer joins— are used, we have to join in the order shown by the joins. The join tree shows the structure of the JOIN expressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join tree nodes. The key words explicitly indicate how the relations will be joined. INNER/OUTER, NATURAL, LEFT/RIGHT/FULL are called join directives.

- The others

  There are the other parts of the query tree representing the group clause, having qualification, DISTINCT clause, SORT clause and set operations. Also, there are some lists for the subsequent use of optimization.

Figure 4.1 is a sample query tree, in which target list, join tree, and qualification are shown. The range table for the query should be {table1, table2}.

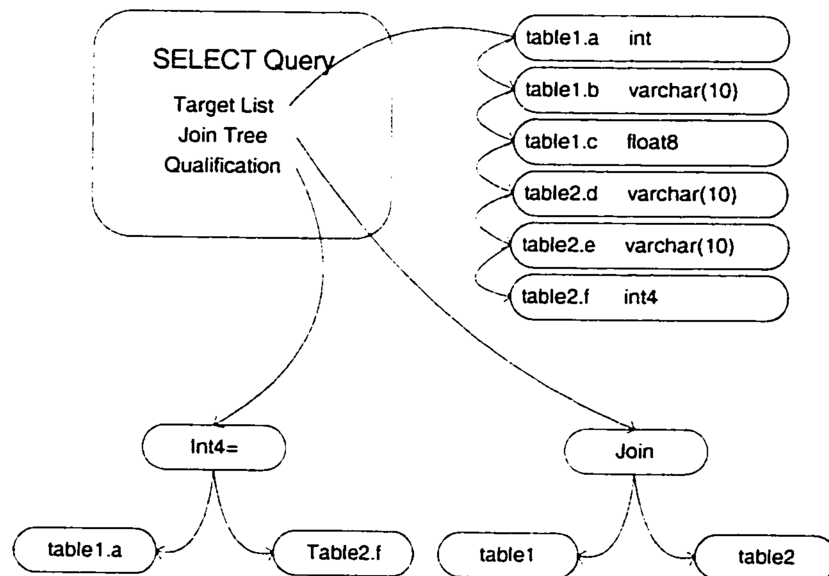

Figure 4.1 A Query Tree

4.2.2 Internal Representation of a Plan

After optimization, a physical plan to answer the query is generated. A physical plan is a tree. Each node of such a tree is a physical operator (e.g. algorithm). The major members of the *struct* plan are:

- type : Type: *enum*. It is a variable indicating the operator of current node. For example, it may be HASHJOIN.
- cost: Type: *double*. It is the estimated cost consumed after the current operator is executed.
- plan_rows: Type: *double*. It is the number of rows the plan is expected to emit.
- plan_width: Type: *int*. It is the average row width in bytes
- target_list: Type: *List*. It is a list of expressions.
- Qualications: Type: *List*. It is a list of implicitly-ANDed qualication conditions
- left_tree: Type: *struct* Plan *. It is a pointer to it left input which is also a node of a plan tree.
- right_tree: Type: *struct* Plan *. It is a pointer to it rigth input which is also a node of a plan tree.
- The other members indicate run time information and other plans which must be executed before the current node is executed.

Figure 4.2 is an example physical plan. The leaf nodes are scans of relations. Because there is an index on table1.a3, and also there is a restriction qualification, a3=42, on it, we can access table1 by index scan. For table2, a sequential scan has to be used. The qualification, a2=b2, implies a join between table1 and table2. The directive INSTINCT indicates that a physical operator, Unique, has to be applied, while the physical operator Unique requires that its input must be sorted, so the physical operator, Sort, is applied first.

Query: Select DISTINCT a1,b1 from table1, table2
WHERE table1.a2=table2.b2 AND table1.a3=42;

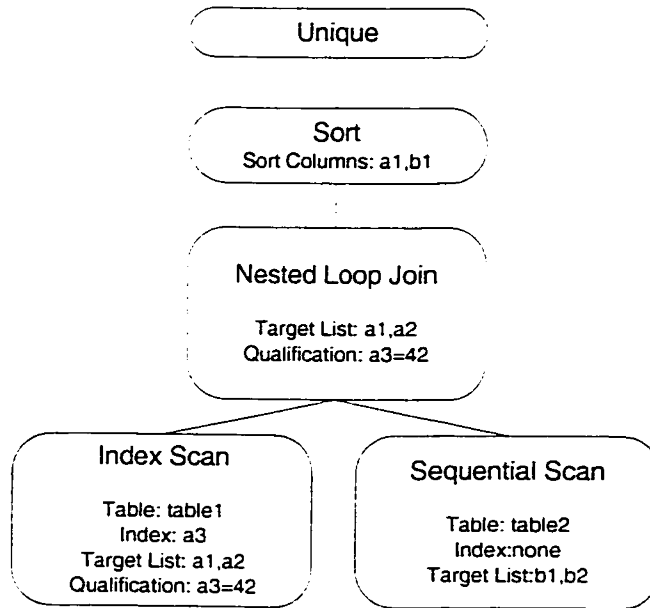

Figure 4.2 A Plan

4.3 Transformative Rules Used in the PostgreSQL Optimizer

While optimizing a query, some transformative rules can be applied without having to know the system statistics. These rules proved to be helpful for generating better physical plans. In this section, the transformative rules used in the optimizer will be listed, and their corresponding consequences will be discussed. Because there is no literature on the transformative rules of the PostgreSQL optimizer, we extracted the following rules from the source code and comments. The version we studied was PostgreSQL7.2.1 [Pos02b].

Rule1: or-to-union

- **Description**: A qualified query with or-operator in its where-clause is converted to union of two queries.
- **Example**:

  A query:

```
SELECT a,b, ... FROM one_table WHERE
(v1 = const1 AND v2 = const2 [ vn = constn ]) OR
```

```
(v1 = const3 AND v2 = const4 [ vn = constn ]) OR
...
[(v1 = constn AND v2 = constn [ vn = constn ])]
```

is converted into:

```
SELECT a,b, ... FROM one_table WHERE
(v1 = const1 AND v2 = const2 [ vn = constn ]) UNION
SELECT a,b, ... FROM one_table WHERE
(v1 = const3 AND v2 = const4 [ vn = constn ]) UNION
...
SELECT a,b, ... FROM one_table WHERE
[(v1 = constn AND v2 = constn [ vn = constn ])]
```

- **Applicability**: To qualify for transformation, the query must be SELECT command without any sub-SELECTs. It must not have a HAVING clause, or a GROUP BY clause. Moreover, it must be a single table.
- **Consequences**: The benefits of the transformation are:
  1. Avoiding the exponential memory consumption of applying Rule 2.
  2. Making it possible to use index access methods. If a qualification resides in an OR-expression, we cannot push it to its corresponding relation, therefore, we cannot make use of index access methods.

Rule 2: constant-expression-simplification

- **Description**: Reduce any recognizably constant sub-expressions of the given expression tree. Simplify Boolean expressions containing constant sub-expressions if possible.
- **Example**: The expression "2 + 2" will be converted into "4". The expression "X or true" will be converted into "true".
- **Applicability**: Any constant sub-expressions occurring in expressions.
- **Consequences**: The constant sub-expressions are simplified or even removed; therefore, in some cases, select-push-down and other operations become possible or at least more convenient after the transformation. For example, after we reduce an expressions: "x OR true" to "true", we do not do any process on the expression.

Rule 3: Expressions-normalization

- **Description**: Convert a qualification to the most useful normalized form, either CNF (AND-of-ORs) or DNF(OR-of-ANDs). Push down NOTs.
- **Example**: The expression "A OR (B AND C) " will be normalized to be "(A OR B) AND (A OR C)". The expression "NOT (A OR B)" will be converted into "(NOT A) AND (NOT B)".
- **Applicability**: All expressions in WHERE-clauses and Join qualifications will be normalized. However, normalization is only carried out in the top AND/OR/NOT portion of the given tree; we do not attempt to normalize Boolean expressions that may appear as arguments of operators or functions in the tree.
- **Consequences**: Query qualifications (WHERE clauses) are ordinarily transformed into CNF, i.e. AND-of-ORs form, because then the optimizer can use any one of the independent AND clauses as a filtering qualification. However, qualifications that are naturally expressed as OR-of-ANDs can suffer an exponential growth in size in this transformation, so we also consider converting it into DNF (OR-of-ANDs), and we may also leave well enough alone if both transformations cause unreasonable growth. The OR-of-ANDs format is useful for index-scan implementation, so we prefer that format when there is just one relation involved.

Rule 4: select-push-down

- **Description**: Push selections down to their corresponding relations as low as possible. In PostgreSQL, if there is no index on a required selection attribute, the selections are applied right after a sequential scan; otherwise, an index-scan is adopted to retrieve the tuples.
- **Applicability**: Restriction qualifications in a where-clause are selection qualifications to be pushed down.

- **Consequences**: As illustrated in almost every database textbook, by select-push-down, undesired tuples are filtered out as early as possible, so normally, costs for subsequent operations decline drastically.

Rule 5: Sub-query-pull-up

- **Description**: When a query contains a simple sub-query in its FROM-clause, the sub-query will be pulled up and merged into the upper query. Sub-query-pull-up must be done recursively. That means the sub-queries' sub-queries should be pulled up first if they can be.
- **Example**: A query: `SELECT * FROM table1, (SELECT * FROM table2 WHERE ...) AS foo WHERE...` will be transformed to be: `SELECT * FROM table1, table2 WHERE ......`.
- **Applicability**: The sub-query to be pulled up must be a simple SELECT query in range table. When a query is not a set operation, and does not have aggregation, grouping, having, distinct, or sorting, it is viewed as a simple query by the PostgreSQL optimizer.

  We do not pull up a sub-query that has any set-returning functions in its target list, otherwise we might wind up inserting set-returning functions into places where they must not go, such as qualifications of higher queries. Also, we do pull up sub-queries, if there are too many sub-queries that can be pulled up, because that will cause unpleasant growth of optimization time.
- **Consequences**: The optimizer will find more possible ways to join the relations. One of the newly added ways may be the best. Sub-queries force particular join path methods and orders for the query. After pulling up sub-queries, the optimizer can choose join order more flexibly; therefore, undesired tuples can be filtered out as early as possible [Pir92]. For example, in the query: `SELECT * FROM table1, (SELECT * FROM table2, table3 WHERE ...) as foo WHERE...`, if the sub-query is not pulled up, table2 and table3 must be joined first, and then a sub-plan is generated. After pulling up the sub-query, the optimizer can join the tree relations in any order it sees proper.

However, that could result in unpleasant growth of optimization time, since the dynamic-programming search has runtime exponential in the number of FROM-items considered. Therefore, we do not merge FROM-lists if the result would have too many FROM-items in one list.

It must be kept in mind that a sub-query in the range table is a part of the join tree, so it must be guaranteed that the join-tree structure is preserved properly after a pull-up.

Some less important rules used in the optimizer are:

- A HAVING clause without aggregation is equivalent to a WHERE clause. Normally a master user will write such a query. But if it happens, the rule will help to do select-push-down, because the qualification can be a restriction expression to a relation.
- Projection-push-down. The PostgreSQL optimizer does not have an algorithm for projection, but each node of a physical plan has a target list, so projections are done as early as possible.

The first three rules are preparation rules, they do not help generate good plans directly, but they make subsequent manipulations more convenient. For example, select-push-down and calculation of selectivity will be done more easily after transformation. The rule select-push-down is the most typical optimization transformation rule, which affects the execution cost a lot. The rule sub-query-pull-up has special importance in a DBMS supporting views, because views will be converted into sub-queries. The rule helps to find more ways to join relations.

The above rules must be applied in a proper order. The PostgreSQL optimizer applies the rules in such an order: or-to-join, constant-expression-simplification, expression-normalization, sub-query-pull-up, and select-push-down.

4.4 Search Strategies Used in the PostgreSQL Optimizer

4.4.1 Overall Search Strategy

Among all the relational operators, the most difficult one to process and optimize is the join. The number of alternative plans to answer a query grows exponentially with the number of joins included in it. Further optimization effort is caused by the support of a

variety of join methods (e.g., nested loop, hash join, merge join in PostgreSQL) to process individual joins and a diversity of indexes (e.g., R-tree, B-tree, hash in PostgreSQL) as access paths for relations.

The current PostgreSQL optimizer performs a near-exhaustive search over the plan space. This query optimization technique is inadequate to support database application domains that involve the need for extensive queries, such as artificial intelligence. Performance difficulties in exploring the space of possible query plans created the demand for a new optimization technique being applied. To handle large join queries, non-exhaustive search is adopted.

Because join is the most complex for optimizing among all operators, the PostgreSQL optimizer focuses principally on optimizing join, where other operators are converted into nodes of a physical plan in a relatively fixed way. Figure 4.3 shows the high-level activities of the optimization.

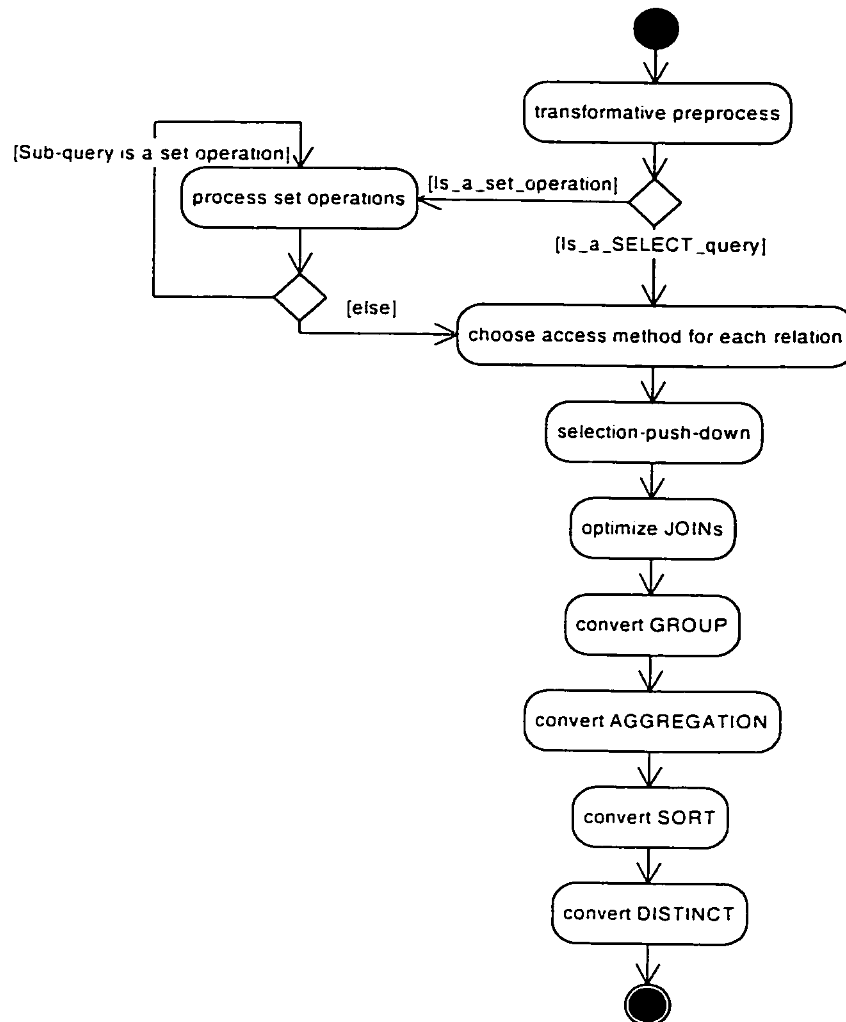

Figure 4.3 The Activity Diagram for PostgreSQL Optimization

During the optimizing process, we build algorithm trees, which are called "paths" in PostgreSQL documentation, representing the different ways of doing a query. Note that in a path, only relation access methods and joins are determined. We select the cheapest path that generates the desired relation and turn it into a Plan to pass to the executor.

First, the optimizer chooses a proper access path for each base relation in the query. Base relations are either primitive tables, or sub-queries that are planned via a separate recursive invocation of the optimizer. Possible paths for a primitive table relation include sequential scan and index scans for any indexes that exist on the table. A sub-query base

relation just has one path, a "SubqueryScan" path, which links to the sub-plan that was built by a recursive invocation of the optimizer. Then joins are optimized. A join takes two inputs that can either be a base relation, or a join. All plausible physical methods and orders are considered. It should be kept in mind that although the different join paths for a query will generate the same result, but the costs will vary a lot because of the system statistics. That is why most effort of the optimization is spent on joins. For a query, if the number of relations in its range table does not exceed a certain value (11 by default), we can afford to do exhaustive search in the join space, so constrained dynamic programming is used. Otherwise, a randomized search strategy must be adopted, because the cost will increase exponentially with the number of joins. Finally, the operators GROUP, AGGREGATION, ORDER, and DISTINCT are converted into the corresponding physical nodes in a fixed way.

4.4.2 The Constrained Dynamic Programming Search Strategy

4.4.2.1 Dynamic Programming Algorithm

Dynamic programming is essentially a table-filling approach for combinatorial optimization problems. When there are n items in the problem, we consider the situations of combination of 1 item, 2 items, ...n items in turn. The solution of a level (except the lowest level) depends on its lower levels. Finally, the optimal solution is found.

As Table 4.2 shows, for a query containing n base relations, in level i, we have c_n^i of possibilities of ways to choose i relations without considering of the join order and physical methods.

Considering permutation of join order, the number of ways (paths) will increase exponentially with the number of relations.

The total number of all tree shapes T(n) for n relations can be given by the recurrence[Gar00]:

$$T(1) = 1$$

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

Table 4.2 The Possible Combinations

Level	Number of Possibilities	Combinations
n	c_n^n	{table1, table2, table3,...tableN}
...	...	......
2	c_n^2	{table1, table2},{table1, table3}...{ table1, tableN}...
1	c_n^1	{table1}, {table2}, {table3}...{tableN}

In dynamic programming, we have to calculate the cheapest costs of combinations in level 1, and then calculate the cost of combinations in upper levels in turn based on the results of lower levels. There are three points we should notice:

1. Because all possibilities are considered, dynamic programming guarantees that the optimal plan must be found.
2. We cannot discard the results of lower levels until all computations are finished, because we need them to make upper level decisions.

   For example, to calculate the cheapest cost of the combination {table1, table2, table3}, the cheapest costs of {table1},{table2},{table3},{table1, table2},{table1, table3},{table2, table3} must be known first. The costs are available by looking up in lower levels.

   When we get the cheapest cost, we also get a path to achieve the result. For instance, among all possible paths to join table1, table2 and table3, we may find the cheapest path is {{table1, table3}, table2}. To be used by higher level calculations, the path must be preserved, until the final optimum is found.
3. The calculation effort will rise exponentially with the number of base relations.

4.4.2.2 The Constrained Dynamic Programming in the PostgreSQL optimizer

The optimizer does use dynamic programming strategy to search for the best physical plan when the number of base relations is smaller than a certain value. However, it improves the strategy by ignoring the combinations that will not be used by the best plan.

An item in a FROM-clause can be a base relation, an explicit join indicated a join directive, or a sub-query. For an explicit join, the join order is fixed. Only physical method and inner/outer position are determined by calculation. The constrained dynamic programming takes an item in a FROM-clause as a basic block.

Study indicates that left-sided and right-sided join trees tend to produce the best plan [Gar00]. Therefore, left-sided and right-sided trees are considered first. Then a subset of bushy trees is considered. Figure 4.4 is the activity diagram for the constrained dynamic programming strategy, where there are n items in the FROM-clause.

The strategy constrains the search space in two steps:

- Generating left-sided/right-sided trees; and
- Generating bushy trees.

A. Generating left-sided/right-sided trees.

For each tree in level i, in order to generate trees in level i+1, only the items in the FROM-clause (except itself and the relations it covers) that have a suitable join clause with it are considered to be candidates to join. If the current tree is not joined to any other item in FROM-clause, we must do a Cartesian product with each item in the FROM-clause except itself. In this way, if an item can be joined to any other items, it does not need to do a Cartesian product with the other items. Typically, calculation will be considerably reduced. The reason why it is safe to do this rests in that a Cartesian product will produce a larger output than a join generally, so the trees based on Cartesian product must be more expensive.

For example, there is a query,

```
SELECT * FROM t1,t2,t3,t4,t5 WHERE t1.c1=t2.c2 AND t3.c3=t4.c4 AND
t2.c5=t3.c5;
```

The WHERE-clause implies three joins: {t1,t2}, {t3,t4} and {t2,t3}

Using the constrained dynamic programming, in level 1, we get the following trees: {t1},{t2},{t3},{t4},{t5}.

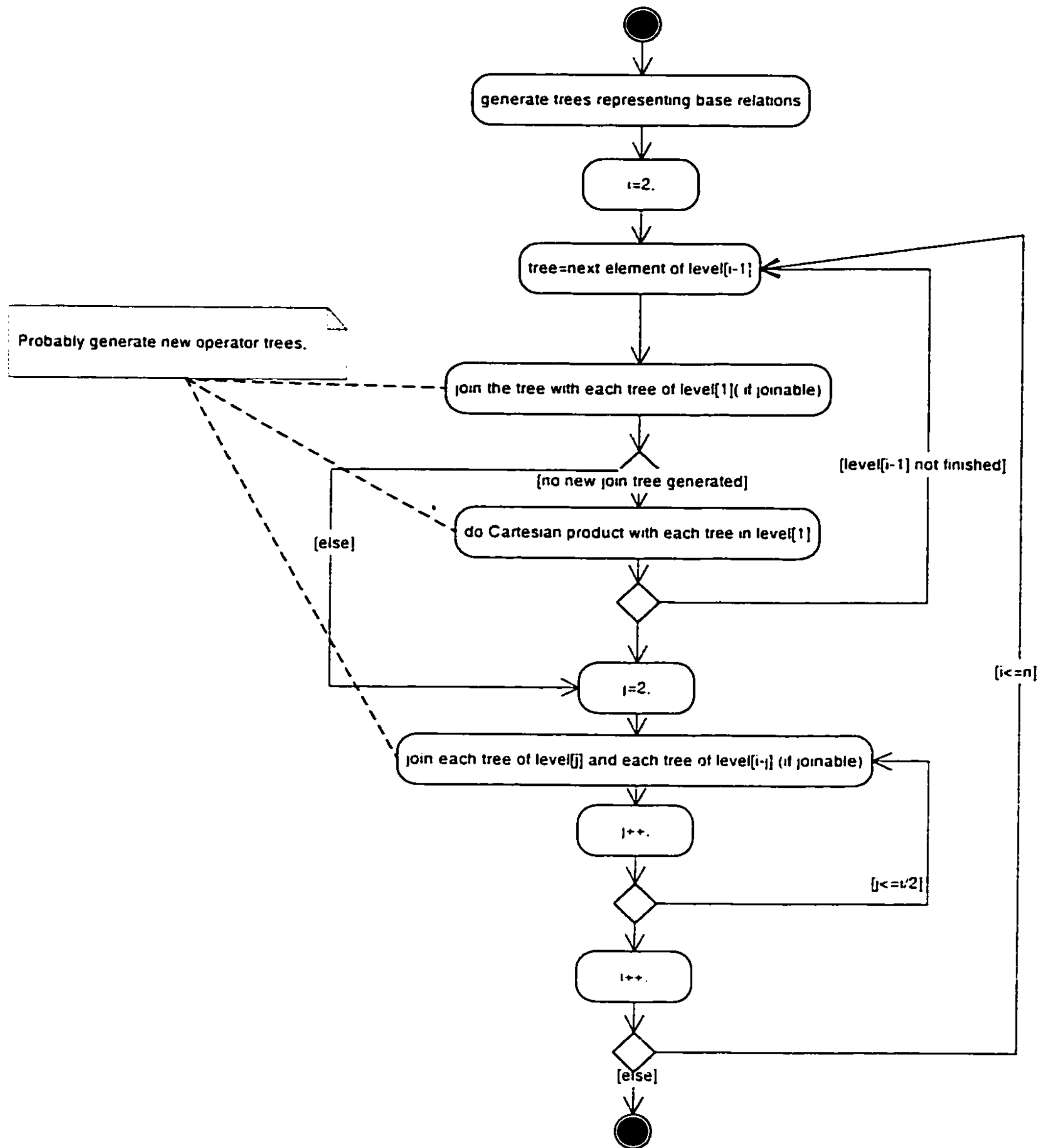

Figure 4.4 Constrained Dynamic Programming

Now, we calculate trees in level 2. There is a join clause between t1 and t2, so the tree {t1,t2} should be generated. So are {t3,t4} and {t2,t3}. There is no item that has join clause to t5, so t5 has to do a Cartesian product with each of the rest. The four trees, {t5,t1},{t5,t2},{t5,t3},{t5,t4}, are generated. So there are 6 trees in level 2.

Similarly, we can get the trees in level 3,

{{t1,t2},t3},{{t3,t4},t2},{{t5,t1},t2},{{t5,t2},t1},{{t5,t2},t3},{{t5,t3},t4},{{t5,t3},t2},{{t5,t4},t3}.

The join trees containing the same relations, for instance {{t5,t3},t4},{{t5,t4},t3,} compete for the best plan, the ones with more expensive costs are removed. So finally, the surviving trees are:

{{t1,t2},t3},{{t3,t4},t2},{{t5,t1},t2},{{t5,t2},t3},{{t5,t3},t4}.

The final winners in level 4 are:

{{{t1,t2},t3},t4},{{{t5,t1},t2},t3},{{{t1,t2},t5},t4},{{{t5,t3},t4},t2}.

The best tree in level 5 can be (depending on the system statistics):

{{{{t1,t2},t3},t4},t5}.

B. Generating bushy trees

Only a pair of join trees with a suitable join clause can be considered to generate a new bushy join tree.

Take the previous query as an example again. In level 1, 2 and 3, there are no bushy trees generated. In level 4, a bushy trees, {{1,2}{3,4}} is generated, because there is a join clause between {t1,t2} and {t3,t4}. The bushy tree will compete for the equivalent trees (the trees containing the same relation as it).

Likewise, in level 5, bushy trees are constructed. They are: {{{t1,t2},t3},{t5,t4}},{{{t3,t4},t2},{t5,t1}},{{{t5,t1},t2},{t3,t4}},{{{t5,t3},t4},{t1,t2}}. As mentioned before, the bushy trees have to compete with their equivalents, only the ones with cheapest costs will be preserved.

We notice that although some combinations are ignored, there is a guard condition in each level. That is: each item in the FROM-clause must appear at least once in each level. The guard condition guarantees that a complete tree that covers all relation can be constructed at the top level.

It is worthwhile to point out that the inner/outer positions and physical methods within a combination must be determined by calculation based on system statistics. For example, we have a combination,{{table1, table3},table2}, we have to determine that which one acts as inner input, {table1, table3},or table2. Also, we have to determine which physical method is adopted, hash-join, nested-loop, or merge-join.

4.4.3 The Genetic Algorithm Query Optimization in PostgreSQL

4.4.3.1 Genetic Algorithm

The genetic algorithm (GA) is a heuristic optimization method that operates through randomized search within a fixed time bound.

First we illustrate some terminologies used in next discussion.

- Individual: An individual represents a solution of the problem.
- Population: The fix-sized sub-set of all possible solutions for the combinatorial optimization problem is considered as a population of individuals.
- Fitness: The degree of adaptation of an individual to its environment is specified by its fitness.
- Chromosome: The coordinates of an individual in the search space are represented by chromosomes, which are in essence a set of character strings.
- Gene: A gene is a subsection of a chromosome that encodes the value of a single parameter being optimized. Typical encodings for a gene could be binary or integer.

Through simulation of the evolutionary operations (recombination, mutation, and selection), new generations of search points are found that show a higher average fitness than their ancestors. Figure 4.5 shows the activity diagram of the genetic algorithm, in which P(t) stands for the generation of ancestors at a time t, while P"(t) represents the generation of descendants at a time t.

It should be pointed out that the genetic algorithm does not guarantee that the optimal solution will be found, but it reaches a fairly good solution in a fixed time. When we cannot afford to search all the space, it is an ideal choice.

According to the PostgreSQL documentation [Pos02a], it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random), because some kind of heuristic information is used.

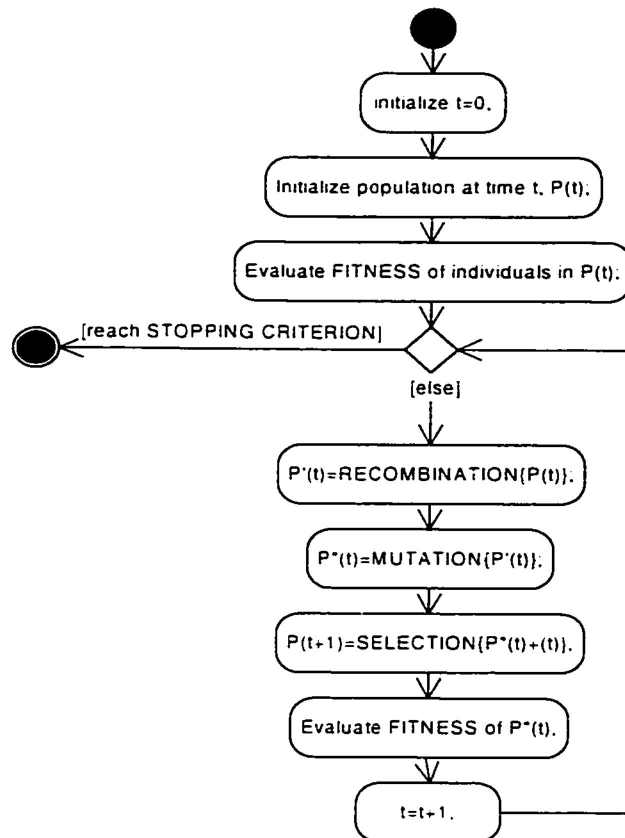

Figure 4.5 The Activity Diagram for Genetic Algorithm

4.4.3.2 Genetic Query Optimization in the PostgreSQL Optimizer

A solution of a query optimization problem can be viewed as a possible route of a traveling salesman problem (TSP)[Mic96]. Possible query plans are encoded as integer strings. Each string represents the join order from one relation of the query to the next, just like a string of city numerals. For example, the query tree represented by Figure 4.6 is encoded by the integer string '1-4-3-2', which means, first join relation '1 and '4, then '3', and then '2', where 1, 2, 3, 4 are relation IDs within the PostgreSQL optimizer. To reduce search time, only left-sided trees are considered, so a sole join tree can be constructed from a chromosome.

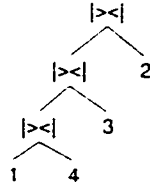

Figure 4.6 A Join Tree

The Genetic Query Optimization (GEQO) strategy is to search for a better solution over the space of such routes in a limited time. Figure 4.7 shows the activity diagram of the GEQO strategy.

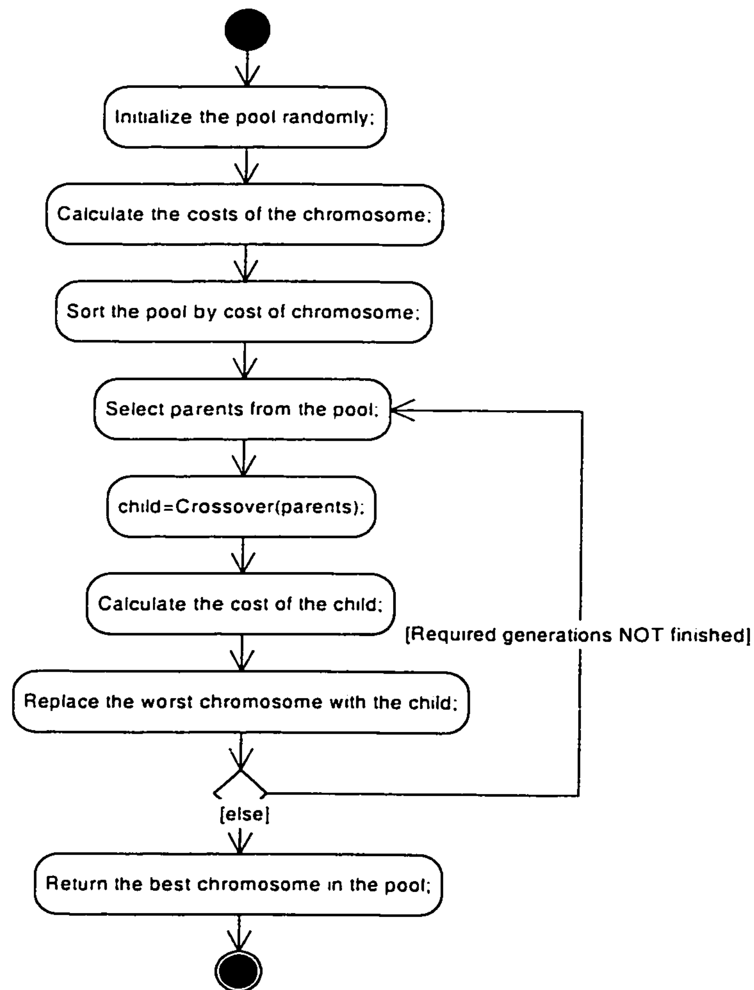

Figure 4.7 The Activity Diagram for GEQO

The population of individuals is put into a pool, which is actually an array of chromosomes. At the beginning, the pool size and number of the needed generations are calculated based on the number of relations involved in a query, and then the chromosomes in the pool are initialized randomly.

In each iteration, parents are selected from the pool based on a given linear bias. A child is generated by crossover of the parents. During crossover, some kind of heuristic information is used. Namely, priority is given to the "shared" edges. In the TSP problem, the shared edges refer to the edges between two cities, which are shared by more than one route.

There are many kinds of ways to do crossover. Five variants, edge recombination crossover, partially matched crossover, cycle crossover, position crossover, and order crossover, are supplied in the PostgreSQL optimizer.

At the end of each iteration, the child will replace the individual with the most expensive cost in the pool. After all generations are finished, the best chromosome in the pool is taken to construct a join tree.

Parts of the GEQO module are adapted from D. Whitley's Genitor algorithm. Specific characteristics of the GEQO implementation in PostgreSQL are [Pos02a]:

- Use of a steady state GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;
- Use of edge recombination crossover which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;
- Mutation as a genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

The GEQO module allows the PostgreSQL query optimizer to support large join queries effectively through non-exhaustive search.

In this chapter, a model optimizer, the PostgreSQL optimizer, is analyzed. Its operators and main data structures are introduced, and its transformative rules are

extracted. Also, its search strategies are illustrated. In the next chapter, we will describe how to implement a PostgreSQL-like optimizer within the OPT++ framework.

Chapter 5 The Implementation in OPT++

In this chapter, the framework-based implementation of a relational optimizer will be described. The internal representation of a query will be introduced. Customization of the three components in OPT++: Algebra, Search Space and Search Strategy, will be presented. Problems encountered in the implementation and corresponding considerations will be discussed.

5.1 Overview

In our project, a PostgreSQL-like optimizer for a relational database was implemented based on OPT++. By PostgreSQL-like, three things are implied: (1) most of logical and physical operators supported by PostgreSQL are also supported by our optimizer; (2) most of its transformative rules are applied in our optimizer; and (3) we use the same search strategies as PostgreSQL.

All operators in Table 4.1, except the set operators, are implemented in our optimizer. However, considering the need of the target database, we focused on SELECT queries, while INSERT, UPDATE, and DELETE queries were not implemented. To be compatible with standard SQL, we modified the parser equipped with OPT++, because it does not support some SQL grammar features, such as sub-query and explicit join.

As with PostgreSQL, we focus mainly on optimization of joins. Two search strategies, constrained dynamic programming and genetic algorithm, are implemented in our optimizer. When there are not too many relations in the range table of a query (11 by default), a near-exhaustive search is employed to find the best plan; otherwise, a semi-random search is performed to try to find as good a plan as possible in a limited time. Other operators are converted into corresponding execution algorithms in a fairly fixed way.

5.2 Internal Representation of a Query

Given a query, a parser parses it into a parse tree, and performs type-checking on the parse tree. Then, it is converted to another tree representing the query, called a query tree. The optimizer will take a query tree as input, and produce an optimal physical plan. The

representation of a query tree affects the implementation of an optimizer heavily, so it is necessary to introduce it before moving to optimization implementation.

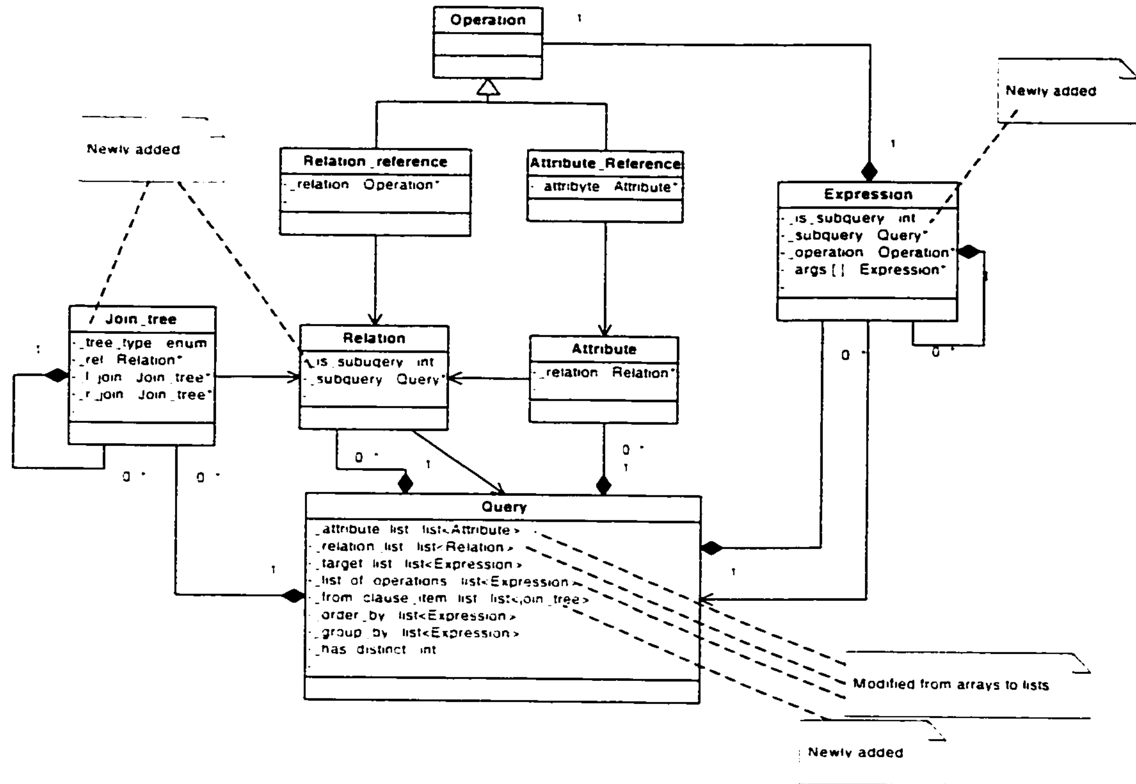

Figure 5.1 The Class Diagram for a Query Tree.

Figure 5.1 is the class diagram for a query tree. Here, the main classes, their members and relationship amongst them are introduced. For clarity, only data members critical for understanding are given. For the same reason, some classes were renamed.

Class Query

Description: The class **Query** represents a SQL query internally.

Data members:

- **`_relation_list`**—a list of pointers to objects of class **Relation**, which represent relations involved in a query, namely, a range table.

- `_attribute_list`—a list of pointers to objects of class **Attribute**, which represent all attributes involved in a query.
- `_target_list`—a list of pointers to objects of class **Expression**, which represents items in a target list. Each item can have an alias.
- `_from_clause_item_list`—a list of pointers to items in the FROM-clause. An item is a join tree represented by an object of class **Join_tree** that points to a relation or a sub-query represented by an object of class **Relation**,.
- `_group_by`—a list of pointers to objects of class **Expression**, which represent the GROUP BY items.
- `_order_by`—a list of pointers to objects of class **Expression**, which represent the ORDER BY items.
- `_has_distinct`—a member of *int* type, indicating whether the query has a DISTINCT operation.

Class Relation

Description: The class **Relation** represents relations involved in a query. A relation can be a real relation, or a sub-query mentioned in the FROM-clause. For a real relation, its object contains all necessary information for optimizer, for example, internal id, number of tuples, size of a tuple etc. Note that a sub-query in the FROM-clause must have an alias so that it can be referenced to.

Class Attribute

Description: The class **Attribute** represents attributes of relations involved in a query. An object contains all necessary attribute information for optimization, e.g., internal id, name, type, size, index etc.

Class Expression

Description: The class **Expression** represents an expression occurring in a query. An expression normally is composed of an operation, and some arguments, which are also expressions. The SQL grammar allows a sub-query to exist in an expression, so it has a pointer to class **Query** to preserve the reference to a sub-query.

Class Operation

Description: The class **Operation** represents operators in expressions. An operation can be an operator, which takes some arguments, for example, arithmetic operator "+". Also, it can be a dummy operator, such as a constant, an attribute reference or a relation reference. It has many subclasses including **Relation_reference** and **Attribute_reference**. The other sub-classes representing different kinds of operators are not shown here. In this way, an expression is constructed recursively.

Class Relation_reference and Attribute_reference

Description: The two classes keep pointers to objects of class **Relation** and **Attribute** respectively so that they can be referenced to. They are defined as sub-classes of class **Operation** to indicate that they are dummy operators.

Class Join_tree:

Description: The class **Join_tree** represents the structure of an item in FROM-clause. An item can be a relation or an explicit join designated by some join directives. The class join tree represents both the two cases by one of its members, _tree_type.

Data members:

- `_rel`—When an object of the class represents a relation, it is a pointer to an object of class **Relation**; otherwise, it should be NULL.
- `_l_join`, `_r_join`— When an object of the class represents a join, they are pointers to objects of class **Join_tree** respectively, acting as input of the current join; otherwise, they should be NULL.

As mentioned above, the OPT++ parser was modified in our project, so the structure of a query tree was modified correspondingly. OPT++ provides the main design of the classes and their relationships. In order that sub-queries and explicit joins can be fitted in a query tree, we made the following modifications to the structure:

- Adding class **Join_tree** to represent explicit join structures;
- Adding `_from_clause_item_list` in class **Query** to fit the representation of sub-queries and explicit joins;
- Adding a reference to class **Query** in class **Relation** so that a base relation can be a sub-query;

- Adding a reference to class **Query** in class **Expression** so that a sub-query can appear in a WHERE-clause;

5.3 Application of Transformative Rules

There are two reasons why transformative rules should be applied. One is that some transformations, for example select-push-down, can help generate a better plan without knowing system statistics. Hence the subsequent execution costs decrease drastically. The other reason is that some transformations, such as making the expressions to be implicit-ANDed , will make following operations more convenient.

Referencing to transformative rules in the PostgreSQL optimizer, the following rules are implemented in our optimizer.

- Expression-normalization
- Select-push-down
- Sub-query-pull-up

While the other two rules, or-to-union and constant-expression-simplification, which are relatively less critical, are not applied.

The implementation of the three rules falls into three cases:

- The framework has implemented it (Expression-normalization);
- The framework left it as a hot spot (Select-push-down);
- Neither did the framework implement the rule, nor did it leave any hot spot for the rule (Sub-query-pull-up)

5.3.1 Expression-normalization

From the standpoint a framework user, the first case is the easiest one. We need not to do anything. When a parse tree is converted into a query tree (internal representation), the expressions are dissected into OR-expressions concatenated by ANDs. The dissected expressions are stored in a list, and the relationship between any two of them is implicitly AND.

5.3.2 Select-push-down

In the second case, we have to do some work, but it is relatively easy, because we just need to implement some hook methods. Select-push-down is implemented during optimization, rather than before optimization as expression normalization. As we learned in chapter 3, OPT++ allows us to change our search strategy by sub-classing class **SearchTree** and **SearchStrategy** and overriding hook methods, *MakeInitialTree* and *DoSearch*. The default search strategy supplied by OPT++, Bottom-up strategy, performs select-push-down by pushing a selection down along a logical tree generated so far whenever possible. However, the strategy is based on an assumption: the strategy component does not know what operators are applied, or in what order the operators are applied. When a sub-tree, which may be part of the final tree, is generated, the strategy tries to all operators in the actual algebra on it to see whether a new node can be added, and therefore the tree can be expanded. Figure 5.2 shows an example of select-push-down. It is hard to understand in code level, and also less efficient, because the optimizer tries to apply all operators repeatedly, even though most of them are not applicable to the current sub-tree.

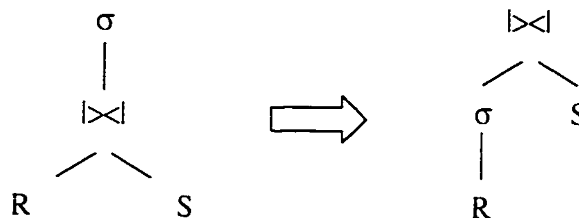

Figure 5.2 Select-push-down

In our strategy, we implement it by trying to apply different operators in a fixed order. We believe they conform to the nature of SQL queries. No matter what a query is, we can always guarantee to get the best plan by following the sequence in Figure 4.3. That is the way the PostgreSQL optimizer does. So we override the *DoSearch* method, and apply Select operator first. After initial trees, which are relation scans, are generated, restricting qualifications are applied to their corresponding relations. In this way, all selections are pushed down, and no actions are required in subsequent operations. Also, the IndexCollapse operator is applied right after the Select operator so that we can make use of indexes on select-attributes. Namely, if there is an index on the select-attribute

coincidentally, the physical method to access the relation will be changed to be index-scan; otherwise, the initial tree is kept intact.

5.3.3 Sub-query-pull-up

There is no consideration on sub-query-pull-up in OPT++, so we have to do it after a query tree is generated and before it is passed to the optimizer. As described in section 4.3, to qualify to be pulled up, a query tree must be a simple query in FROM-clause.

5.3.3.1 Data Structure Changes

Because the framework does not consider sub-query-pull-up, the original representation of a query is not suitable for that. The original framework uses arrays to store a range table, attributes, and qualification expressions. If a sub-query is pulled up, some of the homogenous items will be merged, but arrays cannot be used to do the merging conveniently in C++, because we always need the original objects, instead of copies. Therefore, we changed them to be lists. Also, the framework does not consider explicit joins, so there is no data member for explicit joins. As mentioned in 5.2, class **Join_tree** is added to represent the structures.

5.3.3.2 Sequence for Pulling up a Sub-query

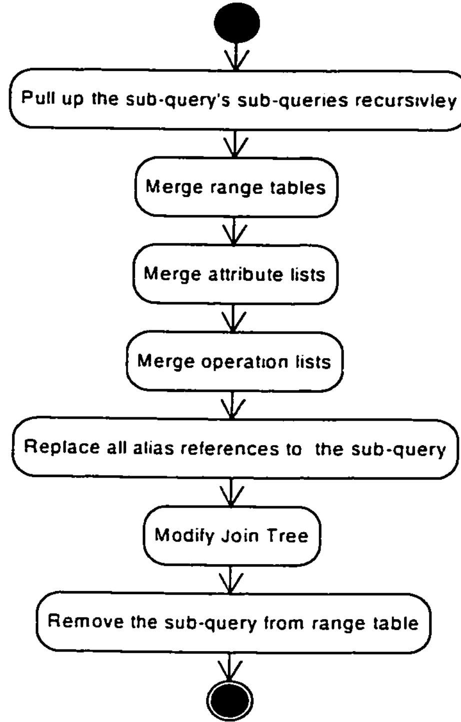

Figure 5.3 The Activity Diagram for Sub-query-pull-up

Figure 5.3 shows the activities of sub-query-pull-up. First, recursively pull up the sub-query's sub-queries if applicable. Then, a sub-query's lists of range table, attributes and operations (i.e. expressions in its target list, WHERE-clause) are merged into the current query. After the lists are merged, we can reference to the relations and attributes that were in the sub-query in current query. Next, all variables referencing to the sub-query by its alias in current query should be replaced with the items in the sub-query's relations and attributes. For example, in the query:

Sample query 1:

```
SELECT foo.a,... FROM r, (SELECT s.x+t.y AS a, ... FROM s,t WHERE...) AS
foo WHERE foo.a...;
```

foo.a appearing in both the target list and the WHERE-clause should replaced with s.x+t.y.

Now, we need to reconstruct the join tree. Sub-queries in a range table fall into two categories according to their positions: (1) It is a separate item in FROM clause, like

SELECT… FROM r,subquery,…WHERE…; (2) It is an item in a explicit join, like SELECT… FROM r INNER JOIN subquery …,…WHERE…; In the first case, adding the sub-query's join trees to the upper query's join tree list suffices. In the second case, especially when the sub-query has multi-item, and even explicit joins in its FROM-clause, we have to construct a new join tree based on the join trees in the sub-query. Figure 5.4 shows the join trees after pull-up for such a query:

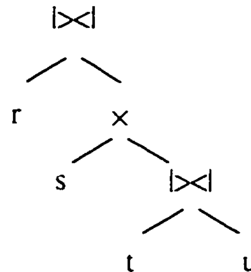

Figure 5.4 A Join Tree after Sub-query-pull-up

Sample query 2:

```
SELECT foo.a,… FROM r natural join  (SELECT s.x+t.y AS a, … FROM s,t
natural join u WHERE…) AS foo WHERE foo.a…;
```

Note that we construct a Cartesian product node when two items have neither explicit nor implicit join. Finally, we need to remove the sub-query' alias from the current query's range table. For example, we need to remove the sub-query's alias, foo, from the range table.

It is worthwhile to point out that the optimizer does not know if a query had some simple sub-queries after the pull-up, so no special treatment is needed for such a query in the optimizer.

5.4 Implementation of the Algebra Component

To customize the algebra component, we only need to define sub-classes of class **DBAlgorithm** and **DBOperator** and establish the relationships between the logical operators and their corresponding physical operators.

The logical and physical operators in our system are almost the same as PostgreSQL's as shown in Table 4.1. We do not implement set operations, in which there is nothing to optimize. There is one more logical operator, IndexCollapse.

Following the recipes given in Jinmiao's thesis [Li01], first we define the logical algebra, then the physical algebra, and finally the class **DBOperatorOperation**.

5.4.1 Define Logical Algebra

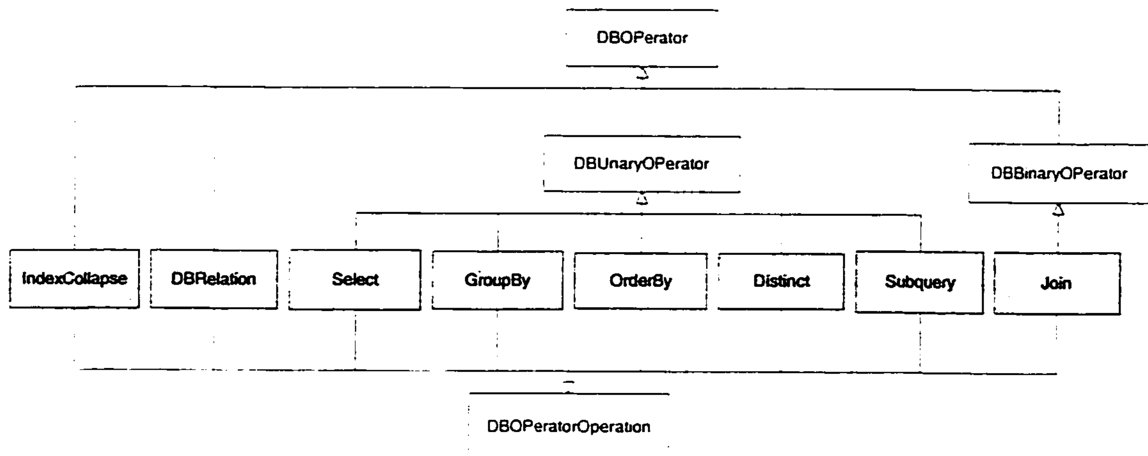

Figure 5.5 The Customized Logical Algebra

5.4.1.1 Sub-classing

As shown in Figure 5.5, eight concrete subclasses (adorned in gray) are defined to represent the logical operators in the logical algebra. They are **DBRelation**, **Select**, **GroupBy**, **OrderBy**, **Distinct**, **SubQuery**, **Join**, **IndexCollapse**. The operator **DBRelation**, representing a relation in a database, takes no input, so it is defined as an immediate sub-class of the abstract class **DBOperator**. Also operator **IndexCollapse** does not behave as either **DBUnaryOperator** or **DBBinaryOperator**, so it is defined as a subclass of **DBOperactor**, although it takes one input. Other sub-classes are defined as sub-classes of **DBUnaryOperator** and **DBBinaryOperator** according to their arities. All the concrete sub-classes also inherit from class **DBOperatorOperation** so that a concrete class can record what operations (i.e., expressions) are being applied by an operator. For example, the class **Join** has to record a join qualification, while class **Select** has to record a filter qualification. The operations will be used to calculate selectivity, determine whether two logical trees are equal, and judge whether a logical tree is complete for a query, as illustrated in section 3.2.

5.4.1.2 Implementation of Hook Methods

According to the framework recipes, the hook methods, *Accept*, *Duplicate*, *MakeLogProps* must be implemented, while *Clones* can be overridden if the default behavior does not fit the requirement.

The following are description for the implementation of the hook methods. For each hook method, we describe our implementation, and then an example is given. The hook methods for other classes were implemented in the same way.

- *Accept*

This is to accept a visitor. There are two kinds of concrete visitors, **ExpandTreeVisitor** and **TreeToPlanVisitor** in our application. By accepting the two visitors, the current operator is applied to the input operator tree, and the new tree is converted into a physical tree.

Note that we reused the Visitor Hierarchy designed for Bottom-up strategy.

Example:

DBRelation::Accept() is implemented as:

```
void DBRealtion::Accept(OperatorTreeVisitor& visitor) {
  visitor.VisitDBRelation(this);
}
```

The rest concrete sub-classes implement the hook method in the same way, except calling their own processing methods defined in visitors. The visitors, in turn, delegate the tasks to generators in the Search Space component.

- *Duplicate*

Given an object, this method is just to duplicate an object. Normally, an original object is generated when the optimizer is instantiated, and stored in a global object of class **OperatorAndAlgorithm**. All new created logical and physical operators are copies of objects in the global object.

The method of all the subclasses are implemented in the same way:

1. Initialize a new object;
2. Copy all members from the original object.

Example:

The method of class **OrderBy** is implemented as follows:

```
DBOperator *OrderBy::Duplicate (void) const
{
  OrderBy *d = new OrderBy (GetListOfAlgorithms ());
  assert (d); *d = *this; return d;
}
```

GetListOfAlgorithms is a public method of class **DBOperator** to return a logical operator's corresponding execution algorithms.

- *MakeLogProps*

This is an important method to calculate the logical property of an operator. This method is implemented only by delegating the task to a constructor of class **OperatorTreeProperty**.

Example:

The method of class Join is implemented as:

```
OperatorTreeProperty *Join::MakeLogProps (OperatorTree *node)
{
  OperatorTreeProperty *l = new OperatorTreeProperty (this, node);
  assert (l); return l;
}
```

The parameter, node, is a sub tree after the current operator, join, is applied.

For each concrete sub-class of class **DBOperator**, a corresponding constructor is implemented. Therefore, for class Join, the corresponding constructor of class **OperatorTreeProperty** is to calculate the size of output, set whether the node is interesting.

- *Clones*

This is a critical method, which is used to determine whether the operator represented by the concrete sub-class can be applied to the input logical sub-tree according to the query.

The hook method for class Join is implemented as follows: Given two inputs, which are sub-trees, if there is a join qualification, a clone is generated; otherwise, no clones are

generated. If a clone is generated, the Join operator will be applied to the two inputs; otherwise, the operator will be ignored.

Example:

Pseudo code for Join::Clones() is as follows:

```
list<DBBinaryOperator>    Join::Clones    (OperatorTree    *leftinput,
OperatorTree *rightinput)
{
    list_t<DBBinayOperator> clones ;
    Calculate join qualifications based on properties of the two
    inputs;
    if(there is a qualification)
    {
        Generate a clone by calling Join::Duplicate();
        Clones.add (new clone);
    }
    return clones;
}
```

For example, the left input tree contains relations, r1, r2, and r3, and the left input contains relations, r4 and r5. If there is a qualification r1.name=r4.name, then a clone should be generated based on the qualification; otherwise, an empty list should be returned.

Note that, because multi-attribute joins are not supported, there is at most one object in the list clones.

5.4.2 Define Physical Algebra

Physical operators are defined to implement the logical operators with which they are associated.

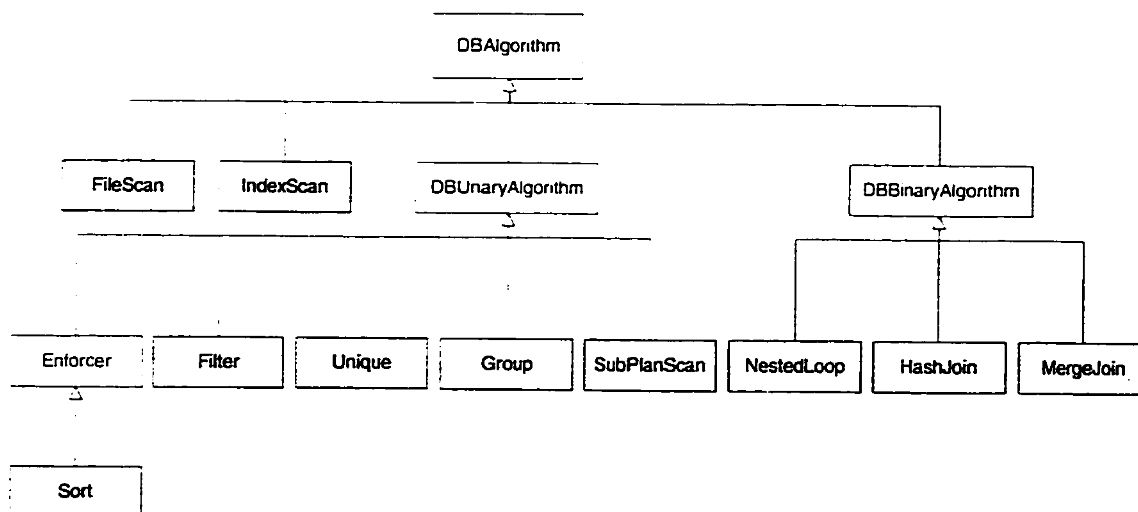

Figure 5.6 The Customized Physical Algebra

5.4.2.1 Sub-classing

In our optimizer, ten concrete subclasses (adorned in gray) of are defined, as shown in Figure 5.6. There are **Filescan**, **IndexScan**, **Filter**, **HashJoin**, **MergeJoin**, **NestedLoop**, **Group**, **Sort**, **Unique** and **SubplanScan**.

Note that Cartesian product is treated as a special join, which is a cross join.

5.4.2.2 Implementation of Hook Methods

For each of the concrete subclasses, the following hook methods are implemented as follows:

- *Duplicate*

The method is used to duplicate an object of a concrete physical operator.

Example:

The method of class **MergeJoin** is implement as:

```
DBAlgorithm *MergeJoin::Duplicate (void) const
{
    MergeJoin *h = new MergeJoin (enforcers); assert (h);
    *h = *this; return h;
}
```

Here, `enforcers` is a list of objects of class **Enforcer**.

- *MakePhyProps*

It is to calculate physical properties such as cost, and sort order. Like in logical operator classes, the task is delegated to one of constructors of class **AlgorithmTreeProperty**.

Example:

The method of Class **MergeJoin** is implemented as:

```
AlgorithmTreeProperty *MergeJoin::MakePhyProps (AlgorithmTree*node)
{
  AlgorithmTreeProperty *p = new AlgorithmTreeProperty (this, node);
  assert (p); return p;
}
```

- *MakePhyNodes*

This method is to generate all possible physical trees based on input(s). It is implemented in class **DBBinaryAlgorithm** and **DBUnaryAlgorithm**, so normally, we do not need to override it.

In our application, all execution algorithms, except Sort and Group, use the default implementation given by the two classes. The reason why the method is overridden in class Sort and Group is that they do not add new expressions to the tree's operation set in the current scheme of the framework, so the new tree will be as viewed as equal to the input tree. Therefore, if we use the default implementation of *MakePhyNodes* in class **DBUnaryAlgorithm**, the new tree will be deleted by the dynamic pruning mechanism. This will be explained in detail in section 5.7.3.

- *Clones*

In our application, the method *Clones* in physical operator class does no more than method *Duplicate*. It just duplicates an algorithm object, puts the copy in a list, and returns the list.

- *Enforce*

A concrete enforcer class must implement a method, *Enforce*. In this method, an enforcer treats its input according to the required properties described in an object of class **AlgorithmTreeProperty**.

The **Sort** class is worth some special attention, because it acts both as an execution algorithm for logical operator OrderBy and an enforcer for the physical operators, MergeJoin, Group and Unique.

The only enforcer in our algebra is Sort, and its method *Enforce* is implemented as (in pseudo code):

```
list_t< AlgorithmTree > Sort::Enforce (AlgorithmTree *node,
            AlgorithmTreeProperty *reqd_props)
{
    list< AlgorithmTree > enforced_outputs;
    if( node has the required properties)
    {
          enforced_outputs.Insert(node);
          return enforced_outputs; // return the unchanged node
    } else
    {
          Generate a new object of class Sort;
          Generate a new algorithm tree, newnode, by applying the
          sort object;
          enforced_outputs.Insert (newnode);
          return enforced_outputs;
    }
}
```

5.4.3 Set up Relationships between the Logical and Physical Algebras

After defining logical and physical operators, we need to set up relationships between them. The constructor of each logical operator class takes a list of corresponding physical operator as input. The relationships are established by adding objects of physical operator classes to corresponding list. The relationships between operation and execution algorithms are list in table 5.1.

Table 5.1 Operators and Associated Algorithms in Our Algebra

Operators	Associated Algorithms
DBRelation	Filescan
IndexCollapse	IndexScan
Select	Filter
Join	HashJoin, MergeJoin, NestedLoop
GroupBy	Group
OrderBy	Sort
Distinct	Unique
SubQuery	SubplanScan

Example:

The code for setting relationship between operator Join and its execution algorithms is like:

```
List<AlgoritmTree> join_algos;
......
assert (hh_join = new HashJoin);
join_aigos.InsertAtEnd (hh_join);
......
assert (join = new Ajoin_t (join_algos));
all_operators.InsertAtEnd (join);
```

Finally, all objects of logical operator classes are put into a list named **`all_operators`** so that the optimizer can visit them.

In our application, we applied one operator only once but in a fixed order, so the order in which we put the operator into the list **`all_operators`** is important. The order must conform to the sequence described in Figure 4.3.

In the Algebra component, implementation of all hooks, except the method *MakePhyNodes* of class **Sort** and **Group**, is routine. Therefore it is very easy to extend the algebra. The reason for the ease rests in that the customization is fully planned by the framework designer.

5.5 Implementation of the Search Space Component

The search space component determines how to apply a logical operator on a logical operator tree to expand the tree so that complete trees can be obtained. How to choose the logical operator and the tree is determined by the search strategy component.

5.5.1 Implement the Visitor Hierarchy

1. Add abstract methods to the class definition of **OperatorTreeVisitor** for each logical operator.

The format of the abstract methods is:

```
virtual void VisitXX (XX* )=0;
```

XX is the name of a physical operator class.

Example:

For class Join we need to add an abstract method like:

```
Virtual void VisitJoin (Join*)=0;
```

2. Add a method to the class definition of **ExpandTreeVistor** and **TreeToPlanVisitor** for each logical operator.

- The methods added to class **ExpandTreeVisitor**

The format of such a method is:

```
virtual void VisitXX (XX* );
{
    XXExpand expand;
    expand.Apply(op,currentTree);
}
```

XX is the name of a physical operator class.

From the implementation, we know the methods just delegate their tasks to subclasses of class **ExpandTreeGenerator** such as **JoinExpand**.

Note that **ExpandTreeVistor** is for the Bottom-up search strategy, but we reused it in our strategy. However, we control the search space further in our search strategy component, as it will be explained in section 5.6.1.

- The methods added to class **TreeToPlanVisitor**

Such methods like *VisitXX* are similar to the methods added to class **ExpandTreeVisitor**. However, all the methods have the same implementation, calling a private method named *VisitDBOperator*. By calling the private method, the class **TreeToPlanVisitor** creates all possible physical plan trees for a logical tree. The private method is provided by the framework, as illustrated in section 3.3.2.1.

Example:

```
void TreeToPlanVisitor::VisitJoin(Join* op) {
  VisitDBOperator(op);
}
```

5.5.2 Implement the Generator Hierarchy

1. Define a concrete subclass in the **ExpandTreeGenerator** hierarchy for each logical operator.

Corresponding to the logical algebra, subclasses defined are **DBRelationExpand**, **SelectExpand**, **GroupExpand**, **OrderExpand**, **UniqueExpand**, **SubQueryExpand**, **JoinExpand**, **UnionExpand**, **IntersectionExpand**, **ExceptExpand** , as shown in Figure 5.7. The concrete classes are adorned in gray.

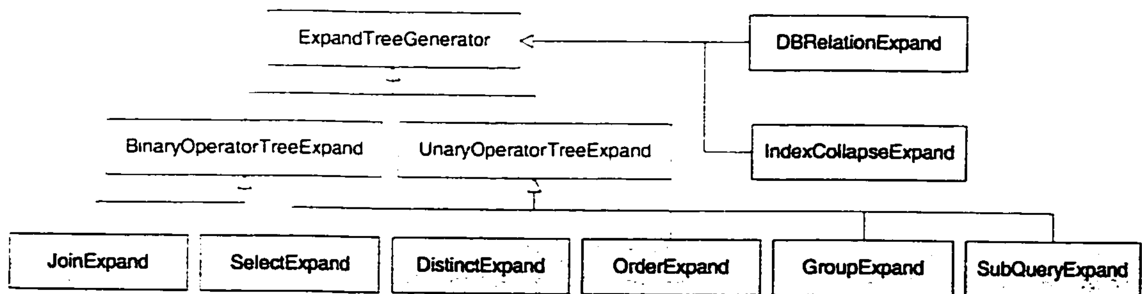

Figure 5.7 The Customized Generator Hierarchy

In our application, such concrete subclasses, except **JoinExpand**, are defined without any implementation, so they just behave as their immediate super-classes, **BinaryOperatorTreeExpand** and **UnaryOperatorTreeExpand**.

Example:

OrderExpand is defined for the logical operator **OrderBy** as a subclass of class **UnaryOperatorTreeExpand** without any implementation, so the method *Apply* of class

UnaryOperatorTreeExpand will be called at runtime. This can be illustrated with Figure 3.6. In the sequence diagram, **JoinExpand** is generated, and its method *apply* is called. However, for other operators, their generator classes have no implementation, the method *Apply* of the class **BinaryOperatorTreeExpand** or **UnaryOperatorTreeExpand** will be invoked at runtime according to the operator's arity.

The method *apply* of class **UnaryOperatorTreeExpand** is implemented by the framework, as illustrated in section 3.3.2.2.

The reason why class **JoinExpand** overrides its super-classes' methods is the key decision of our instantiation. The Search Space component is intended to expand a given tree by applying a logical operator on it. How to choose a tree and an operator is determined by Search Strategy component. All operators, except Join, conform to the framework design well. That is we can expand a tree once applying one operator. However, our strategy only searches all join space, instead of all plan space. **JoinExpand** needs to know all trees to be joined so that it can do dynamic programming and genetic optimization. This does not match the functionality separation between the Search Space and the Search Strategy components well. In our implementation, class **JoinExpand** caches the trees to be joined passed by Search Strategy component until all trees are received. When all trees to be joined are received, the dynamic programming or genetic algorithm is triggered according to the number of items to be joined.

Before we optimize the items in FROM-clause, each item should be optimized. As mentioned in 5.2, one item in FROM-clause can be separate relation (or non-simple sub-query, because simple queries have be pulled up), for which no optimization is needed, or an explicit join. For an explicit join, join order need not be considered, but we still have to decide which input acts as left/right input, and which physical operator is adopted.

A method, *MakeOneExplicitJoin*, is added to class **JoinExpand** to optimize an explicit join. In the method, one of constructors of class **OperatorTree** is called; therefore, the control flow defined by the framework is reused. Especially, the dynamic pruning mechanism is used to delete sub-optimal trees so that the space and time needed are reduced drastically.

Now, each item in the FROM-clause has been optimized. The inputs for dynamic programming and genetic algorithm are ready. Next, we can use the two algorithms to optimize the joins. After the dynamic programming or genetic algorithm is done, the best operator trees are passed to class **SearchTree** of the Search Strategy component for further optimization.

To balance the layout of the chapter, dynamic programming and genetic algorithm are described in 5.7, and the reason why class **JoinExpand** is implemented differently will be explained in section 5.7.2.

2. Define a concrete subclass in **AlgorithmTreeGenerator** hierarchy for each physical operator.

The concrete subclasses corresponding to physical algorithms are defined. For example, for class **FileScan**, there is a **FileScanPlan** class. Class **FileScanPlan**, **IndexScanPlan** do not behave as their super-classes, so their methods are implemented. The remaining classes behave as their intermediate super-classes, so there is no implementation within them.

FileScanPlan is implemented to generate an algorithm tree for a real relation (not a sub-query). **IndexScanPlan** is implemented to change the access algorithm from **FileScan** to **IndexScan**, if there is a select on a relation, and there is an index on the select attribute coincidentally. Class Sort and Group override the method *MakePhyNode* of class **DBUnaryAlgorithm**, in order to avoid triggering the dynamic pruning mechanism. Therefore, class **SortPlan** and **GroupPlan** are not used actually. The reason will be explained the section 5.7.3.

5.6 Implementation of the Search Strategy Component

The Search Space component determines how to explore the plan space. In the component, I need to implement the cost model and our own search strategy.

5.6.1 Implement Property Calculation

The cost model is encapsulated in the two property classes and class **Cost**, so we need to calculate logical and physical properties for operator and algorithm trees by implementing the three classes.

1. Logical Properties Calculation

In this step, a constructor of class **OperatorTreeProperty** for each logical operator is added. In such a constructor, the properties, such number output tuples, operations (i.e. expressions) applied so far, and **_is_interesting**, are calculated.

Because the physical layer of our target database has not been finished yet, the dummy system catalog is used. However, it is easy to change to make use of a real system catalog. To do that, what needed to be changed is selectivity of different predicates, so once more accurate statistics is available, the optimizer can calculate the properties more accurately simply by re-implementing a method named *seldesc* of subclasses of class **Operation** (in Figure 5.1).

We adopt selectivity of predicates provided by the framework. For example, selectivity of a "=" predicate is set to be 0.001 by default.

Example:

The logical properties of class Join is calculated in one of the constructors as follows (pseudo code):

```
OperatorTreeProperty::OperatorTreeProperty(Join    *op,    OperatorTree
*node)
{
    Set index path if applicable;
    If (there is a sort-order)
        _is_interesting=true ;
    calculate  _operation  based  on  properties  of  left  and  right
    inputs;
    double selectivity=1.0;
    if(op->IsCrossJoin())
    {
      selectivity=1.0; //for a cross join,the selectivity is 1.0;
    }else
    {
      calculate selectivity based on join qualifications.
    }
    Calculate  number  of  output  tuples  based  on  selectivity  and
    inputs' properties;
}
```

2. Physical Properties Calculation

In this step, a constructor of class **AlgorithmTreeProperty** for each logical operator is added. In such a constructor, the physical properties such as, **`_is_interesting`**, **`_cost`** are calculated. In our application, if a physical operator tree has some kind of sort order, the attribute, **`_is_interesting`**, is set to be true; otherwise false.

We added two new data members, **`_sort_order`**, **`_required_sort_path`**, to the class **AlgorithmTreeProperty**.

- **`_sort_order`**

It is a list of strings representing sort orders the physical tree have. For example, after executing a merge join, a physical operator will have sort order on the merged attribute. A list is used because there may be a major sort order and some minor sort order, but in the current implementation, we only make use of major sort order.

- **`_required_sort_path`**

It is a string representing a required sort path that is needed by the *Enforce* method of subclasses of class **Enforcer**. In our application, only one concrete enforcer class, **Sort**, is defined, and it needs to know what attribute should be sorted.

Example:

The calculation of MergeJoin's physical property is defined as follows (pseudo code):

```
AlgorithmTreeProperty :: AlgorithmTreeProperty (MergeJoin *algo,
AlgorithmTree *node)
{
    _is_interesting=1; //for a merge join produces some sort order;
    _sort_order.add(left_sort_order);   /*preserve sort order name on
                                          left input;*/
    _sort_order.add(right_sort_order); /*preserve sort order name on
                                          left input;*/
    Calculate the size of output tuples;
    _cost.compute (algo, node); /* call Cost::compute to compute the
                                          cost;*/
}
```

3. Cost Computation

Add a method *Compute* to class **Cost** for each physical operator XX to calculate the cost of the current physical operator tree.

In our application, the cost model suggested by the framework is adopted. Namely, both I/O and CPU cost are considered.

A typically computation is as follows:

Cost=number of pages×I/O cost per page+ number of pages×number of instructions per page×execution cost per instruction

In the equation, the three parameters, I/O cost per page, number of instructions per page and execution cost per page, are available in system catalog, so the only thing to compute is the number of pages. The variable, number of pages, can be calculated based on the property of the inputs and nature of the physical operator.

Example:

For a HashJoin:

number of pages=3 (B(l)+B(r));

B(l): number of pages of left input;

B(r): number of pages of left input;

The reason is that we need 2(B(l)+B(r)) to do hash first, then B(l)+B(r) I/Os are needed to load the hashed inputs.

5.6.2 Implement the PostgreSQL Search Strategy

In our application, a PostgreSQL-like search strategy is implemented. In the strategy, we generate initial trees representing relations. Right after that, selections are pushed down. Then joins are optimized. Finally SortBy, GroupBy etc. are converted into physical nodes in a fixed way. The strategy is described in 4.4.1. In other words, our strategy is a hybrid strategy. The activities in our strategy are the same as PostgreSQL's as illustrated in Figure 4.3.

There are two obvious features in our strategy:

- The logical operators are applied in a fixed order, while the order is actually algebra specific.

- Most effort is spent on joins. The dynamic programming and genetic algorithm are applied only to joins, instead of all the search space.

Although some regions of the search space are not explored, we never miss the regions that contain the optimum. Therefore, the best plan must be found.

In our implementation, the classes defined for the Bottom-up search strategy are reused to achieve our goals whenever possible.

1. Define class **PostgresqlSearchStrategy**

A new class **PostgresqlSearchStrategy** is defined as a subclass of class **SearchStrategy**. In the new class, only one method *CreateSearchTree* is overridden to create its corresponding search tree, an object of class **PostgresqlSearchTree**.

The code is:

```
SearchTree*PostgresqlSearchStrategy::CreateSearchTree(Query* query)
{
    return new PostgresqlSearchTree(query);
}
```

The framework does not consider sub-queries. In order to process sub-queries, the *Opitmize* method of class **SearchStrategy** was modified. Some code for identifying and optimizing sub-queries in expressions and the range table of a query tree was added. To optimize the sub-queries, the optimizer is called recursively.

2. Define class **PostgresqlSearchTree**

A new class **PostgresqlSearchTree** is defined as a sub class of class **SearchTree**

In this class, three main methods, *NewNode*, *MakeIntialTree*, and *DoSearch* of class **SearchTree** are overridden. The other virtual method *ExpandNode* is neither overridden nor used, because we think it is over-featured.

a) MakeIntialTree

In this method, we borrowed its implementation from Bottom-up strategy, in which logical operator trees representing relations or sub-queries are generated, and put into a list.

Note that ExpandTreeVisitor, which is for Bottom-up strategy, is used in the code:

```
void PostgresqlSearchTree::MakeInitialTree(void)
```

```
{
    ExpandTreeVisitor visitor;
    GlobalVariable()->opalgo->get->Accept(visitor);
}
```

Here, get is an object of class **DBRelation.**

b) DoSearch

In the method, the control flow of the strategy is implemented. After initial trees are constructed, each operator in the algebra is applied to all the trees generated by its previous operator, as shown Figure 5.8. The order is implied in the list containing all the operators.

The implementation is a compromise between efficiency and extensibility. The rationale behind the decision is explained in section 5.7.

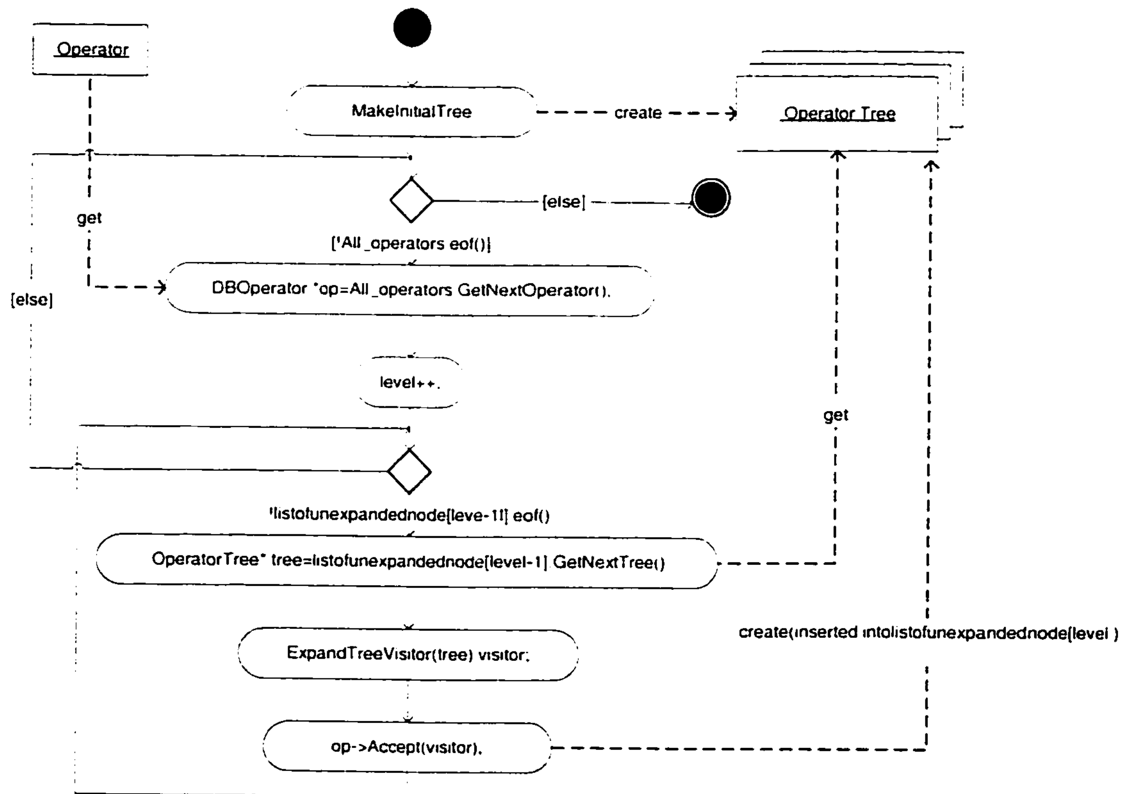

Figure 5.8 The Implementation of the Search Strategy

As far as the whole algebra is concerned, the following operations will be executed, when a query is optimized.

a) Select-push-down

We call code implemented for Bottom-up strategy to do select-push-down. First we get an object of class **Select** from a global variable. Then, its *Accept* method is called.

b) Index collapse

As described in 5.2, the step may find a cheaper way to access a relation, index-scan.

c) Optimizing joins

So far, we get initial logical trees representing relations, probably with selection on them. The trees are stored in a list named `listofunexpandednodes[level]`, which is an element of an array of lists defined in class **SearchStrategy**.

The task of optimizing joins is dispatched to class **JoinExpand**, where dynamic programming and genetic algorithm are employed to search join space.

d) Applying GroupBy, OrderBy and Distinct

In our scheme, after joins are optimized, there may be more than one logical operator tree remaining. The reason is that although some physical trees are sub-optimal, they may have some kind of sort order, so they and their parent logical trees cannot be deleted.

Now, we have to applied logical operators, GroupBy, OrderBy and Distinct, to the best trees in turn, if applicable. By calling the method *Clones* of class **Group** or **Order**, we can determine whether an operator Group or Order should be applied.

As in select-push-down, the control flow defined by the framework for Bottom-up strategy can be used here. However, we cannot make use of the dynamic pruning mechanism here, because they do not add any new operations (i.e. expressions) to the expression set of an operator tree. The reason will be explained in detail in 5.7.

Note that the physical operator Group and Unique, which correspond to the logical operator Group and Distinct, need an enforcer, Sort. Also, Sort is the execution algorithm of logical operator Order.

If a resultant join tree has the required sort order by chance, then no sorting is needed.

e) Return the best plan

Finally, it is time to choose and return the best plan. Among all remaining logical trees, we find a physical plan with the cheapest cost. If the current query is a top-level

SELECT query, the plan is the final plan; otherwise, it could be a sub-plan for its upper query.

5.7 Problems and Implementation Considerations

5.7.1 Generality and Efficiency

As a framework, extensibility is emphasized, so OPT++ implemented a search strategy with high generality by calling hook methods and using *list* data structure. Furthermore, OPT++ suggests the custom search strategies must be implemented totally in terms of virtual methods of predefined abstract classes without making use of knowledge of the actual algebra. By following the rule, the extensibility of a target optimizer will be guaranteed. However, without any knowledge about the actual algebra, we will sacrifice some efficiency, even greatly in some cases.

We can explain the point by analyzing the Bottom-up strategy implementation equipped with the framework. In the Bottom-up strategy, when a logical tree, which may be part of a complete tree, is expanded, all logical operators are applied on it and new operator trees are generated in the process. The process repeated until there are no unexpanded operator trees. The process is presented in Figure 5.9.

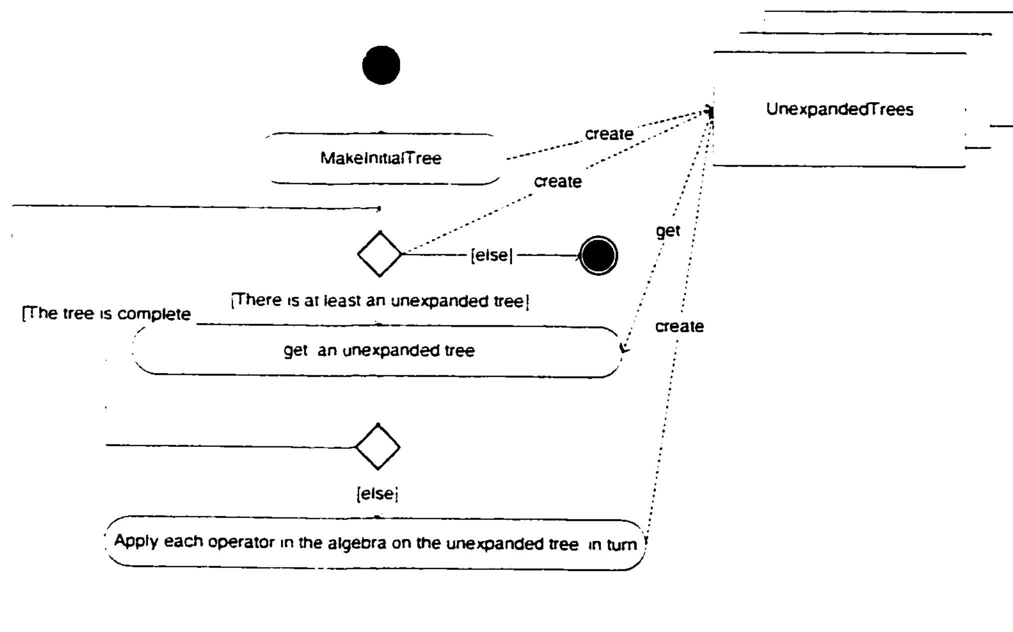

Figure 5.9 The Implementation of Bottom-up Strategy

The implementation is less efficient because the complete search space is explored, even the regions that cannot contain the optimal plan.

For example, in a query:

```
SELECT * FROM employees AS e, departments AS d WHERE
e.dept=d.id AND e.age>25 ORDER BY name;
```

at least such cases should not be considered in optimization:

- Selects are applied after joins;
- OrderBy is not applied last.

When we apply selects after joins, explicit select-push-down is needed. When there are a lot of relations in a query, the number of join combinations could be huge. For each combination, a select-push-down must be done once, if we apply selects after joins. Therefore, the cost of select-push-down may not be negligible. Also, if OrderBy is not applied last, the result sort-order it produces may be destroyed by other algorithm, for example, HashJoin.

In our application, we made a compromise between efficiency and generality. Our strategy applies each logical operator in a fixed order only once for optimizing a query, as described in Figure 5.8. We achieve efficiency by stipulating an order of applying operators. The order conforms to the nature of SQL. In this way, some regions of the search space that cannot contain the optimum are excluded.

We keep space for extension by not mentioning concrete operators in our strategy and delegating tasks to the Search Space component. As shown in Figure 5.8, we get a pointer to each concrete operator from a list (`all_operators` in Figure 5.8), and call its virtual method *Accept* through polymorphism. If a new operator is added, we do not need to change our strategy code, as far as the applying order is respected and the corresponding generator classes are implemented properly.

5.7.2 Framework Mismatch and Solution

The framework is composed of three components, Algebra, Search Space and Search Strategy. Both Search Space and Search Strategy are to control how to explore the plan space. Search Space is designed to control search behavior within an operator, namely given a tree and an operator, The Search Space component determines how the tree is

expanded. Search Strategy is designed to control inter-operator search behavior, i.e., how to select a tree and an operator for the Search Space component. In this way, the control can be tuned in two levels. Generally speaking, the design is flexible.

However, the design mismatches our strategy in part. The problem is that: on the one hand, the Search Space component is design to apply one operator to one operator tree. On the other hand, our strategies to optimize joins, dynamic programming and genetic algorithm, need to know all the operator trees representing the items in the FROM-clause.

To solve the mismatch, there are some alternative solutions. One solution is to treat joins differently in the Search Strategy component without dispatching the task to Search Space component. If so, we have to mention the operator Join in the strategy component. However, the rule guaranteeing extensibility of the target optimizer will be violated, because the framework requires that a custom strategy must be written in terms of virtual methods of the predefined abstract classes without using any actual algebra information.

The solution we adopted is to cache the operator trees to be joined in the Search Space component, specifically class **JoinExpand**. The optimization is not triggered until all trees to be joined are received.

To do so, we need a global variable (a list) to cache the pointers to the operator trees to be joined. The implementation of the method *Apply* class **JoinExpand** is shown in Figure 5.10.

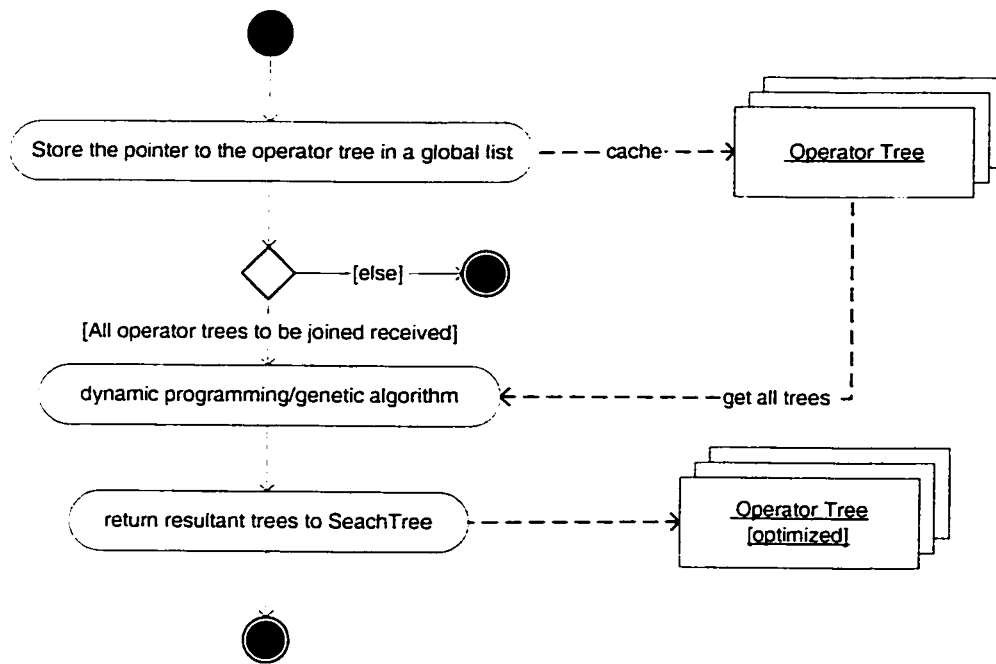

Figure 5.10 The Implementation of the Method JoinExpand::Apply

5.7.3 Special Treatment for GroupBy and OrderBy

One of the most successful aspects of OPT++ rests in its dynamic pruning mechanism. It deletes the physical and logical sub-trees, which are determined not to be a part of the final optimal tree. Using the mechanism is safe and convenient.

However, to process the logical operators GroupBy and OrderBy, the mechanism cannot be used in the current OPT++. As explained in 3.2.2, two logical trees are viewed to be equal when they have same set of operations (i.e. expressions) applied. In the current scheme, relations and attributes are all viewed as expressions, and no duplicates are allowed in the universal expression set for a query. Another fact is that when an initial tree representing a relation is generated, the relation's attributes are also introduced into the expression set of the tree. This is necessary for the materialization operator for relational-object database. However, this causes a problem in our application: The two operators, GroupBy and OrderBy, cannot introduce new expressions.

For example, a query:

SELECT * FROM Employees, Departments WHERE age>25 ORDER BY age;

The attribute, age, can appear only once the universal expression set, `_list_of_operations` in class **Query**.

Therefore, after a join tree is generated, when we apply the OrderBy operator to the join tree, no new expression will be introduced into the expression set of the current tree, because the attribute, age is already there (All attributes including age were introduced when the relation Employees is introduced). The optimizer will consider the new tree is equal to the join tree, and delete the new one. To avoid the incorrect treatment, we have two choices:

- Change implementation of *DBRelation::Clones* not to introduce attributes when introducing relations to the operation set of a logical operator tree.
- Override the method *MakePhyNodes* of class Sort and Group to avoid triggering the pruning mechanism.

We adopted the latter. If we do not override the method, the method of their immediate super-class, **DBUnaryOperator**, will be invoked at runtime, and the pruning mechanism will be triggered.

5.7.4 Constrained Dynamic Programming

We implemented constrained dynamic programming strategy in the same way as PostgreSQL. We enumerate all promising combinations of the items, and then call a constructor of class **OperatorTree** as in the method *MakeOneExplicitJoin*. Similarly, the control flow and dynamic pruning mechanism defined by the framework is reused.

5.7.5 Genetic Algorithm

We implemented the genetic algorithm by imitating the PostgreSQL optimizer. The flowchart of our implementation is described in Figure 4.7.

Classes

Three new classes are defined in our application to implement the genetic algorithm.

- Class **Pool**: representing the population of individuals.
- Class **Chromosome**: representing an individual. It reserves a string of integers, each of which is just the numeral of an item in FROM-clause.
- Class **Edge**: representing an edge between two cities in the TSP problem.

The three classes are designed to be a reusable. The three classes are tightly coupled, but have loose connection with other classes.

The only interfaces with its caller, the class **JoinExpand**, are the constructor, *Pool*, and the public method *gimme_best_chromosome*.

The package required its caller to provide a method named *GeneticEvaluate* to compute the fitness of each chromosome.

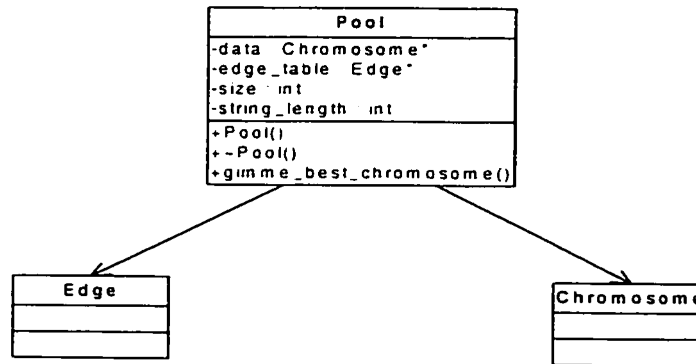

Figure 5.11 The Classes for Genetic Algorithm

In this way, the sub-structure are reused with no or minor modification.

The same bias function for selecting parents as PostgreSQL is adopted, and one of crossovers, Edge Recombination Crossover [Pos02b] is implemented.

Reuse of the Framework.

Although the framework does not provide a similar strategy, we can take advantage of the facilities as in the constrained dynamic programming strategy. To compute the fitness (execution cost) of each chromosome, a logical tree must be generated. Because only left-sided trees are considered, a chromosome, which is a permutation of numerals of items in the FROM-clause, stipulates a unique way to construct a logical tree. Therefore, a constructor of class **OperatorTree** is called to construct operator trees, as in method *MakeOneExplicitJoin*. Likewise, we obtain the same benefits.

Delete logical trees

As mentioned above, in the course of genetic optimization, a good number of complete logical trees are generated tentatively so that fitness of the corresponding chromosomes can be computed. If we do not delete the trees dynamically, a lot of

memory will be wasted. In the dynamic programming approach, only one complete logical tree is generated eventually, while other sub-optimal incomplete trees are dynamically deleted by the pruning mechanism. However, in the genetic algorithm strategy, all the trees are complete. If we use the pruning mechanism, only one complete tree can be generated; others will be pruned out. Therefore we cannot rely on the pruning mechanism. Consequently, we have to implement this ourselves.

The method *DeleteGeneticTree* is defined in class **JoinExpand** to implement the function.

In the method, we delete a logical tree from top to bottom. Although we have to implement the new function, we still reuse the framework by calling one of its global functions, named *ADeleteTree*, which is well-designed function for deleting a logical sub-tree and its associated physical trees.

5.8 Testing the Target Optimizer

To verify the target optimizer, we did correctness and efficiency tests.

5.8.1 Correctness Test

For correctness tests, we optimized a group of sample queries with our target optimizer and the PostgreSQL optimizer, and then we compared the output plans generated by the two optimizers. The following are the test cases and results.

1. Simple Queries

Objective: In the case, we test the simple queries with only SELECT and FROM clauses, but without explicit joins.

Example Query: `SELECT * FROM Persons p, Employees e;`

Results: The two optimizers generate the same results. The output plan does a nested-loop join between the two relations.

2. Selects

Objective: In the case, selects on relations are tested. There are two sub-cases. One is that there is no index on the select attribute. The other is there is such an index.

Example Query:

```
SELECT * FROM Employees  WHERE salary>10000;
```

```
SELECT * FROM Persons p where p.age=25;
```

Note that there is an index on Person.age, while there is no index in relation Employees.

Results: The two optimizers generate the same results. In the output plans, if there is a required index, the relation is accessed through index-scan; otherwise, the relation is accessed through file-scan.

3. Select-push-down

Objective: In the case, select-push-down is tested.

Example Query:

```
select * from Persons p,Employees e where p.name=e.name and
e.name="Smith" and p.age=25;
```

Results: The two optimizers generate the same results. In the output plans, if there is a select on a relation, and the relation is joined with another, select is applied first, and then join. Furthermore, if there is a required index on select attribute, for example, p.age, in the example query, access method to the relation is index-scan.

4. Explicit Joins

Objective: In the case, explicit joins with join directives are tested. Both explicit joins with qualifications and natural joins are tested.

Example Query:

```
SELECT * FROM Persons p INNER JOIN Employees e ON (p.name=p.name);
SELECT * FROM Persons p NATURAL JOIN Employees e, Cities c where
p.name=e.name and p.age=25;
SELECT * FROM table1 NATURAL JOIN table2;
```

Results: The two optimizers generate the results with the same join orders, but the join methods may vary with the system catalogs. In the output plans, join orders are determined by the explicit joins. Especially, in a natural join, if there is no common attribute between the two relations, we just do a Cartesian product. For example, in the third example query, there is no common attribute between table1 and table2, so we do a Cartesian product.

5. Implicit Joins

Objective: In the case where joins implied by WHERE clauses are tested.

Example Query:

```
SELECT * FROM table1, table2, table3 WHERE table1.a011=table2.a021
AND table2.a022=table3.a032;
```

Results: The two optimizers generate equivalent results. Although the join orders and methods may vary with the system catalogs, the join trees containing the same relations are generated. Therefore, they are equivalent.

6. Sub-query-pull-up

Objective: In the case, sub-query-pull-up is tested.

Example Query:

```
SELECT * FROM Departments AS d, (select * from Employees e, Persons
p where p.name=e.name) AS foo;
```

Results: The two optimizers generate the results with the same join orders, but the join methods may vary with the system catalogs. In the output plans, there is no sub-plan generated. This indicates that the simple query in the FROM clause is pulled up.

7. Sub-queries

Objective: In the case, both sub-queries in FROM and WHERE clauses are tested.

Example Query:

```
SELECT * FROM Departments as d, (select name from Employees
e,Persons p where p.name=e.name ORDER BY name) as foo where
foo.dept=d.id;

SELECT * from Departments AS d inner join (select * from Employees
e,Persons p where p.name=e.name ORDER BY name) as foo ON (foo.dept=
d.id);

SELECT * from Employees where name = any (select Persons.name from
Persons where age>25);
```

Note that putting the ORDER BY clause in the sub-queries looks strange, but if we remove them, the sub-queries will be viewed as simple queries and pulled up. They are put there just to prevent the sub-queries from being pulled up.

Results: The two optimizers generate equivalent results. In the output plans, sub-queries are optimized first and corresponding sub-plans are generated.

8. Order by Clause

Objective: In the case, we test the ORDER BY clause.

Example Query:

```
SELECT e.name, p.address FROM Employees e, Persons p where p.name=e.name ORDER BY name;
SELECT * FROM Employees e, Persons p where p.name=e.name ORDER BY salary;
```

Results: The two optimizers generate equivalent results. In the output plans, sub-queries are optimized and corresponding sub-plans are generated. In the first example query, after merge join between the two relations, no sort operation is needed, because the merge join generates the required sort-order. Please note that whether this happens depends on system catalog. In the second query, a sort algorithm must be executed after the join.

The tests indicate that the target optimizer produces correct execution plans on the test queries.

5.8.2 Efficiency Test

To test the efficiency of our optimizer, a group of queries is optimized with both our optimizer and the PostgreSQL optimizer on the same testing platform. Then, the optimization times are compared. Because most of the optimization time is spent on joins, we tested a group of queries with different numbers of joins. We did the tests on a personal computer, with 256M of memory and 1.7 GHz CPU. The operating system is Linux 2.4.18, and the compiler is GNU g++ 2.96. Table 5.2 shows the test results.

Table 5.2 Optimization Times of the Two Optimizers

(microseconds)

Number of Joins	The PostgreSQL Optimizer	The target Optimizer
1	318	310
2	580	770
3	921	2196
4	1884	4774
5	2827	9258
6	4163	16311
7	6168	27750
8	8103	43755
9	10615	65707
10	664873	2224559
11	745303	2590484
12	835139	2945270
13	924679	3280735
14	1009752	3696897
15	1107406	4206457
16	1191265	4690540
17	1482741	5251568
18	1512814	5711181
19	1829051	6392086
20	1922496	6711760

Note that when there are more than 10 joins in a query, the genetic algorithm is used. The genetic algorithm initializes a fixed size (1024 by default) pool of chromosomes, so the time increases drastically here.

From the results, we observe that the target optimizer is less efficient than the PostgreSQL optimizer. We studied the two optimizers from the perspective of efficiency, and found that the following factors contribute to the difference.

- The two container classes provided by OPT++ are inefficient. OPT++ defines two container classes, **Set** and **List**, providing easy-to-use interfaces. They are used very frequently as a basic facility. Unfortunately, they are very inefficient. In our testing environment, accessing an element in a set will cost about 5 microseconds, while a STL set does it in about 0.1 microsecond. In a query with 10 joins, we

need to access elements in sets more than 6000 times. Therefore, a lot time is spent on set and list operations. If the two container classes are improved, the optimization time will decrease to a great extent.

- The PostgreSQL optimizer does not generate logical operator trees, while the target optimizer generates logical operator trees. Although equivalent computations must be conducted in both the two optimizers. We need to allocate memory for the logical trees at runtime, and allocating memory dynamically costs more time than other commands in general.
- The indirection of design results in the efficiency degradation to some degree. Although OPT++ declares it improves extensibility without sacrificing efficiency, our experience indicates efficiency is discounted to some degree. For example, to determine whether two logical sub-trees can be joined, it needs to visit the universal expression set and the expression sets of the two trees, and then gets the result by doing some set operations. For any two sub-trees to be considered, the routine is executed. In PostgreSQL, this is conducted straightforwardly by analyzing expressions in WHERE clauses.

According to Navin[Kar99], on a Sun SPARC-10/40 with 32MB of memory, an OPT++-based optimizer costs about 10 seconds to optimize a query with 10 joins. Considering the platform difference, we believe our target optimizer achieves the efficiency expected by the framework designer. Although the target optimizer is less efficient than the PostgreSQL optimizer, its optimization efficiency is still in a reasonable range. If the defects of the container classes in OPT++ are removed, the efficiency will be closer to that of the PostreSQL optimizer.

5.9 Reuse Summary

The framework is reused in different granularities and levels in our application.

- Architectural Design — The three-component architecture is well designed. Therefore, in our application, no architecture design is needed. Our implementation follows the design of responsibility distribution among the three components. The algebra is implemented by sub-classing class DBOperator and DBAlgorithm.

- Classes and relationships among them — Our experience indicates that structural artifacts tend to have higher reusability. The classes and relations are well designed. The classes OperatorTree, AlgorithmTree , OperatorTreeProperty and AlgorithmTreeProperty can be reused easily to represent logical and physical trees without any changes. While other abstract classes are reused by sub-classing them.
- Exemplary control flow — Bottom-up strategy and transformative strategies are supplied as sample strategies. Not only do they help understand the framework, but also can be reused in part in other strategy. In our approach, we reused the control flow defined by the Bottom-up strategy as mentioned in 3.6.
- Functions — Sometimes, we can also reuse some global or static functions defined by the framework, although we cannot reuse the whole control flow.

  Examples:

  Prune: is an important static method to prune out the equivalents with more expensive. We reuse it in the method *MakeAnExplicitJoin* to optimize an item in FROM-clause.

  ADeleteTree: we reuse the global method to delete a logical tree and its associated physical trees in the genetic algorithm.
- Segments of code — When we override and overload a function, it is possible to find some useful segments of code in the original function. We can copy and reuse them.

5.10 Limitations of Our Implementation

Most of the PostgreSQL optimizer's features are implemented in our optimizer. However, as a study project, there still are some details that are not considered. As far as query optimization is concerned, the limitations of our optimizer are:

(1) Correlated sub-queries [Gar00] are not supported. This is the most complex case of sub-query processing. Although the basic strategy is the same as processing other kinds of sub-queries, more work is needed to do for this kind of sub-queries.

(2) Minor sort-orders are not used. We make use of only major sort-order, while the PostgreSQL optimizer does consider the minor sort-orders.

(3) Not all data types are considered. We considered only some typical data type in attribute definition, while the PostgreSQL optimizer take a great effort to treat more precise data types. For example, for *int* type, they consider int8, int16 etc.

(4) More accurate system statistics are needed. We use a dummy system catalog, in which only estimated statistics is available. In the PostgreSQL system catalog, much more accurate statistics are maintained. For example, for an attribute of a relation, numbers of most common values are available.

Chapter 6 Issues in Framework-Based Development

Framework-based development requires new some special considerations compared with traditional application development. While a lot of attention has been paid on methods for framework development, problems in using framework are not studied too much. In this chapter, general issues in framework-based development are discussed. We clearly know the one instantiation iteration for one framework is far from enough to general rules, so for each issue, related work is summarized first, and then suggestions are given based on our experience.

Because our experience is on a white-box framework, all the issues are discussed in terms of white-box frameworks.

6.1 Development Process

6.1.1 Process for General Software Development

Generally speaking, independent of the software life cycle model chosen, the software development process can be divided into such phases [Bus93]:

- Domain Analysis

The Domain Analysis or Subject Analysis phase is aimed at getting an understanding of the problem. The result of this phase is a description of the task that is to be solved by the software.

- Requirement Analysis

Requirement Analysis aims to elicit and identify both the functional and non-functional requirements for the intended software to be developed. The result is normally a list of requirements that has to be fulfilled by the software.

- System Analysis

The Systems Analysis phase concentrates on what to do in the application. The entities identified in the real world task are mapped to a software model. The system's functionality is mapped to software components and their relationships and the components responsibilities are defined. The result of this phase is called the conceptual framework.

- Architectural Design

The Architectural or System design phase deals with the main structuring of the application to be built. The architectural framework for the application is chosen and the conceptual framework is mapped to the architectural framework.

The subsystems and the relationships between the subsystems are specified. The result of the phase is a model of the application comprising its components, their responsibilities, high-level data structures as well as their relationships.

- Detailed Design

In the Detailed Design phase the responsibilities and data structures are specified in full detail. Principles and policies for implementation of the data structures and the subsystems are defined.

- Implementation

The final implementation of the system takes place in the Implementation phase using the results from the Architectural and Detailed design phases.

- Testing

Finally, the functional and non-functional requirements for the application are validated in component tests, system tests, and integration tests.

- Maintenance

When the application is in operation it will after some time require modifications due to new requirements or operating environments. These changes are handled in the Maintenance phase.

6.1.2 Process for Framework-Based Development

Related Work

Mattsson [Mat96] proposed a development process for framework-based development. The process is composed of such activities:

- Activity 1: Define the conceptual framework for the application. It includes the assignment of functionality to components, the specification of the relationships between them, and the organization of collaboration among them.
- Activity 2: Select an object-oriented framework for the application.

- Activity 3: Map the entities of the conceptual framework for the application into the selected object-oriented framework and subsystems/sub-frameworks (if any).
- Activity 4: Specify or revise the relationships between the sub-frameworks of the application's object-oriented framework. The selected framework may be composed of a number of sub-frameworks with predefined relationships, which have to be revised or remain unchanged.
- Activity 5: Structure the various sub-frameworks of the object-oriented framework. The components of the conceptual framework that has been mapped into each sub-framework are to be further organized with the purpose of achieving the required non-functional properties for the application as a whole.
- Activity 6: Structure the different components of a sub-framework. This includes deciding which parts of the sub-framework will continue to be hot spots and which parts that will be frozen spots.
- Activity 7: Implement the software system. Code all the classes that have been specified.

As Mattsson emphasized in his thesis, the activities are general. The activities in instantiation of different frameworks may vary a lot. Moreover, there is no practice supporting his advisory process.

Although Garry Froehlich did not propose explicit activities, he provided some general technique for different stages of framework-based application. [For99]

- Analysis

Besides activities in general development process, choosing a framework is also a main task in the phase. He proposed some guidelines for choosing frameworks.

- Learning to Use the Framework

In framework-based application, understanding a framework takes considerable time and effort.

- Design and Implementation

Since the framework should already define an abstract design of the application and provide much of the implementation, users have much less work to do at this stage.

- Testing

There is no special requirement on applications developed from frameworks. Both common functionality from the framework and application specific functionality need to be tested.

Although little work has been done on the issue, there is some consensus:

- There is no existing method for framework-based application, but we need it.
- Process for framework-based application should be domain specific, or even framework specific.

Our experience

Architecture design is not needed in our case. Because we instantiated only one framework, instead of cooperating multi-frameworks, the architecture was defined be the framework. The OPT++ decomposes the system into three components, and defines the collaborations among them. Our instance followed the architecture.

Studying a framework helps analysis. Framework can be reused in many levels including domain analysis. When we do analysis, we usually do not know what to do and how to do exactly. A well-documented framework provides us the framework developer's considerations. Therefore, the phases studying a framework may start from domain analysis until implementation.

6.2 Understanding Frameworks

When a white-box framework is used, it is necessary to understand the concepts and architectural style of the framework in order to develop applications that conform to the framework.

However, understanding frameworks is nontrivial work in framework development. In most cases, it is difficult to understand framework, because: [But00]

- The design is abstract;
- The design in incomplete;
- The design may provide flexibility that is not needed in our application;
- Collaborations and relationships may be indirect and obscure.

Therefore, learning is the key issue in framework instantiation, and sufficient time should be allocated for it.

Related work

To improve the understandability of frameworks, a large amount of work has been done on framework documentation. The documentation techniques include: [But00]

- Example applications
- Recipes / cookbooks
- Contracts
- Design patterns
- Framework overview
- Reference manual

Besides documentation methods, there are also other proposals on the problem. To communicate with common notation between framework developers and users, an UML profile, UML-F was proposed by Marcus Fontoura et al. [Fon02] This UML extension allows the explicit representation of framework variation points. It can be used to assist framework development and instantiation. UML-F provides not only a notation, but also some kind of methodology. They believe that the hook and template relationship cannot be represented with standard UML relationships. Two principles, unification and composition, were proposed to assist framework instantiation.

Also, an empirical study has been conducted to find out how the framework user can understand a framework more effectively. Forrest Shull [Shu02] studies the process of using and learning a framework to develop graphical user interfaces by students in an academic setting. He proposes that techniques based on examples are the most suitable for supporting learning, especially for novice users. He classifies the style of studying framework into hierarchy-based and example-based. The result shows that learning by example (as opposed to gaining familiarity with the framework itself first) is useful for helping beginning learners produce working systems quickly.

An alternative way to attack the understanding obstacle is to reduce the need for understanding the framework. Some approaches have been proposed to automate the instantiation to some degree. Oliveira's approach [Oli02] is to represent the framework design with XML. Instantiation process characteristics like sequencing and dependency can be specified through unambiguous representations. The framework then can be

instantiated declaratively. M. F. Fontoura [Fon00] presented the same idea, but domain specific languages are used to represent the framework design and instantiation requirement in his approach.

In each approach mention above, tools are developed to aid understanding framework.

Our experience

Our experience shows both framework providers and users can do. Framework providers can improve the understandability by:

- Providing reasonable degree of abstraction

  Both over-featured and under-featured spots do harm to users' understanding. In OPT++, we believe some methods, for example the method, *ExpandNode*, for the Bottom-up strategy is over-featured to be part of an abstract class.

- Programming in good style

  Good programming style will help users understand the designer's intent. For example, in C++, const member methods will require users not to change any data members of the class when they are overridden.

- Using De facto standards

  There are many de facto standards in software society and in the problem domain. Using standards will improve understanding of frameworks. For example, STL (The Standard Template Library) is well accepted in the C++ circle. No doubt, STL containers will be more understandable than self-defined container classes. Probably because STL was not mature enough when OPT++ was developed, OPT++ defines some container classes, such as List, Set.

- Offering sample data

  Among with exemplars, sample data should be given to make the exemplars more effective. At runtime, sample data may trigger some routines that cannot be executed when users give random inputs. Sample SQL queries will trigger the routines that are not straightforward to users. User then can gain better understanding by following the routines.

Framework users can understand a framework more effectively by:

- Understanding domain knowledge

Frameworks are not for end users, but for application developers. Therefore, framework users must possess enough domain knowledge. In our project, we cannot reuse the framework without query optimization knowledge.

- Mastering De facto standards

  Now that the framework developers may design frameworks with standard technology, mastering the standards will accelerate the understanding process. We take it for granted that application developers are familiar with the standards in his fields.

- Using debugging tools

  For a programmer, it is very intuitive to follow the execution of program step by step. For a white box framework, we do that by using debugging tools, such as GDB on Linux. In this case, exemplars and sample data are very helpful.

Our experience also supports one of Forrest Shull's conclusions [Shu00]. Example approach is more effective at the beginning of understanding the framework. That conforms to nature of cognition, from particularity to generality. However, we also found following recipes will achieve completeness easily at late stages of implementation.

Understanding frameworks may start with domain analysis, and extend to implementation phases. When a framework is complex, it is impossible and unnecessary to understand the framework completely before we start our design and implementation. With the process of our implementation, our understanding will deepen.

6.3 Unplanned Customization

No matter how elaborate framework development is, there always are some things that are not anticipated and planed. Therefore, we need do some unplanned customization sometimes.

There is very little literature talking about the issue, so, based on our experience, we propose the following suggestions.

- Respect the basic framework design

  We believe modifications to frameworks should be allowed, because in this way, frameworks may evolve. However, the basic framework design must be respected. In our project, we modified the representation of a query tree to fit sub-

queries optimization and explicit joins. These modifications are minor and do not change the basic design of the framework. The most important parts of OPT++, we believe, are operator/algorithm tree representation and dynamic pruning mechanism. Therefore, they are never modified.

- Sacrifice generality when necessary

  Some unplanned customization is caused by generality of frameworks. Sometimes, not only framework itself is general, but also it requires its instances to possess generality of some degree. However, we may know more our domain more clearly, so it is possible to make specific assumptions about our domain. In this case, sacrificing generality may release framework users from constraints, and thus, solve the unplanned problem. In OPT++, the Search Strategy component requires a strategy to be written absolutely in terms of predefined abstract methods without making any assumptions about the actual algebra. However, the optimization of SQL queries does have some kind of order. This is not planned in the framework. We solve the problem by ordering the optimization of operators as introduced in 5.7.1.

- Reassign the functionality of components

  Frameworks provide architecture reuse by defining the components and their relationships. However, the decomposition does always fit our applications. In this case, we may change the functionality of some components. In OPT++, the Search Space component is design to apply one operator on one given tree, while how to select the operator and the tree is determined by the Search Strategy component. In our application, we need to know all the trees in order to do dynamic programming and genetic optimization. Therefore, the function of class **JoinExpand** is changed to cache the trees to be joined and then do search in join space. This has been explained in 5.7.2.

- Report modifications to framework developers

  Frameworks are generalized from specific applications, and they evolve in instantiation. Problems encountered in instantiation may help framework developers to refine the framework. For example, in our project, modification of

the query tree representation can be reused by other applications. Thus, it should be reported to the framework developer.

6.4 Need for Tools

Object-oriented frameworks can be complex and difficult to use. Tool support can greatly aid developers in using these frameworks. In our process of instantiation, there was no tool supporting our development, but we realized tools would help a lot in framework-based application.

Related work

Along with the different methodologies for framework development and instantiation, many tools have been proposed. Each tool is based on some kind of model or method.

Garry Froehlich et al. developed a tool called HookMaster[Fro98] for their hook model. Based on their hook model, the tool helps describe how the framework is intended to be used, and shows where changes can be made. The hook tool aids users by extending the UML language to include hooks and by semi-automatically enacting the changes within hooks.

For the UML-F approach [Fon02], a supporting tool was presented. It can guide users to create a new subclass implementing hook methods. The tool prompts the application developer about all the required information to complete the method for each variation point in the framework structure. The instantiation tool is a wizard driven by cookbooks.

Tools are more important in the approach of instantiation automation. To support their declarative approach, Oliveira et al. proposed an XML based framework instantiation tool called xFIT [Oli02]. It can interact with the reuser to capture the application specific requirement (Application Specific Increments (ASI) in their paper) in a window-based environment. This ASI semantic is then used to generate a complete final design based on the framework represented in their model.

Our experience

From our experience, we realized tools could be very helpful in all stages of framework-based application.

Ideally, a tool for framework instantiation should have such functionalities:

- Basic visualization and navigation

The class hierarchy should be visualized. The framework classes and instance classes can be viewed clearly. Hot spots and hook methods can be identified graphically. Navigations among related classes and methods are easy. The functionality can be implemented language-specifically. but method-independently.

- Instantiation guidance

  Based on framework documentation. a tool should guide the instantiation process step by step. For example. what classes should be derived. what methods should be overridden or implemented. This must be implemented in a method specific way.

- Violation check

  Some violations can be found by compilers during compilation. It may be impossible for a tool to find some violations to frameworks constraints. However. there are some violations a tool can find. For example. applications override some methods that are not supposed to be overridden.

- Code generation

  A considerable proportion of work in framework-based development is to derive subclasses and implement hook methods. A tool can help generate the skeleton code for the classes and methods. Microsoft Visual C++ is a good example of such a tool.

Of course. at the final stage. a framework itself becomes a tool. an application generator. However. before we achieve that. tools will help a lot when reusing white-box framework.

As Brooks argued in his essay [Bro95]. there is no silver bullet for software development. There is no exception in framework technology. Although methods are needed in framework development and instantiation. no single approach can attack all problems. As we have seen. solutions are often domain specific. or even framework specific.

Chapter 7 Conclusion

In this thesis we described our work, the framework-based implementation of an optimizer for a relational database. In addition, more general issues in framework-based development were discussed.

OPT++, the framework we instantiated, is a well-designed framework, according to our instantiation experience. On the whole, the three-component decomposition conforms to the nature of query optimization. Especially, logical operators, execution algorithms and corresponding operator trees and algorithm trees are well represented. The properties of operator trees and algorithm trees are well generalized. In these classes, neither over-feature nor under-feature was found. The relationships among the classes are well defined. We did not find any defects in this part of the framework. Dynamic pruning mechanism is elaborately designed, and all special cases are processed properly. However, separation between the Search Space and Search Strategy component does not fit all applications. In our implementation, we reassigned the functionality of the two components in part. Furthermore, as Kabra claimed, the framework user has to balance between efficiency and extensibility of the target optimizer. We made a compromise between the two. Namely, we focused on efficiency, but kept the optimizer extensible wherever possible.

Stemming from a pioneer relational database system, PostgreSQL provides a good model optimizer to study. It supports SQL92/99 standard. Typical logical operators and execution algorithms are implemented in it. Some transformative rules are applied to preprocess a query tree to make subsequent optimization to be performed more conveniently. Other rules serve the goal of improving optimization without system statistics. Its search strategy is a sequential approach on the whole. The operators involved in a query are considered in a fixed order. Most of optimization effort is paid on joins. For joins, constrained dynamic programming is employed to search join space if there are not too many items to be joined; otherwise, genetic algorithm is adopted to do randomized search guided by some heuristic. The approach excludes some regions in search space that must not contain the optimum, captures and solves the key problem. In this way, optimization efficiency is improved.

In our application. typical operators for relational algebra are supported. Main transformative rules extracted from the PostgreSQL optimizer are applied. A PostgreSQL-like search strategy is implemented. Constrained dynamic programming and genetic algorithm are incorporated to optimize joins. Additionally, we modified the framework to fit sub-queries and explicit joins. In the course of the framework instantiation, we found planned customization is fairly routine. The customization of the Algebra component is in this case. However. unplanned customization needs more consideration.

Understanding a framework is a key problem in framework-based development. Having studied existing documentation techniques. we propose some suggestions for the issue from the perspective of a framework user. The suggestions include: providing reasonable degree of abstraction. adopting good programming style. using de facto standards, offering sample data. and using debugging tools. The solution to unplanned customization tends to be case specific. We give some general suggestions on the problem. Tools are developed along with the proposal of methodologies in the field. However, the tools are associated with specific methodologies. or specific frameworks On the basis of our experience. we described basic requirements for tools supporting framework instantiation.

Frameworks do improve productivity. It would be impossible to finish an optimizer in master's thesis work without the framework. A white-box framework, although requiring much effort to understand its internals. can be reused in analysis, design and code. Our experience also indicates that structural artifacts, e.g., the Algebra component in OPT++, tend to have higher reusability. Framework-based development has its own characteristics. It is a consensus that we need corresponding methods for this kind of development. A good many methods for framework development and instantiation are being investigated, but studies show that the solutions are apt to be domain specific, or framework-specific. The issues in the field. including the ones we discussed. deserve more study.

Bibliography

[Boo99] Grady Booch. James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley. 1999.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 2nd edition*. Addison-Wesley & Benjamin Cummings, Jr., 1995.

[Bus93] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl. *Pattern-Oriented Software Architecture*. Draft copy, Siemens, AG, 1993

[But02] G. Butler, L. Chen. X. D. Chen, A. Gaffar, J. M. Li, L. G. Xu. *The Know-It-All Project: A case study in framework development and evolution, Domain Oriented Systems Development: Perspectives and Practices*. Kiyoshi Itoh, Satoshi Kumagai (eds), Taylor & Francis, UK. 2002.

[But00] G. Butler, R.K. Keller, H. Mili. *A framework for framework documentation.* ACM Computing Surveys 32.1 (March 2000) electronic symposium.

[Fay99] M. Fayad, D. Schmidt. and R. Johnson (eds). *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley and Sons, New York. September 1999.

[Fon00] M. F. Fontoura, C. Braga. L. Moura, and C. J. Lucena. *Using Domain Specific Languages to* Instantiate *Object-Oriented Frameworks.* IEEE Proceedings - Software, 147(4). 109-116, 2000.

[Fon02] Marcus Fontoura, Wolfgang Pree, Bernhard Rumpe. *The UML Profile For Framework Architectures*. Addison-Wesley, 2002.

[Fre87] J.C. Freytag. *A Rule-Based View of Query Optimization.* In Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco (1987) 173-180.

[Fro00] G. Froehlich, H. J. Hoover, P. G. Sorenson. *Choosing an Object-Oriented Domain Framework*. ACM Press, New York, NY, USA. Article No. 17 Periodical-Issue-Article, 2000.

[Fro97] G. Froehlich, H. J. Hoover, Liu, L. and P. G. Sorenson. *Reusing Application Frameworks Through Hooks*. ACM, Special Issue on Object-Oriented Application

Frameworks, Vol. 40, No. 10, October 1997.

[Fro97] G. Froehlich, H. J. Hoover, Liu, L. and P. G. Sorenson. *Hooking into Object-Oriented Application Frameworks*. Proceedings of the 1997 International Conference on Software Engineering.

[Fro98] G. Froehlich, H. J. Hoover, Liu, L. and P. G. Sorenson. *Requirements for a Hooks Tool*.1998. http://www.cs.ualberta.ca/~softeng/papers/ssr04.pdf

[Fro99] G. Froehlich, H. J. Hoover, Liu, L. and P. G. Sorenson. *Using Object-Oriented Frameworks*. In Handbook of Object Technology, S. Zamir (ed.), CRC Press, New York, 1999: 26-1:26-22.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gar00] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice-Hall, 2000.

[Gol84] Adele Goldberg. *Smalltalk-80: The interface Programming Environment*. Addison-Wesley, Reading, Massachusetts. 1984.

[IBM02] IBM, *SanFrancisco Common Business Objects User Guide*. http://www-3.ibm.com/software/ad/sanfrancisco/education.html

[Joh88] Ralph Johnson and Brian Foote. *Designing Reusable Classes*. Journal of Object-Oriented Programming.1988.

[Kab99] Navin Kabra and David J. Dewitt. *OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization*. VLDB Journal, vol 8, no.1, pp. 55-78, May, 1999.

[Lee88] Mavis K. Lee, Johann Christoph Freytag, and Guy M. Lohman. *Implementing an Interpreter for Functional Rules in a Query Optimizer*. Proceedings of the 14th VLDB Conference, Los Angeles, California. pp. 55-78, 1988.

[Li01] Jinmiao Li, *An Object-Oriented Framework For Extensible Query Optimization*, Master Thesis, Concordia University, 2001.

[Mat96] M. Mattsson, *Object-oriented Frameworks - A Survey of Methodological Issues*,

Licentiate Thesis, Department of Computer Science, Lund University, 1996.

[Mic96] Zbigniew Michalewicz. *Genetic Algorithms+Data Structures= Evolution Programs*. Springer, 1996.

[Oli02] Toacy C. Oliveira, Paulo S. C. Alencar, and Donald D. Cowan. *Towards a Declarative Approach to Framework Instantiation*, http://www.cs.ubc.ca/~kdvolder/Workshops/ASE2002/DMP/papers/07oliveira-et-al.pdf

[Pir92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. *Extensible/rule based query rewrite optimization in Starburst*. In Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 39–48, 1992.

[Pos02a] *PostgreSQL 7.2.1 Documentation*. http://www.postgresql.org/idocs/

[Pos02b] *PostgreSQL 7.2.1 Source Code*. http://www.postgresql.org/

[Pre94] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[Shu00] F. Shull, F. Lanubile, and V. Basili. *Investigating Reading Techniques for Framework Learning*. IEEE, (Vol. 26, No. 11) pp. 1101-1118, November 2000.

[Wei88] A. Weinand, E. Gamma, R. Marty. *ET++ - An Object-Oriented Application Framework in C++*. Proceedings of the OOPSLA'88, pp. 46 – 57, 1988.