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.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

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

AN OBJECT-ORIENTED FRAMEWORK FOR EXTENSIBLE QUERY OPTIMIZATION

JINMIAO LI

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

JUNE 2001

Abstract

An Object-Oriented Framework for Extensible Query Optimization

Jinmiao Li

Query optimization is one of the important components of a Database Management System that directly affect the performance that the user sees.

The query, written in a language like SQL, is parsed and turned into a parse tree representing the structure of the query. The parse tree is then transformed into an expression tree of relational algebra called a logical query plan. The logical query plan is turned into a set of physical query plans by selecting algorithms to implement each operator in it, and by selecting an order of execution for these operators. The execution costs for these physical query plans are then evaluated. The one that has least cost and that represents the complete query is selected as the optimal plan and is used to execute the query.

Although query optimization has been studied for decades, building a query optimization system is still a "black art". Problems with existing query optimization frameworks are that either the addition of new query operators/algorithms is fixed or the search strategy that is used to explore the optimal plan is fixed with respect to the query algebra. There exists some improvement that uses object-oriented techniques to achieve both flexibility and extensibility in query optimization, but this improvement is very limited because the techniques that are used are limited.

This thesis proposes a reusable architecture for extensible query optimization. A query optimization system is physically divided into three major components. The framework is designed to span across these components, where each contributes to a single purpose in the system. Design patterns and object-oriented techniques are used to de-couple these components and improve the flexibility within each component. This thesis also makes a clear separation of the search strategy and the search tree. This separation conforms to a design convention, that is, to separate interface from implementation. We believe this separation promotes the reusability of the system and is good for clarity. Also, we define a search strategy interface that allows different

search strategies to be easily installed and be used interchangeably. Switching from one search strategy to another only requires modification of two lines of code within the same component. The design is further verified in light of implementation in C++. Moreover, we believe the documentation of a framework is as important as its design. We attempt to provide a series of framework documents to assist the application developer to better reuse it. These documents include the framework overview, the framework design, examples of customization, and the framework cookbook.

Acknowledgements

First of all, I would like to express my warmest thanks to my thesis supervisor, Dr. Gregory Butler, for his patience and invaluable guidance. I really appreciate his kindheartedness and humanity, and all timely and inspiring discussions he had with me. I am grateful for his first course in software design, COMP647 (Software Design Methodologies). Without it, this thesis would have been impossible to achieve.

There are many people who deserve my gratitude. First, I am thankful to all the professors who shared their knowledge with me. Special thanks to Dr. V.S.Alagar, Dr. Jaroslav Opatrny, Dr. T.Radhakrishnan, Dr. Khalid J. Siddiqui, and Mr. Aiman Hanna. I am forever grateful to them for their inspiration, encouragement, and help. I also want to thank Dr. Peter Grogono and Dr. Joey Paquet for their time and their patience to read this thesis, and giving me valuable comments. In addition, I would like to thank all fellow students who graciously helped me, especially those who are in Dr. Butler's software engineering research group.

I gratefully acknowledge Concordia University for offering me a Concordia Graduate Fellowship, Alcatel Canada Inc. for offering me a scholarship for my last academic semester at Concordia, and the Quebec government for all financial support I received during my studies at Concordia.

I dearly thank my parents and my parents-in-law, who, despite being on the other side of the globe, keep encouraging me to achieve this challenging work. Finally, I deeply thank my husband, Yun Mai, whose guidance, support, encouragement, and love help me go through my graduate studies at Concordia University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Query optimization has existed as a research subject for decades. But developing, extending, and modifying a query optimization system still remains a difficult task. Most existing query optimization systems have two major drawbacks:

- Those that allow easy addition of database operators and algorithms often have a fixed search strategy that limits the search space in which the "best" query plan can be found.
- Those that allow extensible search strategies often have a fixed query algebra.

OPT++ [20] is a query optimization framework written in C++. It is developed by the University of Wisconsin, U.S.A. It uses object-oriented design to simplify the task of developing, extending, and modifying a query optimization system. First, it allows new operators/algorithms to be easily added. Second, it offers a choice of different search strategies so that various heuristics can be experimented to limit or extend the search space explored. Third, the flexibility in both the query algebra and the search strategy can be achieved without compromising the efficiency of the system.

Most of our thesis work is inspired by OPT++.

1.1 Related Work and the Problem

Query optimization is one of the important components in a DBMS system. It takes the parsed representation of a query written in a given query language as inputs, and

outputs the best access plan that will be used to execute the query. Its responsibility is to identify an efficient access plan for evaluating a query. It generates alternative plans, estimates the execution costs for them, and chooses the best one (the one with least cost) as the result of query optimization.

Extensible query optimization systems proposed in the literature mainly fall into two categories:

1. The traditional extensible query optimization system which is characterized by either

   - Offer a fixed search strategy while the addition of new operators and algorithms is easy, or
   - Allow extensible control over the search strategy.

2. The "pure" extensible query optimization system that uses object-oriented design to achieve a flexible search strategy, and for any search strategy, the addition of new operators and algorithms is easy.

A query optimization system that offers a fixed search strategy is always a rule-based system. It often implements a rule rewriter to perform equivalent transformation on the query expressions. Representatives of this kind of system include: the System-R style Optimizer [15], Starburst project [16] [31], the Exodus Optimizer Generator [7], and the Volcano Optimizer Generator [8]. The system-R style Optimizer designs sets of rules to translate a query into a physical plan. One set of them is to convert the query into an algebraic tree. Other sets are used to generate access paths, join orders, and join methods. The optimizer developed in the Starburst project first uses a set of production rules to transform the query heuristically into equivalent queries hopefully one has less execution cost. A set of grammar-like production rules is then applied to construct physical query plans in a bottom-up fashion. Execution costs for these physical query plans are estimated and the sub-optimal plans are pruned out. Optimizers generated by Exodus and Volcano first use algebraic transformation rules to generate all possible operator trees that represent the input query. They then use implementation rules to generate physical query plans according to these operator trees.

The concept of a traditional query optimization system that allows extensible control over the search strategy is to divide a query optimization system into regions, which are responsible for different parts of optimization. In this system, the query is passed through these various regions to be optimized. Some examples of these systems include: Mitchell's region-based optimizer [9], Sciore's modular optimizer [4], and Kemper's blackboard architecture [1]. The region-based optimizer creates a hierarchy of regions. The parent region dynamically controls the sequence of regions that the query is passed through. Instead, in the blackboard approach, knowledge sources are responsible for controlling the path in which the query is passed through various regions.

A "pure" extensible query optimization system uses object-oriented design to achieve the goal that both the search strategy and the query algebra are extensible. Lanzelotte [25] describes an object-oriented design for extending the search strategy in a query optimization system. Kabra inherits the design of the search strategy from Lanzelotte in his OPT++ [20]. But OPT++ differs from that described by Lanzelotte in the modeling of the query algebra and the search space. In particular, OPT++ makes a clear separation between the logical algebra (operator tree) and the physical algebra (access plan). Kabra argues that this separation is necessary for efficiency of a query optimization system as well as for clarity and extensibility.

OPT++ has made significant improvements in the design of an extensible query optimization system [20]:

- Like the Volcano Optimizer Generator and the Starburst optimizer, OPT++ incorporates extensible specification of logical algebra operators, execution algorithms, logical and physical properties, and selectivity and cost estimation functions. Interesting physical properties, input constraints for execution algorithms and enforcers ("glue" operators) are also supported.

- The search strategies that are used in Exodus and Volcano are both built into OPT++. In other words, OPT++ offers a choice of search strategies that allow the optimizer-implementor to experiment with different search strategies and find the one that is best suited for that database system or even mix the search strategies if needed.

- By making very specific assumptions about the kinds of manipulations that are

allowed on the operator trees and access plans, OPT++ is able to put a lot of functionality of query optimization into the part of the code that does not depend upon the specific query algebra.

- The flexibility that OPT++ provides is achieved without compromising the efficiency of the optimizer.

The biggest advantage of OPT++ is that it accommodates the existing query optimization systems and makes good use of the object-oriented techniques to achieve great extensibility in both the search strategy and the query algebra. In spite of the fact that OPT++ is very extensible, it has the following drawbacks:

1. It only partially implements the design. That is, most design ideas are theoretical and have not been verified in light of implementation. For instance, the division of three components in the query optimization system is not implemented.

2. Poor partition of the framework. It puts the whole framework in the Search Strategy component and thus blurs the responsibility distribution of the system.

3. The design of the system is extensible, but not reusable. For instance, different components are strongly coupled by implementation details. It also violates information encapsulation by exposing many data structures and implementation details.

4. It is equipped with a few search strategies that are ready for use, but switching from one to another may require all the customization code in the Search Space component to be changed.

5. Framework documentation is insufficient.

1.2 Our Work

The first step to develop a framework for extensible query optimization is to gain enough understanding of its context. To better understand the mechanism of extensible query optimization, we prototyped a database that included SampleOpt [19], a

customized example of OPT++. The incorporation of a query optimization system into the prototype DBMS involved:

1. Defining an interface to wrap the query optimization system.
2. Customizing the system catalog.
3. Defining an interface for the user-defined query parser and the optimization system.
4. Defining and implementing a mechanism that translated the optimal plan into a self executable plan.

The second step was to develop a reusable extensible query optimization system. Our design work started from the modeling of system requirements, which included:

1. Use case analysis — To capture the intended behaviors of extensible query optimization.
2. Use case diagram — To model the system in its context.

Following the use case modeling was the architectural design, which mainly included:

1. Partitioning the system.
2. Defining distribution of responsibility.
3. Designing class structure.
4. Defining relationships.
5. Identifying the critical problems and providing solutions to them.

Next came the detailed design which included:

1. Defining data structure.
2. Implementing the algorithms for methods.
3. Implementing the relationships.

The last thing to do was to turn the detailed design into programs that make up the final product: an extensible query optimization system.

After we had gained sufficient knowledge about extensible query optimization and had developed such an application, we began to consider evolution of this application into a framework. In the framework design, we did the following:

1. Identifying hot spots [30].
2. Defining protocols for customization.
3. Writing framework documentation.

Throughout this thesis work, we

- Adhered to the UML for object-oriented analysis and modeling, and effectively applied the UML modeling techniques. We believe that the right models and modeling techniques can clearly illustrate the most challenging design problems and offer insights to solutions that address them.
- Kept simplifying the system design because a simple design was easy to understand, easy to implement, and thus easy to complete.
- Reused successful experience such as design patterns and the partial design and implementation of OPT++.

1.3 Contribution of the Thesis

The principal contribution of this thesis is that it proposes a reusable architecture for extensible query optimization. In this architecture, a query optimization system is physically divided into three major components. The query optimization framework we are building is designed to span across these components, where each component contributes to a single purpose in the system. Design patterns and object-oriented techniques are used to de-couple these components and improve the flexibility within each component.

This thesis also makes a clear separation of the search strategy and the search tree. This separation conforms to a design convention: separate interface from implementation. We believe this separation promotes the reusability of the system and

is good for clarity. Also, we define a search strategy interface that allows different search strategies to be easily installed and be used interchangeably. Switching from one search strategy to another only requires modification of two lines of code within the same component.

In order to prove the feasibility of the software design for the query optimization framework, a query optimization system was built for relational database with some extensions to object-relational constructs such as set-valued attributes and reference-valued attributes. Our experience shows that the software design for this system is correct and simple enough to be implemented.

Moreover, we believe the documentation of a framework is as important as its design. We attempt to provide a series of framework documents to assist the application developer to better reuse it. These documents include:

- A framework overview that describes the context of this framework and its problem, solution, and consequences.
- A framework design that describes the architectural and detailed design of the framework.
- Examples of customization that illustrate the typical uses of this framework.
- A framework cookbook that provides a general guideline for customization.

1.4 Layout of the Thesis

Chapter 2 provides a detailed description of the background that includes object-oriented design, UML modeling, patterns and frameworks, query optimization basics, and OPT++. We hope the background information is rich enough to lead to a better understanding of this thesis.

Chapter 3 is the framework overview. Although it is an informal description of the query optimization framework in short, it is expressed in a structure that is easy to follow. The framework overview describes the problem, the context, the solution, and the consequences.

Chapter 4 presents the framework design, which includes the overall structure, design of each component, dynamic behavior, and cost evaluation. The reason we

put cost evaluation in a separate section is that it relates to the most critical problem of query optimization and deserves some special treatment.

Chapter 5 shows examples of customization. These examples are arranged from simple through to advanced. They reflect the general uses of the query optimization framework.

Chapter 6 is the framework cookbook. It is a customization summary of the query optimization framework and provides general guidelines for the application developer to reuse it.

Chapter 7 concludes this thesis with an outline of its contributions and gives suggestions for future work.

We assume the reader is familiar with the C++ programming language [2].

Chapter 2

Background

2.1 Object-Oriented Design

2.1.1 What Is Object-Oriented Design

Software design is creative work. In the most general sense, it is an abstract description of how something is built. The process of design is a process of making decisions that involves finding a set of potential solutions to the problem, making tradeoffs between various alternatives, analyzing their strengths and weaknesses, and making decision to select the "best" one that address the problem precisely.

Object-oriented design is a software design approach that concerns with developing an object-oriented model of a software system to implement the identified requirements. It differs from the functional approach in that it views a software system as a community of objects, each has its own state and behaviors. Objects interact each other comprising the global solution space for the problem.

The basic concept in the object-oriented world is the object. Object-oriented design describes a solution to a problem in terms of objects. Sommerville gives the object a definition [14]:

> An object is an entity that has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

A group of objects with the same state attributes and operations is called a class.

Objects are created according to the class definition, which is a template for its objects. The class definition includes declarations of all the attributes and services which should be associated with an object of that class.

Some basic characteristics of an object-oriented design are [14], [29]:

- Everything is an object. Objects are abstractions of real-world or system entities which are responsible for managing their own private state and offering services to other objects.

- Objects are independent entities that may readily be changed because state and representation information is held within the object. Changes to this representation may be made without reference to other system objects.

- System functionality is expressed in terms of operations or services associated with each object. Computation is performed by objects communicating with each other, requesting that other objects perform actions rather than share variables. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary to complete the task.

2.1.2 Inheritance and Polymorphism

We will discuss two important concepts in object-oriented design: Inheritance and polymorphism. The intuitive meaning of inheritance is to organize the classes of a system into a hierarchy. Child classes have all the properties of their parent class in addition to the properties defined in themselves. Budd [29] describes two explicit benefits of inheritance:

- Software Reusability and Code Sharing
  When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten. That means, these functions can be written once and reused. Benefits of reusable code also include increased reliability and the decreased maintenance cost.

- Consistency of Interface
  When two or more classes inherit from the same superclass, we are assured that the behavior they inherit will be the same in all cases.

One important benefit of inheritance is that it provides programmers with the ability to construct reusable software components. The concept of software component promotes information hiding and makes rapid prototyping easy. On one hand, a programmer who reuses a software component only needs to understand the nature of the component and its interface. It is not necessary for the programmer to have detailed information concerning matters such as the techniques used to implement the component. On the other hand, when a software system is constructed largely out of reusable components, development time can be concentrated on understanding the new and unusual portion of the system. Thus, a prototype system can be quickly developed and experimented.

Despite the great benefits of inheritance, it is expensive because: the size of the system code is increased, the system becomes more complex because in order to understand the behavior of a class we need to examine that of its ancestors, and the system execution speed is decreased. But in today's software design, when the hardware setting is no longer a major concern and the problems of maintaining and evolving the legacy systems loom large, the benefits that the inheritance brings to us dominate.

Polymorphism is another concept of importance in the object-oriented design that closely relates to inheritance. The term polymorphic means "many forms" in its Greek roots. The major meaning of polymorphism in the object-oriented design is that a name is allowed to refer to a value of its declared type or any other subtypes of its declared type. So, depending on the type of this object, an operation on it can demonstrate distinguishable behavior.

Inheritance and polymorphism make software reuse in the object-oriented field range from possible to even easy. Design pattern and framework are two important routes to software reuse. Please refer to Section 2.3.1 for details.

2.1.3 Evaluation Criteria for Object-Oriented Design

Reiss [26] summarizes the most important evaluation criteria for an object-oriented design:

- Correctness

  A design must be correct. It must address and solve all the issues brought up

in the problem definition, and the solution it presents must actually solve the given problem within the specified constraints.

- Simplicity

  Classes, methods, and the overall design should all be as simple as possible.

- Cohesion and Good Coupling

  Everything in a class should be directed at a single purpose. Classes serving more than one purpose are generally better split into two simpler classes. Operations should serve a single purpose. Classes should never depend on knowing the internal implementations of other classes. All relationships among classes should be based on operations, not on data elements. The number of operations one class provides for another (and the number of parameters in those operations) should be minimized.

- Information Hiding

  Information hiding enhances simplicity by concealing implementation details within a particular class or method and separating those details from other classes. It also provides for risk management since any aspects of the system in which change might be required can be hidden inside a particular class or method so that later changes affect only that class and not the remainder of the system.

- Error Handing

  Error handing, to be done correctly, must be thought about as part of the initial problem statement and should be an important design criterion in considering alternative designs.

In summary, a good object-oriented design is not only a correct design. It should be as simple as possible. Meantime, it promotes cohesion, good coupling, and information hiding, and it will consider the error handing mechanism in the early design phase.

2.2 The Unified Modeling Language

In general, text and pictures are used to describe a software design. Informal descriptions are fine when one is doing an individual design. But problems come when a joint project is carried out by a group of designers. The design done by one needs to be understood by another, maybe many years after the original design was done. In this case, it is important to have standard design notations with fixed meanings that everyone can understand and agree to.

Standard design notations are used to express the design work, right after the problem is well understood and the design for the solution is well known. They may not highlight the wicked problems that may exist in the problem domain. They may not help the designer to increasingly comprehend the complexity of the system in its entirety. In order to expose the weakest problem in a system and help us to better understand the system we are building, we need a modeling language.

Unified Modeling Language (UML) is such a modeling language that is best to express an object-oriented design with standard design notations. Most definitions in this section are taken from [10].

2.2.1 Conceptual Model of the UML

To understand the UML, we need to form a conceptual model of the language. The vocabulary of the UML encompasses three kinds of building blocks: things, relationships, and diagrams. Things are abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

2.2.1.1 Things in the UML

There are four kinds of things in the UML: Structural things, Behavioral things, Grouping things, and Annotational things.

Structural Things Structural things are the nouns of UML models, representing elements that are either conceptual or physical. In the UML, basic structural things include class, interface, collaboration, use case, active class, component, and node.

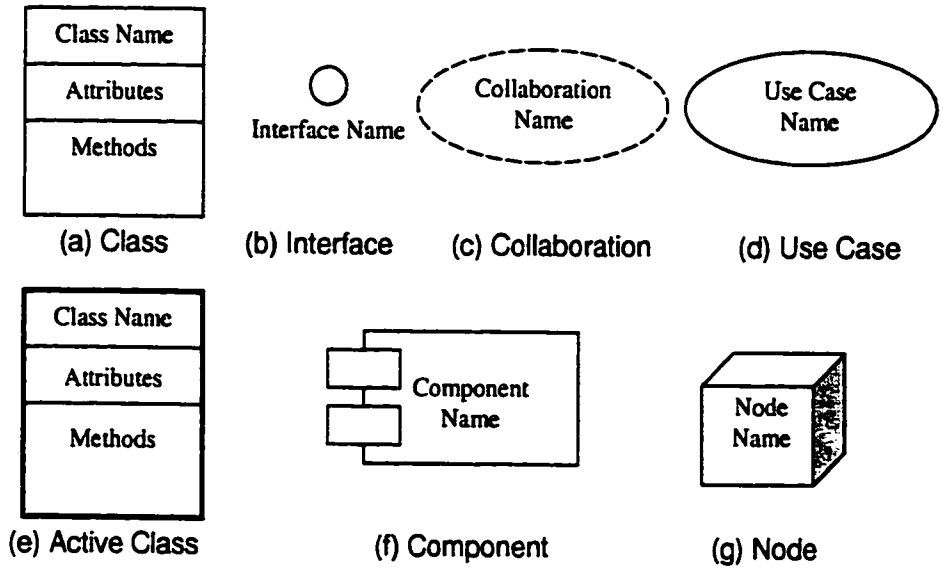

Figure 1: Structural Things in the UML

Class — a template of a set of objects. It reflects the same concept in the object-oriented design. In the UML, it is rendered as a rectangle (see Figure 1(a)).

Interface — a collection of operations that specify a service of a class or component. It describes the observable behavior or data structure for that element. In the UML, it is rendered as a circle (see Figure 1(b)).

Collaboration — an interaction and a society of roles and other elements that work together to provide some cooperative behavior that is bigger than the sum of all the elements. In the UML, it is rendered as an ellipse with dashed lines (see Figure 1(c)).

Use case — a description of set of sequence actions that a system performs that yields an observable result of value to a particular actor. It is realized by a collaboration. In the UML, it is rendered as an ellipse with solid lines (see Figure 1(d)).

Active class — a class whose objects own one or more processes or threads and therefore can initiate control activity. It looks like a class except that its objects represent elements whose behavior is concurrent with other elements. In the UML, it is rendered as a rectangle with heavy

lines (see Figure 1(e)).

Component — a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Examples of components are Java beans, COM+, etc. In the UML, a component is rendered as a rectangle with tabs(see Figure 1(f)).

Node — a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often processing capability. In the UML, it is rendered as a cube (see Figure 1(g)).

Behavioral Things Behavioral things are verbs of a model, representing behavior over time and space. There are primarily two kinds behavioral things: interaction and state machine.

(a) Interaction (b) State Machine

Figure 2: Behavioral Things in the UML

Interaction — a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of messages, action sequences (the behavior invoked by a message), and links (the connection between objects). In the UML, a message is rendered as a directed line (see Figure 2(a)).

State machine — a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state machine. A state machine involves a number of states,

transitions (the flow from state to state), events (things that trigger a transition) and activities (the response to a transition). In the UML, a state is rendered as a rounded rectangle (see Figure 2(b)).

Grouping Things Grouping things are the organizational parts of the UML. The primary grouping things in the UML are packages.

Figure 3: Grouping Things in the UML

Package — a general purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. A package in the UML is purely conceptual, that means it exists only at development time. In the UML, it is rendered as a tabbed folder (see Figure 3).

Annotational Things Annotational things are the explanatory parts of UML models. The primary annotational thing in the UML is a note, which is a symbol for constraints and comments attached to an element or a collection of elements. In the UML, a note is rendered as a rectangle with a folded corner (see Figure 4).

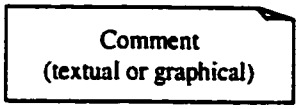

Figure 4: Annotational Things in the UML

2.2.1.2 Relationships in the UML

There are four kinds of relationships in the UML: Dependency, Association, Generation. and Realization

Dependency — a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics

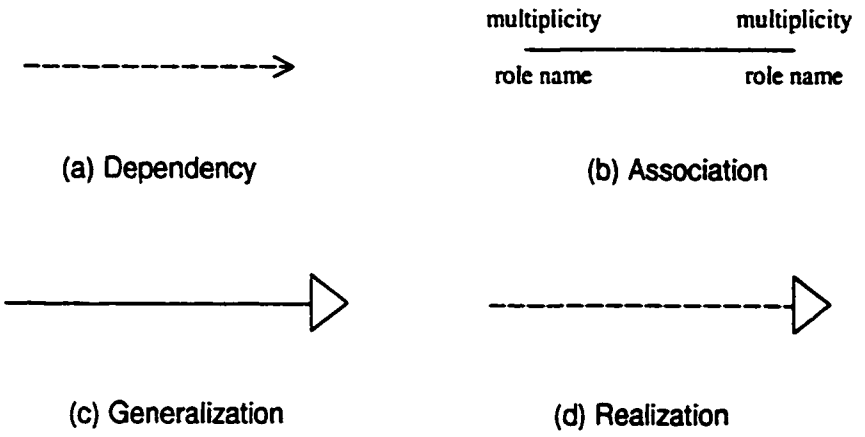

Figure 5: Relationships in the UML

of the other thing (the dependent thing). In the UML, it is rendered as a dashed line (see Figure 5(a)).

Association — a structural relationship that describes a set of links, each is a connection among objects. Aggregation is a special kind of association representing a structural relationship between a whole and its parts. In the UML, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments such as multiplicity and role names (see Figure 5(b)).

Generalization — a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In the UML, the generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent (see Figure 5(c)).

Realization — a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. Realization relationships are used either between interfaces and the classes/components that realize them or between use cases and the collaborations that realize them. In the UML, a realization relationship is rendered as a cross between a generalization and a dependency relationship (see Figure 5(d)).

2.2.1.3 Diagrams in the UML

A diagram is the graphical presentation of a set of elements. It is a projection onto a system visualizing this system from one specific perspective. In the UML, there are nine diagrams: Class diagram, Object diagram, Use case diagram, Sequence diagram, Collaboration diagram, Statechart diagram, Activity diagram, Component diagram, and Deployment diagram.

Class diagram — the most commonly used diagram in the UML to address the static design view of a system. It consists of a set of classes, interfaces, and collaborations and their relationships.

Object diagram — static snapshots of instances of the things found in class diagrams. It consists of a set of objects and their relationships, and is used to address the static design view or static process view of a system from the perspective of real or prototypical cases.

Use case diagram — important diagram in organizing and modeling the behaviors of a system to address the static use case view of the system. It consists of a set of use cases and actors (a special kind of class) and their relationships.

Sequence diagram — most commonly used diagram for interaction of a set of objects with emphasis on the time-ordering of messages to address the dynamic view of a system.

Collaboration diagram — important diagram to address the dynamic view of a system that shows interaction of a set of objects with emphasis on the structural organization of the objects that send and receive messages.

Statechart diagram — important diagram to address the dynamic view of a system especially in modeling the behavior of an interface, class, or collaboration and with emphasis on the event-ordered behavior of an object which is especially useful in modeling reactive systems. It consists of states, transitions, events, and activities.

Activity diagram — a special kind of a statechart diagram with emphasis on modeling the function of a system and the flow of control among objects.

Component diagram — a diagram that addresses the static implementation view of a system and shows the organizations and dependencies among a set of components.

Deployment diagram — a diagram that addresses the static deployment view of an architecture and shows the configuration of run-time processing nodes and the components that live on them.

2.2.2 Use the Right Modeling Techniques

A good modeling language is necessary but not sufficient for a good software design. Instead, the right modeling techniques and practical experience are important.

First, it is not mandatory to use all kinds of diagrams for a system design. In fact, it is not necessary. The role of modeling is to help the developer to better understand the problem and its solution. In many software designs, it is often to present the static view and dynamic view of the system with a small subset of the diagrams.

Second, one kind of diagram is only a projection onto a system from one perspective. It is a mistake to use only one kind of diagram throughout the system design. Alone the class diagrams or sequence diagrams do not guarantee a good understanding of a system. Class diagrams only address the static views of the system. For a better understanding of the behaviors of this system, diagrams that show the dynamic views of the system are required. Which dynamic diagrams to use depends on what kinds of applications are to be modeled. For example, in a reactive system, statechart diagrams may be used more often than the sequence diagrams or collaboration diagrams. In a general application, class diagrams are necessary to present the overall and partial structure of the system. Sequence diagrams are used more often at the beginning of the modeling. When more and more details are added later, collaboration diagrams may be used to show the interaction of a large number of objects and their complex relationships.

Third, diagrams are better to be presented together with a detailed text description. A diagram is an abstraction of the system. Despite the fact that the graphical diagrams can transcend the meanings of some texts, many details are still omitted.

A good design document is made up of a balanced number of diagrams, each accompanied by a detailed textual description of it.

2.3 Patterns and Frameworks

2.3.1 Patterns, Design Patterns, and Frameworks

Observation shows that the experts reuse successful experience over time. Patterns arise when specific problem/solution pairs are abstracted and the common factors are distilled out. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [3]. A context is a situation in which the problem occurs. "Problem" refers to a problem that arises repeatedly in the given context. "Solution" is a proven resolution of the problem.

A design pattern is a medium scale pattern. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context. Gamma et al. [5] categorizes a collection of design patterns and describes them in details. Design patterns that are used in this thesis are selected from those defined by Gamma et al.

A framework is a pattern arising at the system architectural level. It is a collection of abstract classes that provides an infrastructure common to a family of applications. A framework dictates certain roles and responsibilities amongst its classes, and specifies the standard protocols for their collaboration. Frameworks exist to support the development of a family of applications. Variability is factored out as hot spots and a simple mechanism to customize each hot spot is provided. "Hot spot" is a term coined by Pree in his book [30]. A framework is characterized by:

- Partial design;
- Incomplete implementation;
- Inversion of system control, and it contains the part of control that invokes the methods supplied by the user;
- Reuse arises in all stages of system analysis, design and implementation.

Buschmann et al. [6] summarizes some properties of patterns for software architecture:

1. A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.

2. Patterns document existing, well-proven design experience.

3. Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.

4. Patterns provide a common vocabulary and understanding for design principles.

5. Patterns are a means to documenting software architectures.

6. Patterns support the construction of software with defined properties.

7. Patterns help you build complex and heterogeneous software architectures.

8. Patterns help you to manage software complexity.

2.3.2 Developing A Framework

A framework may begin as an application that evolves to a framework, and other applications are developed to confirm the reusability of this framework before it is rolled out for general use [11].

A framework evolves over time. Uses of a framework may expose some insufficiency and incompleteness in the framework design. The framework is then refined to accommodate the new raised issues and the old ones. A framework evolves as a wider application domain is covered, hot spots [30] are more precisely identified, customization is concisely specified, and all the jargons are clearly defined.

The major steps in developing an application framework can be summarized as [23], [28]:

1. Identify and analyze the application domain and identify the framework. If the application domain is large, it should be decomposed into a set of possible frameworks that can be used to build a solution. Analyze existing software solutions to identify their commonality and the differences.

2. Identify the primary abstractions. Clarify the role and responsibility of each abstraction. Design the main communication protocols between the primary abstractions. Document them clearly and precisely.

3. Design how a user interacts with the framework. Provide concrete examples of the user interaction, and provide a main program illustrating how the abstract classes are related to each other and to the classes for user interaction.

4. Implement, test, and maintain the design.

5. Iterate with new applications in the same domain.

The design and implementation of frameworks relies heavily on abstract classes, inheritance, and polymorphism.

2.3.3 Documenting A Framework

Butler [11] summarizes various styles of documentation for a framework and provides the guidelines on how to document a framework to assist the application developers.

Examples of these framework documentation styles include:

1. Examples.

   Each example illustrates a single new hot spot [30], starting with the simplest and most common form of reuse for that hot spot, and eventually providing a complete coverage.

2. Cookbooks and Recipes

   A recipe describes how to perform a typical example of reuse during application development. The information is presented in informal natural language, perhaps with some pictures, and usually with sample source code. A cookbook is a collection of recipes. A guide to the contents of the recipes is generally provided.

3. Contracts

   A contract specifies a set of communicating participants and their contractual obligations.

4. Design Patterns

   A design pattern provides an abstraction above the level of classes and objects. It captures design experience at the micro-architecture level by specifying the relationship between classes and objects involved in a particular design problem.

5. Framework Overview

   A framework overview describes the context of this framework, defines the jargon of the domain, and delineates its scope: what is covered by this framework and what is not, as well as what is fixed and what is flexible in the framework.

6. Reference Manual

   A reference manual for an object-oriented system consists of a description of each class, together with descriptions of global variables, constants, and types.

7. Design Notebooks

   A design notebook collects together information related to the design of hardware. The information will include background theory, analysis of situations, and a discussion of engineering tradeoffs.

8. Other

   Examples of these styles may include a use case or scenarios [13] that describes the intended functionality, and a time thread [24] for a scenario can depict when and where the scenario involves the framework and when and where it involves the customized code.

The audience that Butler refers to is the application developer, who may be somewhat inexperienced as developer, or in object-oriented technology, and may be somewhat ignorant of the application domain.

Butler [11] concludes that, to document a framework to assist the application developer, we need:

First, an overview of the framework should be prepared, both as a live presentation and as the first recipe in the cookbook.

Second, a set of example applications that have been specifically designed as documentation tools is required. The examples should be graded from simple through to advanced, and should incrementally introduce one hot spot at a time. A hot spot that is very flexible may need several examples to illustrate its range of variability, from straightforward customization through to elaborate customization with all the bells and whistles.

Third, a cookbook of recipes should be written, and organized along the lines of Johnson's pattern language [22]. The recipes should use the example applications to make their discussion concrete. There will be cross-references between recipes,

recipes and source code, as well as recipes and some available documentation such as a reference manual, contracts, or design patterns.

2.4 Query Optimization and OPT++

2.4.1 Query Optimization

Query optimization is one of the important components in a DBMS system. It takes the parsed representation of a query written in a given query language as inputs, and outputs the best access plan that will be used to execute the query. Its responsibility is to identify an efficient access plan for evaluating a query. It generates alternative plans, estimates the execution costs for them, and chooses the best one (the one with least cost) as the result of query optimization.

2.4.1.1 Query Optimization in the DBMS Architecture

Ramakrishnan [21] presents a detailed view of query optimization in the DBMS architecture as shown in Figure 6.

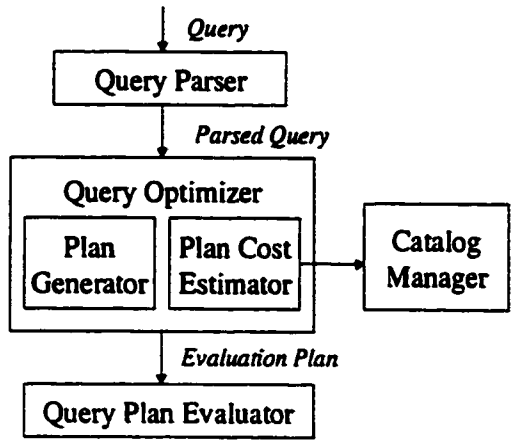

Figure 6: Query Parsing, Optimization, and Execution

The query optimizer takes the output of the query parser, the parsed query, as input, and produces an evaluation plan that will serve as the input to the query plan evaluator. It provides the functionality for both plan generation and plan cost estimation.

Query optimization involves two basic steps:

1. Enumerate alternative plans. Since the search space in which the possible plans exist is huge, typically only a subset of these plans are considered.

2. Estimate the execution cost for each plan according to a cost model, and choose the optimal one with least cost.

2.4.1.2 Basic Concepts

Basic concepts in query optimization include: Operator, Algorithm, Logical query plan, and Physical query plan.

Operator— An operation performed on the database. It is the basic element that makes up of a logical query plan tree. It is also called a logical operator. Select and Join are two frequently used logical operators in a relational database system.

Algorithm — It often refers to an implementation of an operator. For example, Filter and Nested Loop Join are two common algorithms for the operators Select and Join in the relational database. In some rare cases, an algorithm may not have a corresponding operator in the database. For examples, a Sort algorithm that enforces a sort order to its output does not have a corresponding operator. An algorithm is also called an execution algorithm or a physical operator.

Logical query plan — An algebraic expression that represents the particular operations to be performed on data and the necessary constraints regarding order of operations. It is a tree of logical operators, and is also known as an operator tree, or a logical plan.

Physical query plan — A tree of algorithms that specifies the particular order of operations and the algorithm that is used to implement each operation. It is also known as an access plan, an evaluation plan, an algorithm tree, a physical plan, or simply a plan. When enumerating possible physical plans derivable from a given logical plan, we select for each physical plan [12]:

1. An order, and grouping for associative-and-commutative operations like joins, unions, and intersections.
2. An algorithm for each operator in the logical plan, for instance, deciding whether a Nested Loop Join or a Hash Join should be used.
3. Additional operators — scanning, sorting, and so on — that are needed for the physical plans but that were not present explicitly in the logical plan.
4. The way in which arguments are passed from one operator to the next, for instance, by sorting the intermediate result on disk or by using iteration and passing an argument one main-memory buffer at a time.

We assume that a query can be logically represented as an operator tree. Because of the association and commutativity properties of some operators, a query may have several operator tree representations. Also, since an operator may have more than one implementation algorithm, an operator tree may have more than one algorithm tree.

The following is an example based on the SQL query:

select e.name, p.address **from** Persons p, Employees e **where** p.name=e.name;

There may be many operator trees that can be used to represent this query. We only show two of them in Figure 7. Note that exchanging the left and right trees will give rise to two other operator trees.

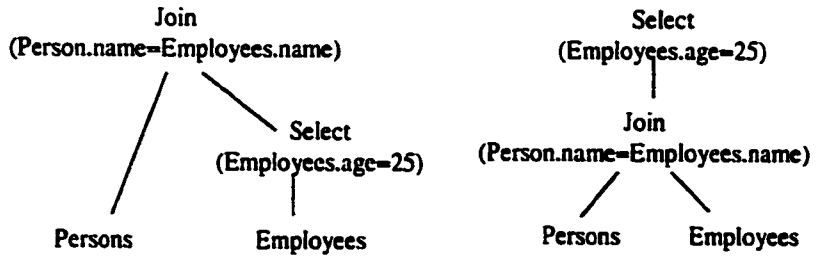

Figure 7: Operator Trees

The first operator tree is a join tree, representing a join between the relation Persons and the result of applying the Select operator to the relation Employees.

The second is a select tree, representing the Select operator is applying to the join result of the relations Persons and Employees.

In general, translating an operator tree to an algorithm tree consists of replacing each logical operator in the operator tree with one execution algorithm for that logical operator. Look at the join tree, since there exist several execution algorithms for the Join operator, each will lead to an algorithm tree. In addition, the Select operator and the relation Employees may be implemented as an Index Scan if there is an index on the attribute age for the relation Employees.

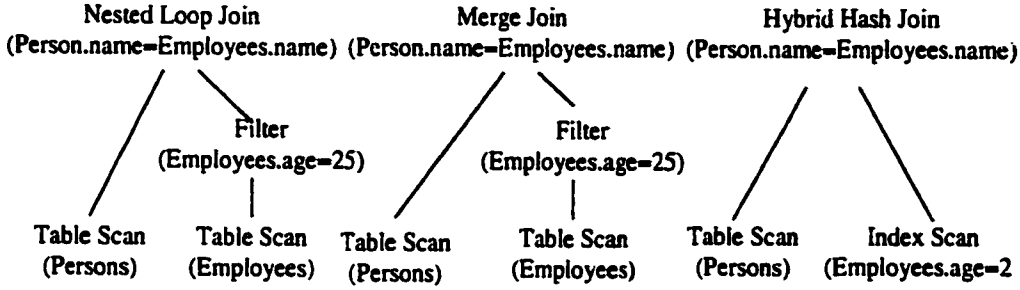

Figure 8: Algorithm Trees

Some algorithm trees for the join tree are shown in Figure 8. We assume that there are three implementation algorithms for the Join operator: Nested Loop Join, Merge Join, and Hybrid Hash Join. We also assume the relation Employees has an index on the attribute age. Compare Figure 7 to Figure 8, we can see that each relation in the operator tree is substituted by the physical operator Table Scan on that relation. Operator Select is converted to physical operator Filter. Operator Join is converted to Nested Loop Join, Merge Join, or Hybrid Hash Join. A combination of the operator Select and the relation Employees can be converted to a Table Scan followed by a Filter or just an Index Scan.

2.4.1.3 Search Strategy

A search strategy is the approach used to explore the space of all plans for the optimal plan. Specifically, it is responsible for building the logical plans, converting them to physical plans, performing cost evaluation, and finally choosing the best one that has least cost.

The baseline search strategy approach is exhaustive. In this method, all possible combinations of logical plans and physical plans are considered, execution cost is estimated for each possible physical plan, and the one with least cost is chosen at the end.

One example of the exhaustive approach is the Bottom-Up Search Strategy, which is used in the System-R style optimization. In this search strategy, only a subset of possible plans is examined. That is, only the best plans or the plans that are sub-optimal but that have some interesting properties (such as sorted, indexed) are considered when the plans for a larger expression are computed.

Figure 9 shows the algorithm of the Bottom-Up Search Strategy in an activity diagram of UML. The initial operator trees are created according to the relations involved in the query. One operator tree is picked from the operator tree repository. If it can be expanded, it is expanded to larger operator tree(s), whose corresponding access plans are also created. Execution costs for all access plans are estimated and compared to each other. Equivalent access plans that have higher costs are eliminated. The process repeats where the next operator tree in the operator tree repository is selected. If the selected operator tree can not be expanded, the next one following it in the repository is selected instead. If none of the operator trees can be expanded, the process terminates. The cheapest access plan that represents the complete query is returned as the result of the Bottom-Up search approach.

Note that the newly created operator trees that are results of tree expansion are stored in the operator tree repository for further optimization, and all operator trees in the repository are stored according to their increasingly size.

2.4.1.4 Cost Estimation

Cost estimation in query optimization determines the execution cost for each access plan. The resource that is consumed when an access plan is executed includes: CPU time, I/O cost, memory, maybe communication cost in a network environment, or a combination of them. Since the cost estimation algorithm provides the core functionality that will be executed repeatedly during the query optimization process, efficiently evaluating the costs of the access plans is of fundamental importance.

Chaudhuri [27] summarizes the basic estimation framework deriving from the System-R system developed by the IBM:

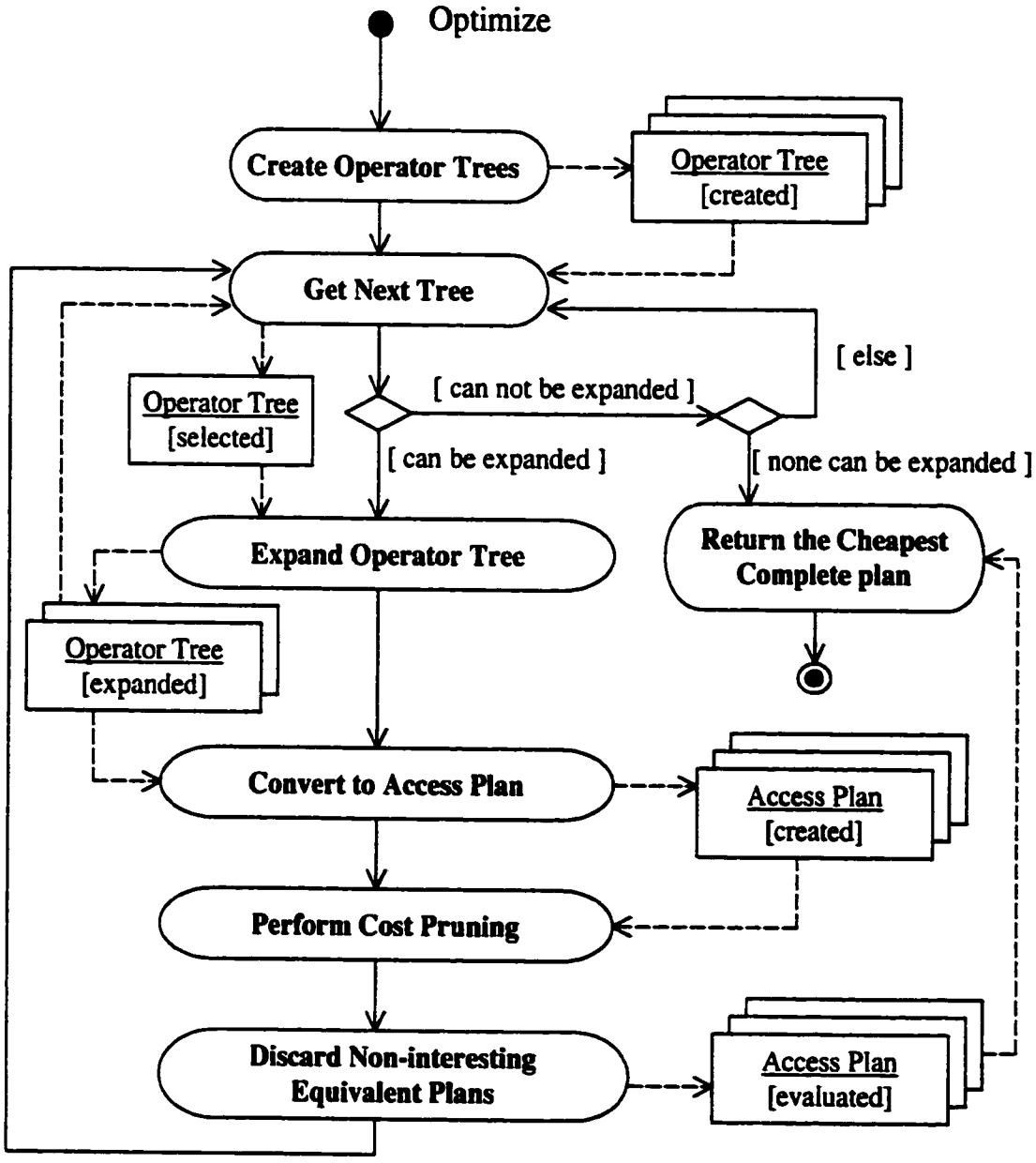

Figure 9: Bottom-Up Search Strategy

1. Collect statistical summaries of data that has been stored.

2. Given an operator and the statistical summary for each of its input data streams, determine the:

   (a) Statistical summary of the output data stream, and

   (b) Estimated cost of executing the operation.

Step 2 can be applied iteratively to an operator tree of arbitrary depth to derive the cost for each of its operators. The cost for an access plan can be computed by combining the costs of all the operator nodes in the tree.

The term "statistical summary of data" refers to the statistical property of the current logical query plan. Logical query plans for the same query or the same part of the query should have the same statistical properties. But, the access plans corresponding to the same query or the same part of the query may have various estimated execution costs. That is the reason why at the end of the search, only the access plan with the least estimated cost is selected to execute the query.

2.4.2 The OPT++ Framework

OPT++ [20] is a query optimization framework written in C++. It is developed by the University of Wisconsin, U.S.A. It uses object-oriented design to simplify the task of developing, extending, and modifying a query optimization system. First, it allows new operators/algorithms to be easily added. Second, it offers a choice of different search strategies so that various heuristics can be experimented to limit or extend the search space explored. Third, the flexibility in both the query algebra and the search strategy can be achieved without compromising the efficiency of the system.

2.4.2.1 Basic Design

Figure 10 shows the basic system design of OPT++ [20]. A query optimization system built from OPT++ consists of two parts of code: one is the code provided by the OPT++ framework, the other is the code supplied by the optimizer-implementor. OPT++ defines a search strategy component and a few abstract classes. The search strategy component defines a class hierarchy for search approaches that may be used

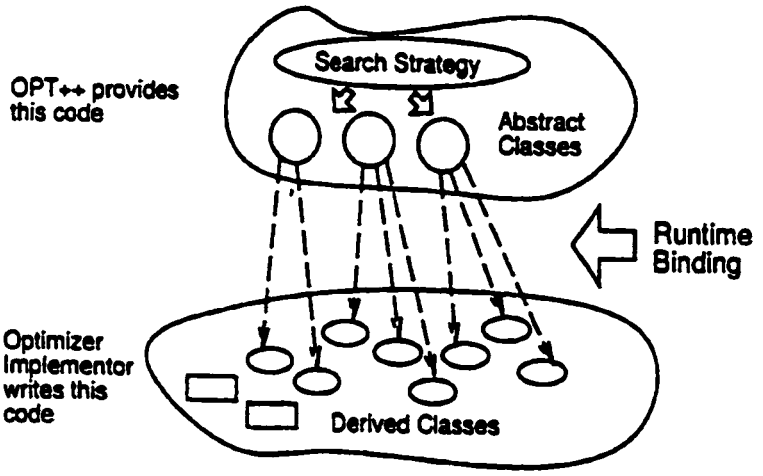

Figure 10: Basic System Design of OPT++

to explore the optimal plan in query optimization. The abstract classes define interfaces whose implementations should be supplied by the optimizer-implementor. The search strategy is written entirely in terms of these abstract classes.

The OPT++ framework reaps the benefits of inheritance and run-time binding of C++ to achieve the system's extensibility. It makes very specific assumptions about the kinds of manipulations that are allowed on the operator trees and the access plans. Abstract classes that represent interfaces of the operators in the database, their corresponding algorithms, and the operations on the operator trees and algorithm trees are all defined in the OPT++ framework. When developing a query optimization system from OPT++, the optimizer-implementor defines the actual database algebra in terms of the interfaces defined in OPT++. Algorithms for these database operators and the manipulations on the operator trees and the algorithm trees are also written according to the interfaces defined in OPT++. With the run-time binding mechanism in C++, the search strategies defined in the OPT++ framework will call the actual implementation of the operators, their algorithms, and tree/plan transformation.

2.4.2.2 Search Strategy

OPT++ claims that it is equipped with three search strategies: Bottom-Up Search Strategy, Transformative Search Strategy, and Simulated Annealing Search Strategy. But in its sample application SampleOpt [19], only the Bottom-Up Search Strategy is implemented. Please refer to Section 2.4.1.3 for detailed description of its algorithm.

Figure 11 and Figure 12 show algorithms of the Transformative Search Strategy and the Simulated Annealing Search Strategy described in the OPT++ documentation. We present them as activity diagrams of the UML.

The Transformative Search Strategy is based on tree transformation. The optimizer-implementor defines the tree transformation rules (algebraic laws). These rules are then repeatedly applied to operator trees hopefully leading to new operator trees whose access plans have less execution costs.

The Transformative Search Strategy begins with the creation of a complete operator tree that represents the complete query. We call it the initial operator tree. If one or more algebraic laws can be applied to the initial operator tree, they are applied and may lead to a few new operator trees. Corresponding access plans for the newly created operator trees are also created. Their execution costs are computed and compared to each other. Equivalent access plans that have higher costs are eliminated. The process repeats where another operator tree is selected for transformation. If no operator tree can be further transformed, the search process terminates. The access plan that has the lowest cost is returned as the result of Transformation searching.

The Simulated Annealing Search Strategy is a randomized search strategy. It defines a "temperature" variable to control the termination of search. This variable is initialized when the search begins. Similar to the Transformative Search Strategy, the first step is to create the initial operator tree, a complete operator tree that represents the complete query. Corresponding access plan of the initial operator tree is also created. We call it the current access plan. A random selection is then performed. If the option "select plan" is selected, another access plan for this operator tree is created representing a different execution solution for this operator tree. Otherwise, a tree transformation is performed on the initial operator tree. A new equivalent operator tree may be generated. A corresponding access plan for this new operator tree is also created. We call it the new access plan. The execution cost for the new access plan is computed and it is compared to that of the current access plan. The access plan that has higher cost is eliminated, and the current access plan is reset to the one that has lower cost. The "temperature" is then decreased. The process repeats for the next random selection. If the "temperature" equals to zero or there is no improvement in the execution cost for a certain number of steps, the search process ends. The current access plan is returned as the result of Simulated Annealing searching.

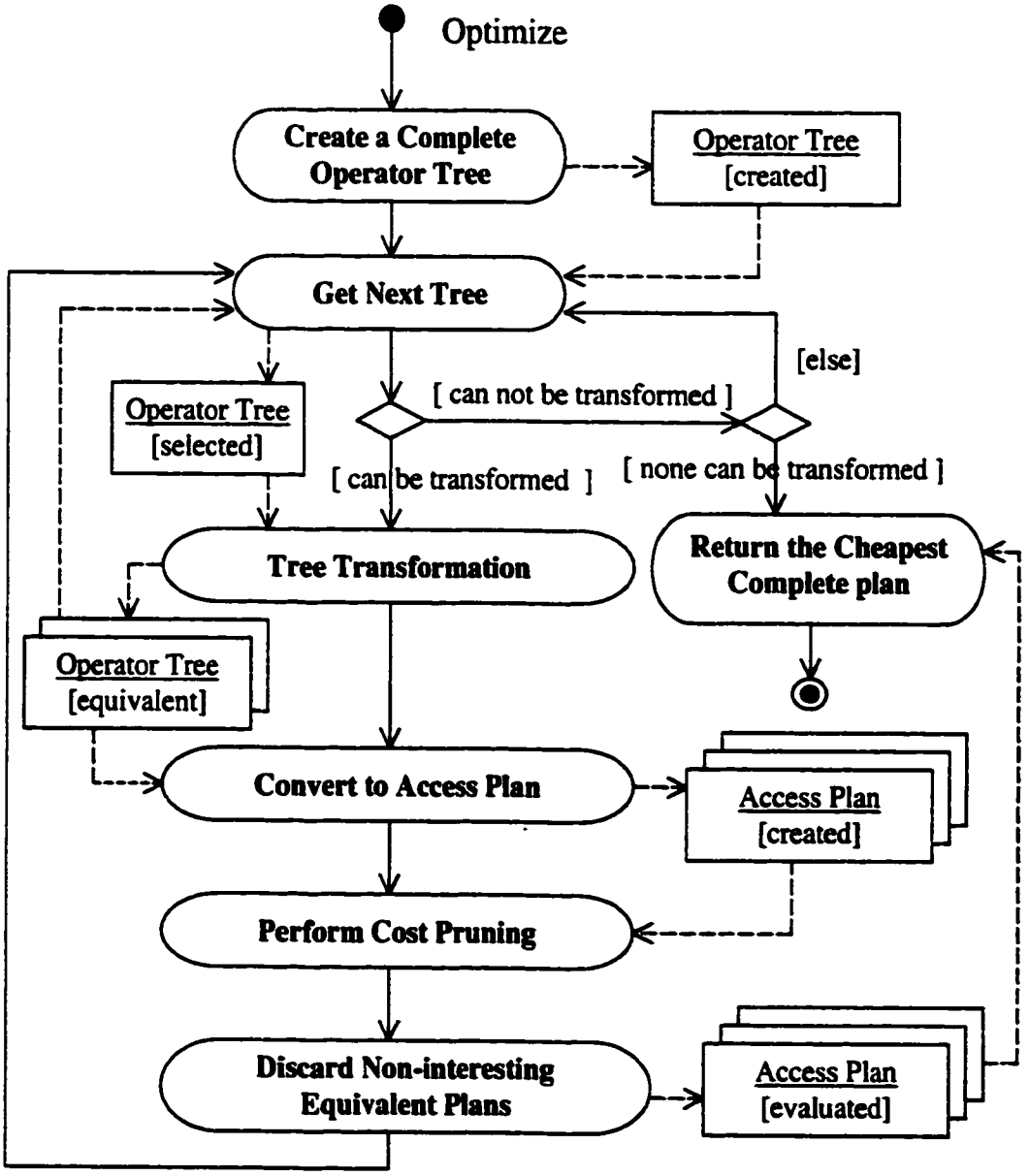

Figure 11: Transformative Search Strategy

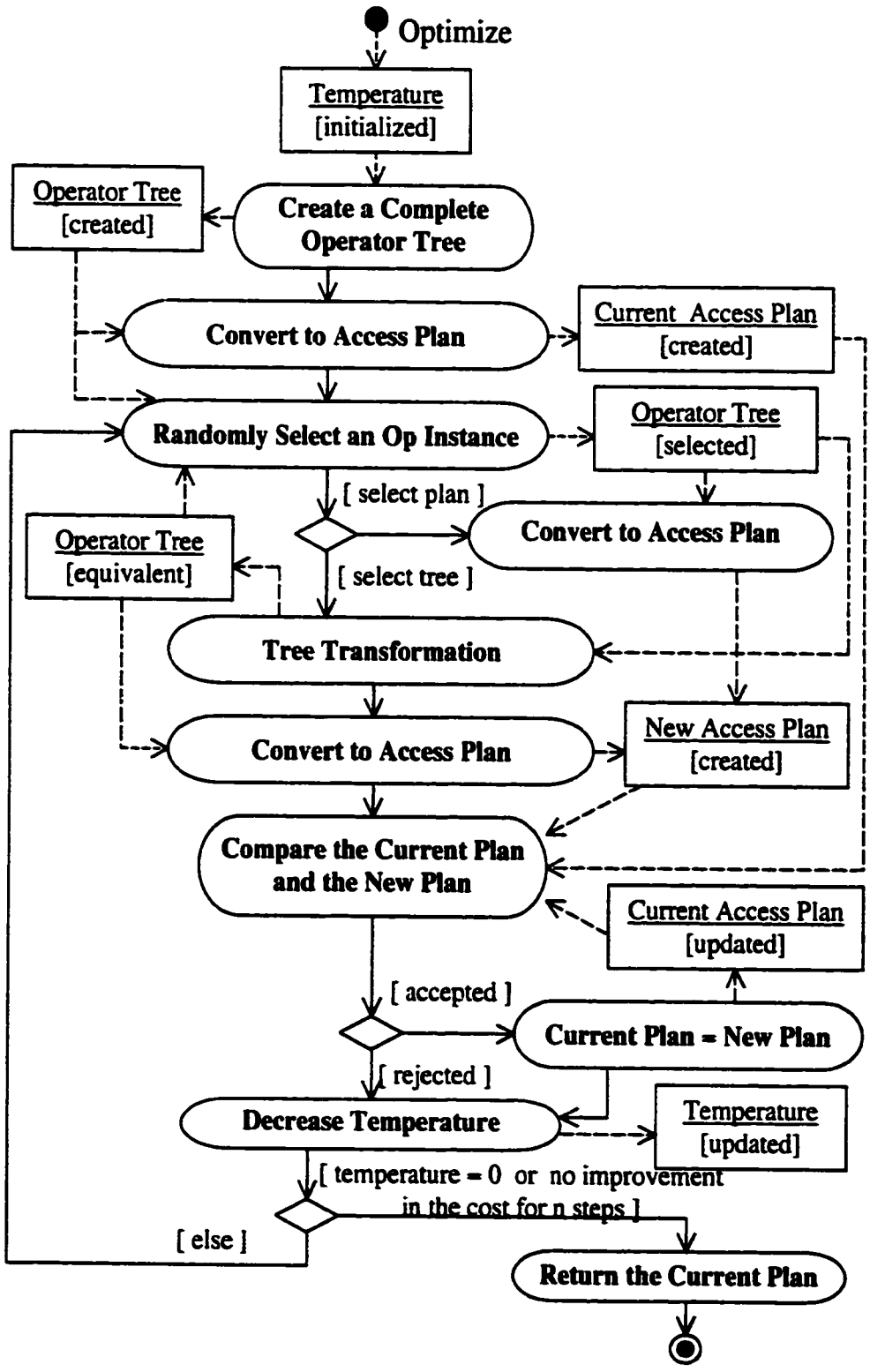

Figure 12: Simulated Annealing Search Strategy

Chapter 3

Framework Overview

This chapter is an overview of the query optimization framework, in which the context, problem, solution (static view and dynamic view), and consequences are discussed in detail.

Name Query Optimization Framework

Type Whitebox framework. This implies that this framework is reused by inheritance of its base classes and overriding of its predefined hook methods.

Example Consider a simple University Course Query system. The purpose of this system is to let the students query the courses they take and the professors query the courses they teach in the current semester. The data in the database may be:

Atomic: Each student has a unique student id; each professor has a unique faculty id; each course has a unique course id.

Set value: A student can register more than one course. A professor can teach more than one course. So for each student or professor, there is a set of courses s/he takes or teaches.

Reference link: The courses a student takes are stored as reference links to the corresponding courses in the course table. The courses a professor teaches are stored as reference links to the corresponding courses in the course table. Similarly, for each course, there are references to the student table representing all the students who take this course. There are also references to the professor table representing all the

professors who teach this course. The reference links amongst these tables allow easy navigation from one table to another.

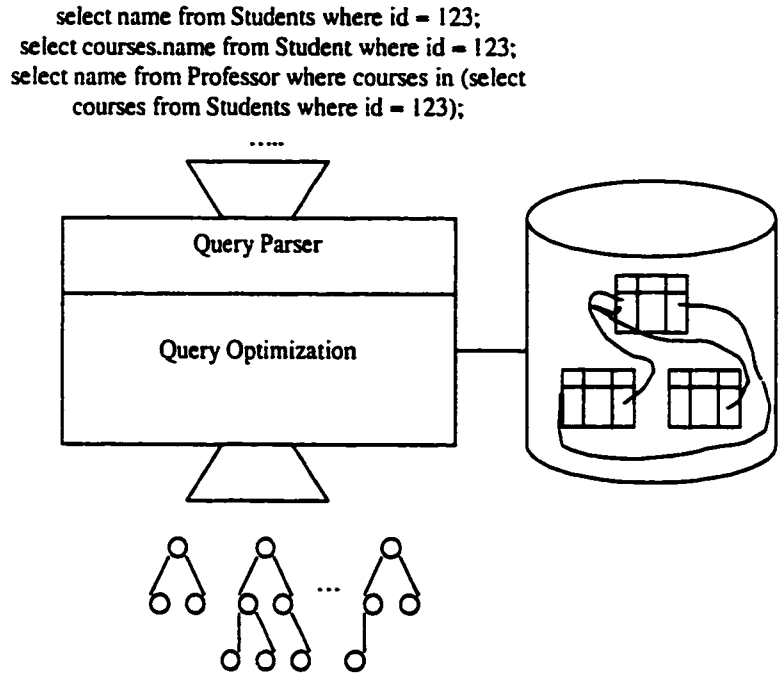

Figure 13: An Example of Query Optimization

Figure 13 shows that the user-written queries are parsed by the Query Parser. They are then optimized by the Query Optimization System according to the information stored in the database. Tree-like plans are produced at the end.

The Query Optimization System should make it easy to experiment with different search strategies, e.g. bottom-up optimization, transformation optimization, randomized optimization, or a combination of them. Because we don't know which one is better than the other in our system before we try them. It should also make it easy to change the database representation. For example, the database may be simplified to only allow atomic values; it may be extended to add some object-oriented constructs such as functions; or the index information may be added. Furthermore, the University Course Query system may evolve to a University Course Registration system. That means, in addition to providing the query functionality, the system should provide functionality for data addition and data update because students and professors may use it to register for the courses they take or teach.

No matter what changes there may be, the majority of the query optimization

should be kept unchanged. In other words, we need not develop a query optimization system from scratch to cater for any changing requirement. Developing a new query optimization system and modifying the existing optimization system are both easy.

Context Extensible query optimization for the DBMS environment.

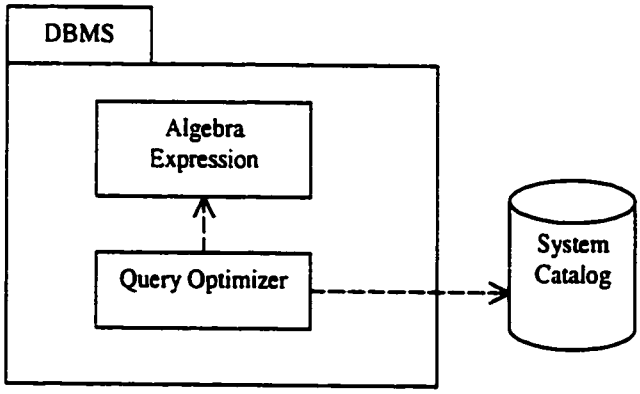

Figure 14: Query Optimization in Its Context

Figure 14 shows the query optimization in its context. There are three pieces of information implied in it:

- Query optimization is one of the components of the DBMS.
- Query optimization depends on the algebra expression, which is used to represent a query. Any query to be optimized must be translated into an algebra expression in a format accepted by the query optimization.
- Query optimization depends on the system catalog information. The optimization extracts information such as relations, indexes, costs, and so on from the system catalog, builds the plan trees, and decides which plan is superior to the others.

Figure 15 is a close look at query optimization. The central part is the Query Optimization Core. It implements the majority functionality of query optimization. There are two volatile parts in query optimization. One is the Search Strategy, the other is the Database Algebra. On one hand, the Query Optimization Core depends on the search strategy that is used for search, which is supplied by the Search Strategy

part. What search strategy to use is up to the user who customizes the system. On the other hand, the Query Optimization Core depends on the Database Algebra, which encapsulates the logical operators and physical operators in a database that can be relational, object-relational, or object-oriented.

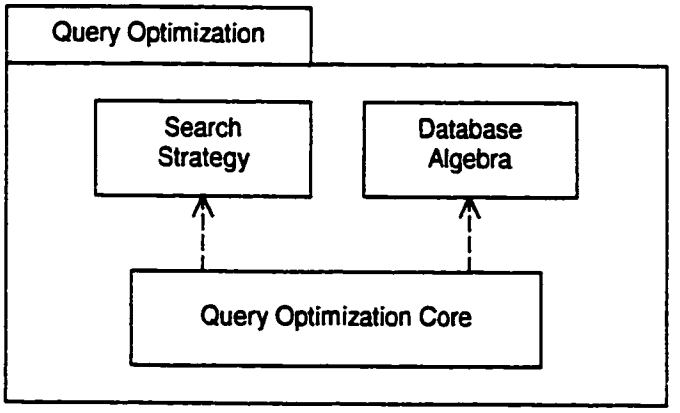

Figure 15: A Close Look at Query Optimization

Problem There exist requirements conflicts for the query optimization system. Some people may prefer to use it in a relational database system, some may prefer an object-relational system, some may prefer an object-oriented system.

There are various search approaches that are used in query optimization such as Bottomup, Transformation, Simulated Annealing, Iterated Improvement, Two Phase Optimization, etc. We do not know which one is best suited for the system we are building until we experiment on them.

Building a query optimization system is hard and expensive if it is tied to the specific database and/or search strategy. This can result in the need to develop and maintain a few substantially different query optimization systems, one for each database or search strategy.

The forces that influence the solution are:

- The same query optimization system can be easily modified and extended to build a database system that is relational, object-relational, or object-oriented.
- Different search strategy approaches can be tried on the newly built system before the best one is chosen.

- New constructs of the database algebra should be easy to add; old ones can be easy to modify and remove.
- Code which implements the core functionality of query optimization can be reused again and again with minimum changes. The common functionality may include the system control flow, the cost model and cost pruning, representation of the tree structures for the logical query plan and physical query plan, etc.

Solution Kabra proposed a solution structure. He developed a query optimization framework named OPT++ [20] that implements the core functionality of query optimization. But he put the whole framework in the Search Strategy component. We believe a clear partition of the query optimization system is necessary. Each component should contribute to a single role. It is the collaboration of different components that makes the whole system lively. A framework that spans across different components is then natural and reasonable. Figure 16 shows our solution structure.

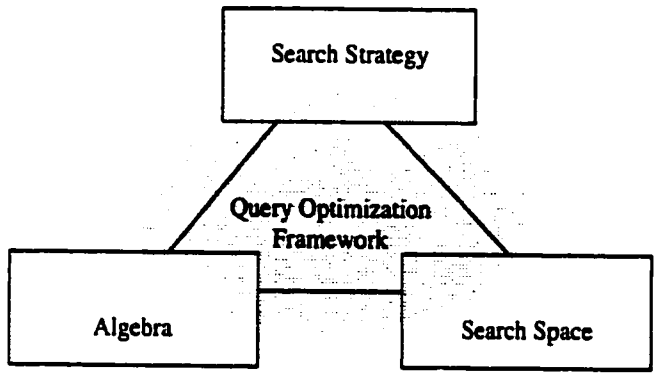

Figure 16: Solution Structure

The Search Strategy component encapsulates the search approaches that are used to control the exploration of the optimal plan. Different search strategies can be built in this component. It also encapsulates the cost model for plan evaluation and it implements the cost estimation model. Representation of the tree structures for the logical query plan and the physical query plan is also incorporated.

The Algebra component incorporates the representation of the operations on the database and the various implementation algorithms for these operations. It is an abstraction of the potential operations on the various kinds of databases.

The Search Space component defines what the search space is. It decides what logical query plans and physical query plans are built. The abstraction of the kinds of operations that may perform on the database are encapsulated in this component.

The separation of the Search Strategy component, the Algebra component, and the Search Space component has an explicit advantage in that it clarifies the roles of different components in query optimization and helps to better understand the complex query optimization system. On the other hand, each component contributes to only one purpose and has a focus. Relationships amongst these components can be simplified so that individual component can be experimented and improved with minimum impact on the others. We believe this separation is the first step towards building a good query optimization framework.

Note that the framework encapsulates the relationships amongst the components. It specifies the protocols about how these components communicate. In such a way, the majority of the code including the complex implementation of these relationships are implemented in this framework. It therefore eases the work to develop a new query optimization system and reduces the overhead that requires the optimizer-implementor to understand and implement these relationships.

Structure Figure 17 shows the simplified structure of the query optimization framework. It is simplified because it represents the overall structure at a high level of abstraction and hides the implementation details. A simplified structure is easy to understand and can help to form a conceptual model of this framework.

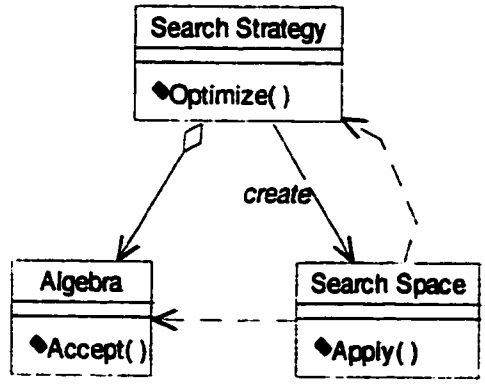

Figure 17: Simplified Architecture

The Search Strategy component encapsulates the system control flow. It implements different search approaches to explore the optimal plan. At any moment in time, only one search approach dominates. This component also encapsulates the cost model and implements the cost estimation framework. It contains the logical query plans and physical query plans that have been built, and it performs the cost evaluation on these plans and prunes out the sub-optimal ones. The Search Strategy component maintains an aggregation reference to the Algebra component because each logical query plan is represented by applying a logical operator (in the Algebra component) to its root node, and each physical query plan is represented by applying a physical operator (in the Algebra component) to its root node. The Search Strategy component defines a visible method *Optimize* that can be called when a query is to optimize.

The Algebra component defines the logical operators (operations on the database) and the physical operators (implementation algorithms for these logical operators) in the database. The *Accept* method allows Logical operators to be traversed by some visitors that represent operations on them. It supports easy extension of operations that are performed on these operators without changing and re-compiling the Algebra component.

The Search Space component defines what the search space is. It defines the operations on the Algebra. The effects of executing the operations in the Search Space component are

- An operator tree is transformed to another. The new tree becomes bigger as a result of tree expansion or these two trees are equivalent as a result of equivalent tree transformation.

- A logical query plan is transformed to a physical query plan.

In our example system, operators such as Select, Join, Materialization, Unnest and their corresponding implementation algorithms are implemented by subclassing the base classes defined in the Algebra component. Operations that perform on these operators/algorithms including rules for transformation are implemented in the Search Space component by subclassing its base classes or by following the protocols defined in the framework. The Search Strategy component can be kept unchanged. It can also be overridden or extended by subclassing the existing search approaches.

Dynamic Figure 18 describes the dynamic behavior of how a query is optimized in a sequence diagram. It is an abstract scenario based on the simplified overall structure. We will discuss this sequence diagram with two scenarios. For simplicity, only one operator instance and one algorithm instance are shown in the sequence diagram.

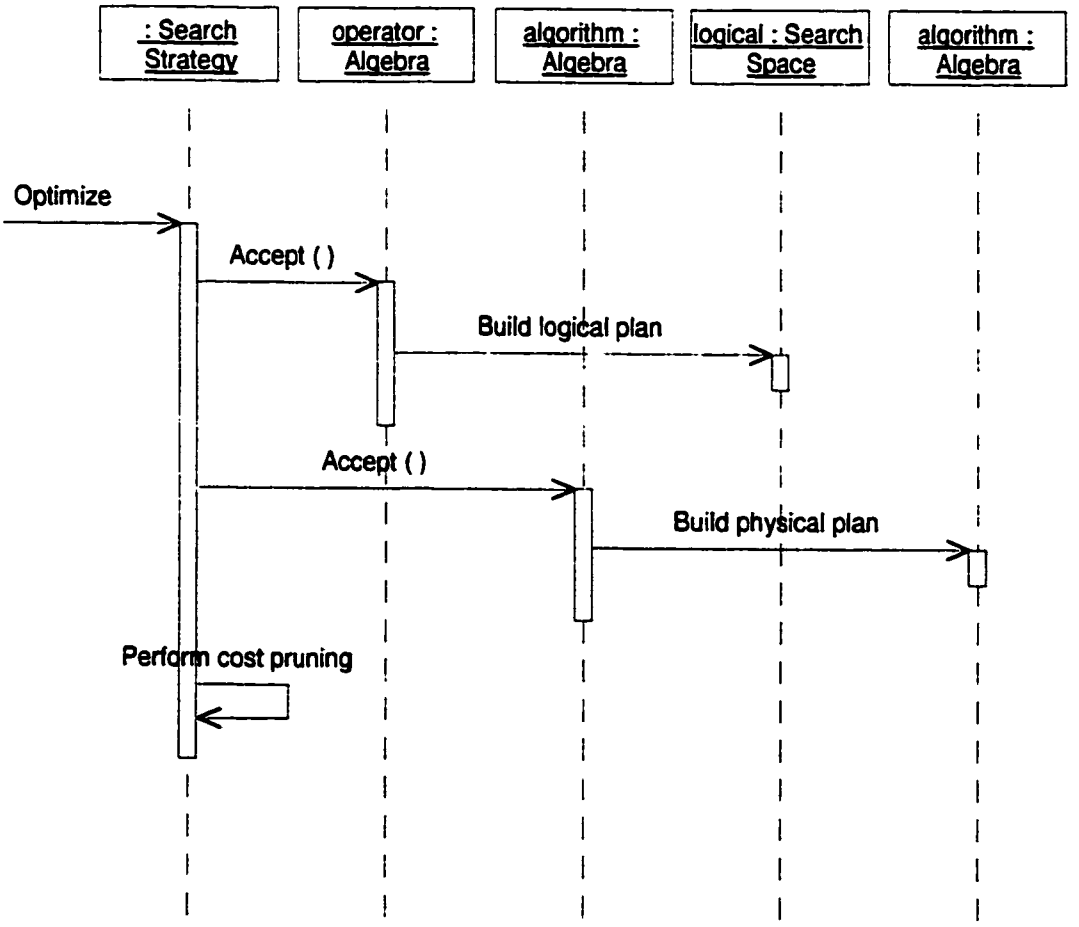

Figure 18: Dynamic Behavior of Query Optimization

Scenario 1 shows the Bottomup optimization approach:

1. The Search Strategy component accepts the user request for optimizing a query.
2. The logical operators in the Algebra component are applied, which in turn call the logical search space to build initial logical query plan trees.
3. The physical operators in the Algebra component are applied, which in turn call

the physical search space to build the physical query plan trees for the logical query plans created in step 2.

4. The Search Strategy component performs the cost pruning on the physical query plans created in step 3. Equivalent plans that have higher costs are pruned out.

5. Apply all logical operators to expand the newly created logical query plan trees.

6. Apply all physical operators to convert the newly created logical query plans into physical query plans.

7. Perform cost pruning on the physical query plans generated in step 6.

8. Repeat step 5 to 7 until no logical query plan trees can be expanded.

9. Return the physical query plan with the least cost whose corresponding logical query plan represents the complete query.

Scenario 2 shows the Transformation optimization approach:

1. The Search Strategy component accepts the user request for optimizing a query.

2. The logical operators in the Algebra component are applied, which in turn call the logical search space to build a logical query plan tree that represents the complete query.

3. The physical operators in the Algebra component are applied, which in turn call the physical search space to build the physical query plan trees for the newly created logical query plan.

4. The Search Strategy component performs the cost pruning on all newly created physical query plans. Equivalent plans that have higher costs are pruned out.

5. Algebraic laws that are defined in the logical search space are used to transform the newly created logical query plan to its equivalents.

6. Build the corresponding physical query plans for the logical query plans generated in step 5.

7. Perform cost pruning on the physical query plans generated in step 6.

8. Repeat step 5 to 7 until no algebraic laws can be used to transform the existing logical query plan trees.

9. Return the physical query plan with the least cost.

Known Uses The System-R style optimizer proposed a cost estimation framework and implemented it. The cost estimation in this query optimization framework uses the same cost estimation model as that of the System-R optimizer.

The separation of the logical and physical operators, the implementation of the logical and physical properties, selectivity, input constraints for execution algorithms, enforcers ("glue" operators), cost estimation functions can also be found in the Volcano Optimizer Generator, the Starburst optimizer, and OPT++. Search strategies that are built in this framework can also be found in the Exodus Optimizer Generator, the Volcano Optimizer Generator, and OPT++.

The object-oriented techniques such as inheritance and late binding that are used in this framework have been proved to be beneficial to make a query optimizer extensible in OPT++.

Consequences The reuse of the query optimization framework has the following benefits:

- Building a new query optimization system is easy.
  The majority code in the query optimization including the control flow for the search process, the different search strategy approaches, the manipulation of the tree structure, the relationships amongst the components, and the cost estimation functionality are already implemented in this framework. Building a new query optimization system using this framework only needs to define the actual logical and physical operators in database, and the operations that are performed on them. Compared to the complexity of a query optimization system, writing this part of code is easy.

- Modifying a query optimization system built from this framework is easy.
  This framework is equipped with different search strategy approaches and allows the optimizer-implementor to experiment with them and choose one that is best suited for that system. These search strategies can also be easily modified or extended by simply subclassing the built-in search strategy classes.

Limiting or extending the search space is easy. The implementation details about how the search space is shaped are encapsulated in the Search Space component. Any changes to them are limited within this component.

Extending the database algebra is easy. Adding new database constructs or new execution algorithms only need to subclass the base classes in the Algebra component and add the operations performed on these constructs in the Search Space component.

- Maintaining this framework is easy.
  This framework is implemented with proven design patterns and object-oriented techniques. Its structure is extensible and flexible to facilitate future maintenance and evolution.

The use of the query optimization framework has the following limitations:

- Complexity.
  This framework is implemented as a whitebox framework. As any whitebox framework, it is reused by inheritance. This means that the optimizer-implementor needs to understand the interfaces defined in the framework in order to implement them.

- Based on assumptions.
  This framework is built on top of the abstract operators and execution algorithms. It put specific assumptions on the kinds of manipulations on these operators and their execution algorithms. The liability is that if these assumptions are incomplete, the framework is less general.

- Changing interfaces is hard.
  The search strategies are written with respect to the interfaces of the abstract operators and their execution algorithms. Any change of these interfaces means that the application systems built from this framework have to be changed accordingly.

Chapter 4

Framework Design

This chapter presents an object-oriented design for the query optimization framework. The structural aspects of this framework are fully discussed, including the overall structure, detailed design of each component, and design of the exception handling. Then the dynamic aspects of this framework are illustrated. The scenarios of how the user interacts with this system and how a query is optimized with different approaches are presented. Finally the cost evaluation for the physical query plans is illustrated. The reason to put the cost evaluation in a separate section is that it is the most critical part of query optimization and deserves some special treatment.

4.1 Overall Structure

Figure 19 shows the overall class diagram. In this figure, components of this framework are represented as packages and only the top-level classes in each component are shown.

The Search Strategy component encapsulates the system control flow. It implements different search approaches that are used to perform the search in the search space. At any instance of time, only one search approach dominates. It also encapsulates the cost model and implements the cost estimation framework. It contains the logical query plans and physical query plans that have been built, and it performs the cost evaluation on these plans and prunes out the sub-optimal ones. The Search Strategy component maintains an aggregation reference to the Algebra component because each logical query plan is represented by applying an operator (in the Algebra

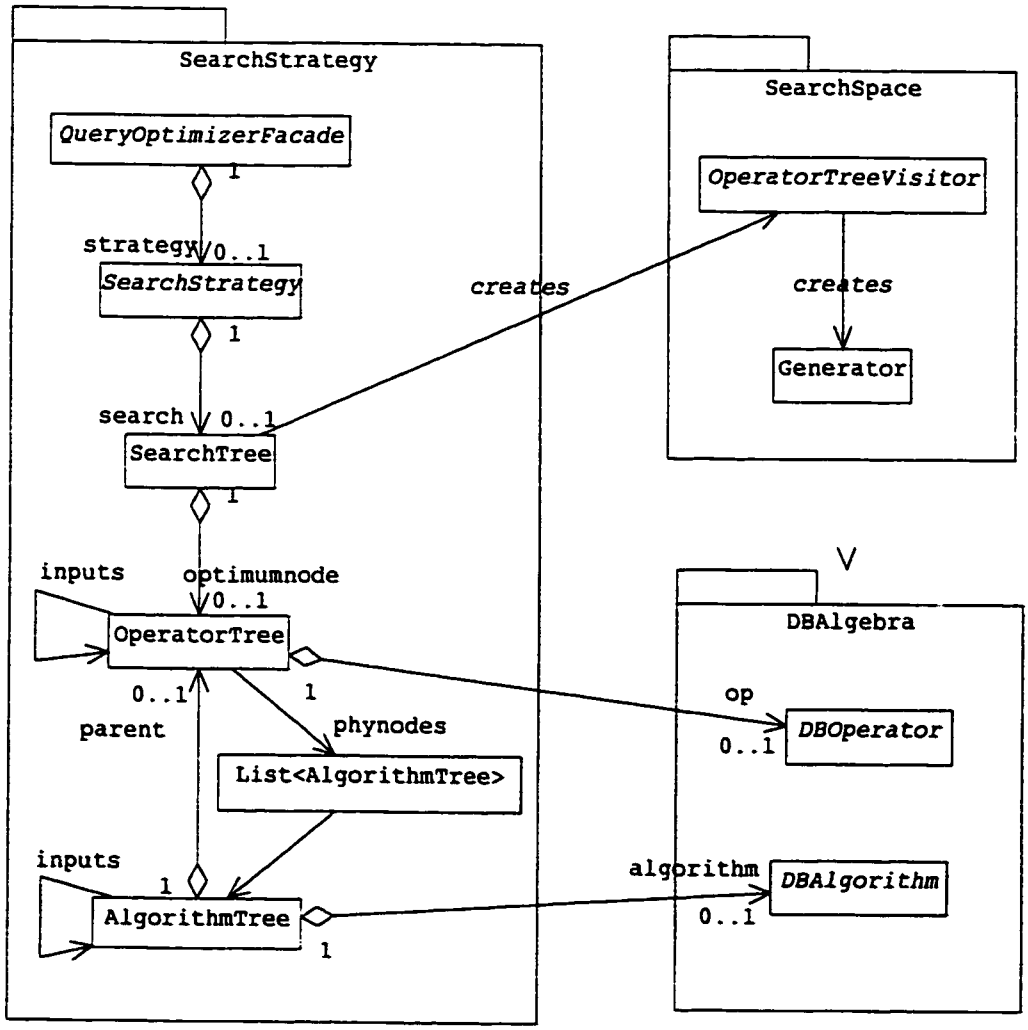

Figure 19: Overall Class Diagram

Component Search Strategy	Collaborators + Search Space + Algebra
Responsibility + Implements search strategies. + Controls search process. + Performs cost pruning.	

Table 1: CRC Card for the Search Strategy Component

component) to its root node, and each physical query plan is represented by applying an algorithm (in the Algebra component) to its root node.

The major classes in the Search Strategy component include: the QUERYOPTIMIZERFACADE, the SEARCHSTRATEGY, the SEARCHTREE, the OPERATORTREE, and the ALGORITHMTREE. The class QUERYOPTIMIZERFACADE is the facade of this system. It simplifies the use of the system. The SEARCHSTRATEGY is an abstract class and provides interfaces for all search strategy approaches that are used in query optimization. The SEARCHTREE represents the search tree that is used to explore the optimal plan. It implements the search strategy interfaces. The OPERATORTREE represents a logical query plan of a query. It is an algebraic expression that represents the particular operations to be performed on data and the necessary constraints regarding order of operations. The ALGORITHMTREE represents a physical query plan of a query. It is a tree of algorithms that specifies the particular order of operations and the algorithm used to implement each operation.

Table 1 shows the CRC card [29] for the Search Strategy component. The Search Strategy component collaborates with the Algebra component and the Search Space component. It utilizes the available resource provided by the other two components to perform the search process.

The Algebra component defines the logical operators (operations on the database) and the physical operators (execution algorithms for these logical operators) in the database.

There are two major classes in the Algebra component: the DBOPERATOR and the DBALGORITHM. The former defines the interfaces for all possible logical operators in the database system. The latter defines the interfaces for all possible execution algorithms for these logical operators. They are all abstract classes and depend on

Component Algebra	Collaborators + Search Strategy + Search Space
Responsibility + Define logical operators in the database. + Define physical operators in the database.	

Table 2: CRC Card for the Algebra Component

the optimizer-implementor to define the actual database algebra. The definitions of the interfaces for the logical and physical operators should consider easy extension of possible operations that are performed on them.

Table 2 shows the CRC card [29] for the Algebra component. The Algebra component collaborates with the other two components to provide the basic database operators and execution algorithms that will be used to build the search tree.

The Search Space component defines what the search space is. It defines the operations on the Algebra. The effects of executing the operations in the Search Space component are:

- A tree is transformed to another tree. The new tree becomes bigger as a result of expansion or these two trees are equivalent as a result of some equivalent transformations.

- A logical query plan is converted to a physical query plan.

Conceptually, there are two search spaces. One is the logical search space. It is a space of logical query plans and decides how one logical query plan derives from another. The other is the physical search space. It is a space of physical query plans and decides how one physical query plan derives from another. The conversion from the logical query plan to the physical query plan can be deemed as an operation on the logical search space.

There are two major classes in the Search Space component. One is the OPERATORTREEVISITOR and the other is the Generator. The OPERATORTREEVISITOR is an abstract class and defines all operations performed on the Algebra component. It serves as a bridge between the Algebra component and the Search Space component. The Generator is not a class defined in this framework. In fact, there are three generator classes, each is defined for one of the following operations:

Component Search Space	Collaborators + Search Strategy + Algebra
Responsibility + Tree expansion / tree transformation. + Converts a logical plan to a physical plan.	

Table 3: CRC Card for the Search Space Component

- Tree expansion
- Tree transformation
- Convert a logical query plan to a physical query plan.

Table 3 shows the CRC card [29] for the Search Space component. The Search Space component collaborates with the other two components. It defines the search space that is used to explore the best physical query plan that may exist in the search space.

4.2 The Search Strategy Component

4.2.1 Structure

Figure 20 shows the main class diagram of the Search Strategy component. The class details are shown in Figure 21, Figure 22, Figure 23, Figure 24, and Figure 25. The QUERYOPTIMIZERFACADE class is an abstract class. It defines the interfaces for the entry point of this system. There are two concrete facade classes in this framework: the QUERYOPTIMIZERFACADEWITHPARSER and the QUERYOPTIMIZERFACADE-WITHFORMATEDFILE. They differ in the way they pre-process queries. The former incorporates a query parser. It can accept user-input queries, parse them, and perform some pre-processing on these queries before further optimization. The incorporation of a query parser makes the query optimization system standalone, which is especially useful when the query optimization system needs to be experimented over and over before delivery. The latter reads queries from files that are written in a certain format specified in the framework.

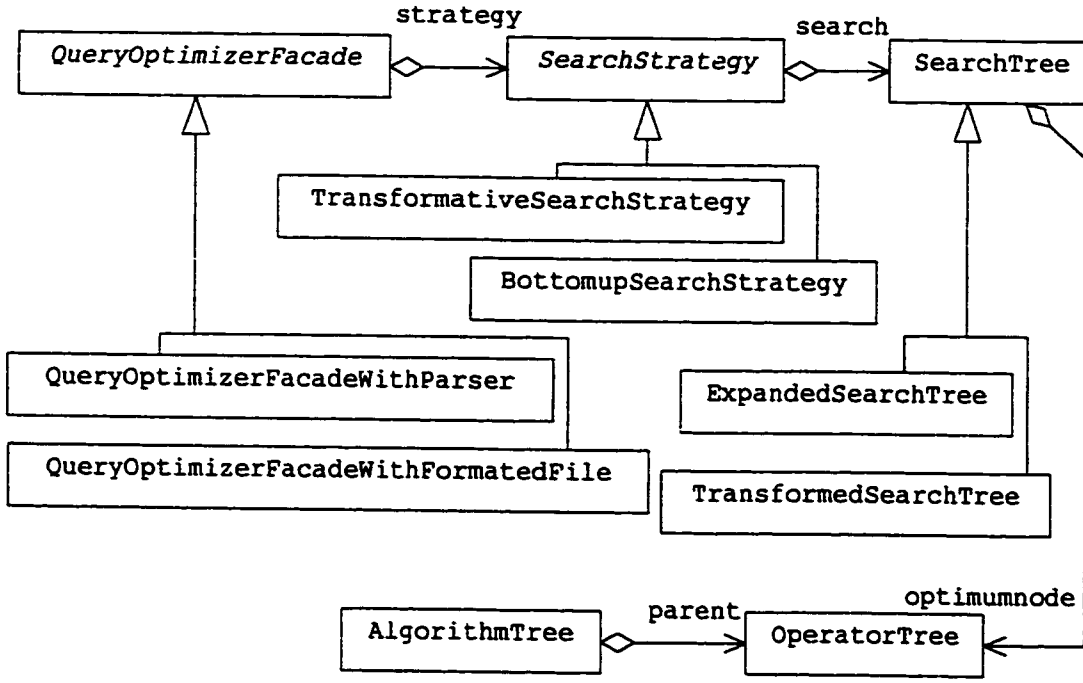

Figure 20: The Search Strategy Component

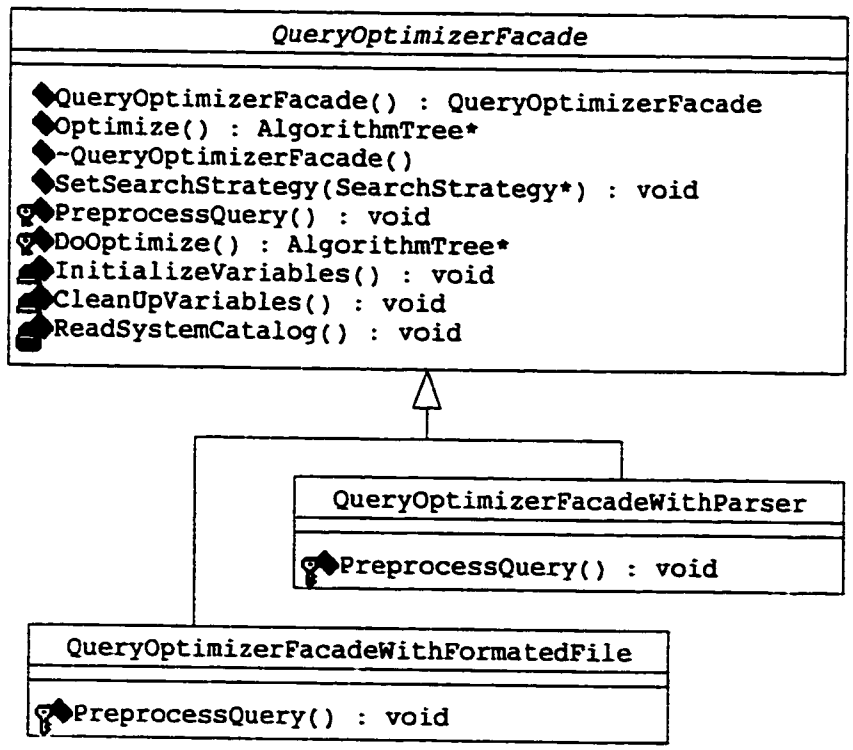

Figure 21: The Facade Hierarchy

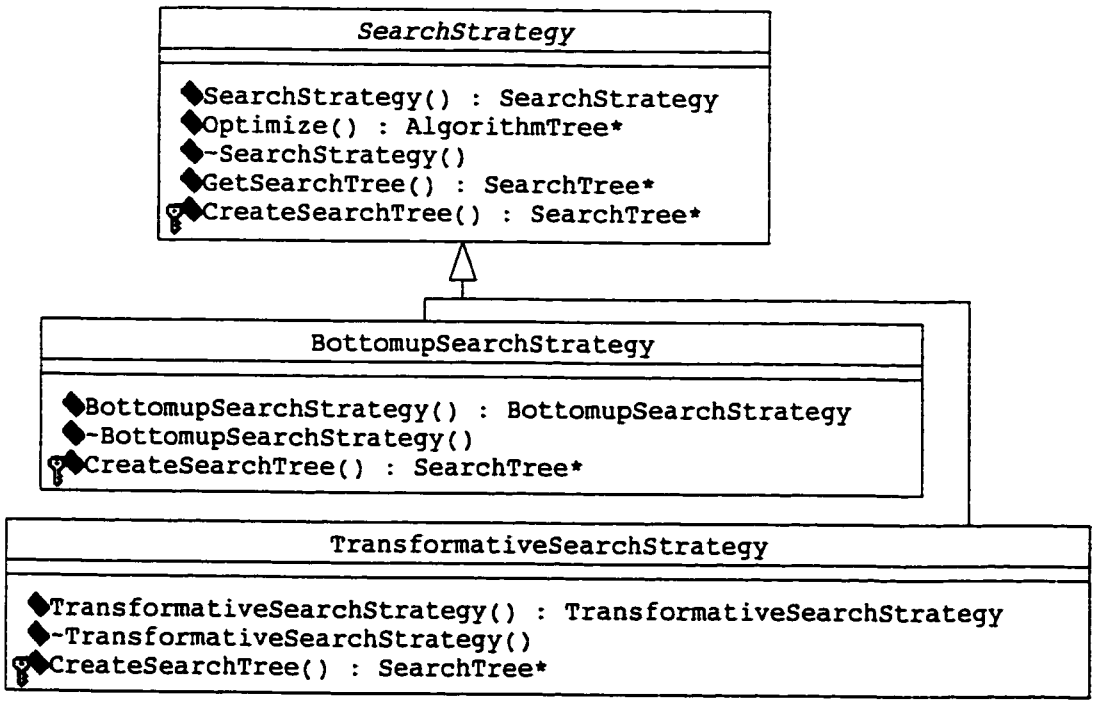

Figure 22: The Search Strategy Hierarchy

The SEARCHSTRATEGY class is an abstract class and defines the interfaces for the search approaches in the query optimization. The SEARCHSTRATEGY class consists of a search tree object that is used to perform the search. There are two search strategies implemented in this framework. One is the BOTTOMUPSEARCHSTRATEGY, and the other is the TRANSFORMATIVESEARCHSTRATEGY. They differ in that they use different search trees to perform the search process.

The SEARCHTREE class represents the search tree that is used to explore the optimal plan in query optimization. It encapsulates complex data structures that are used in the search. Two search tree subclasses are defined in this framework. The EXPANDEDSEARCHTREE is used by the BOTTOMUPSEARCHSTRATEGY, and the TRANSFORMEDSEARCHTREE is used by the TRANSFORMATIVESEARCHSTRATEGY. The SEARCHTREE consists of an OPERATORTREE object, which represents the optimal logical query plan (whose physical query plan has least cost). It also consists of two lists of OPERATORTREE objects: one is a list of unexpanded operator trees that will be expanded in the optimization, the other is a list of atomic operator trees that may be used to combine with other operator trees in the tree expansion process.

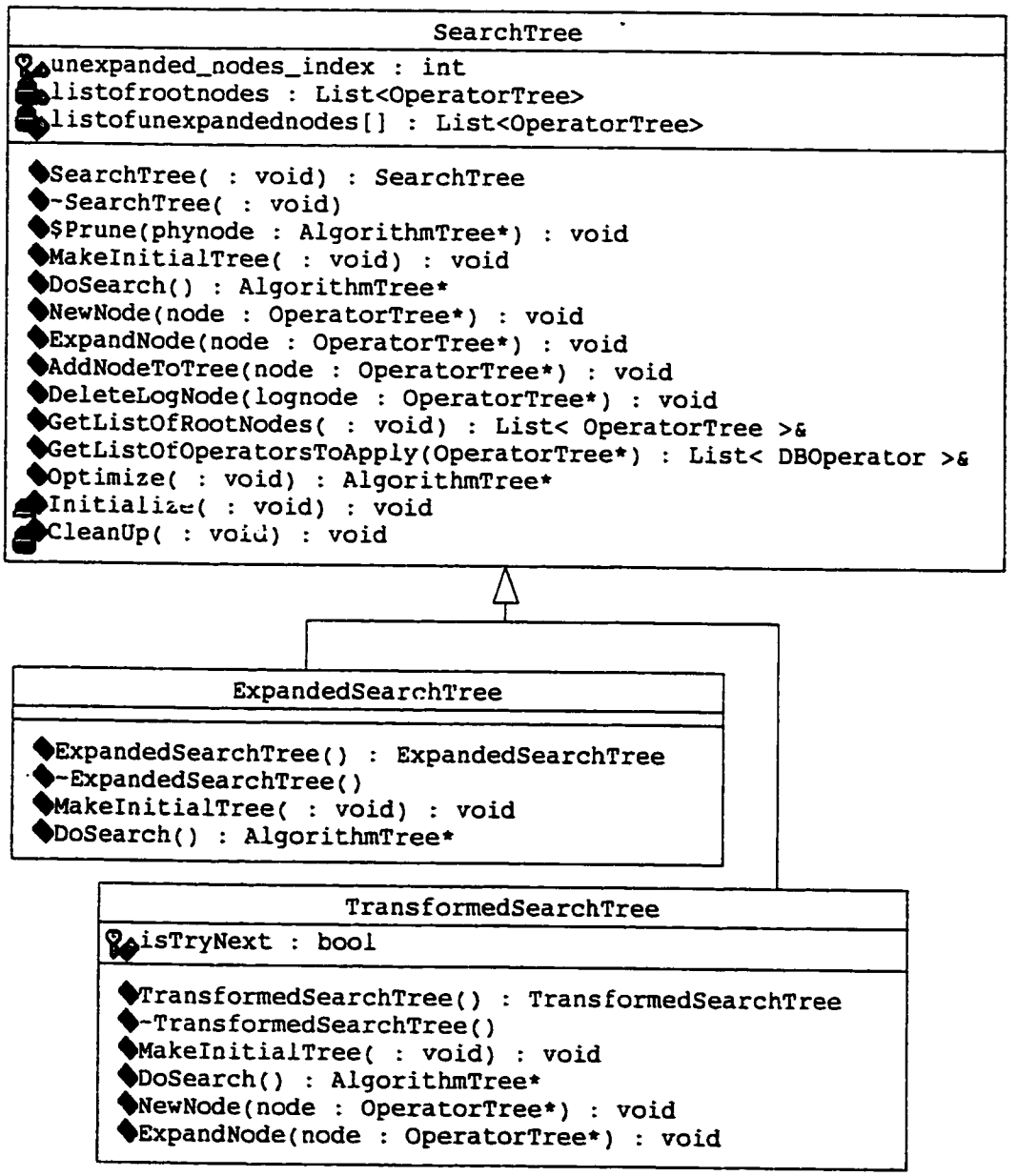

Figure 23: The Search Tree Hierarchy

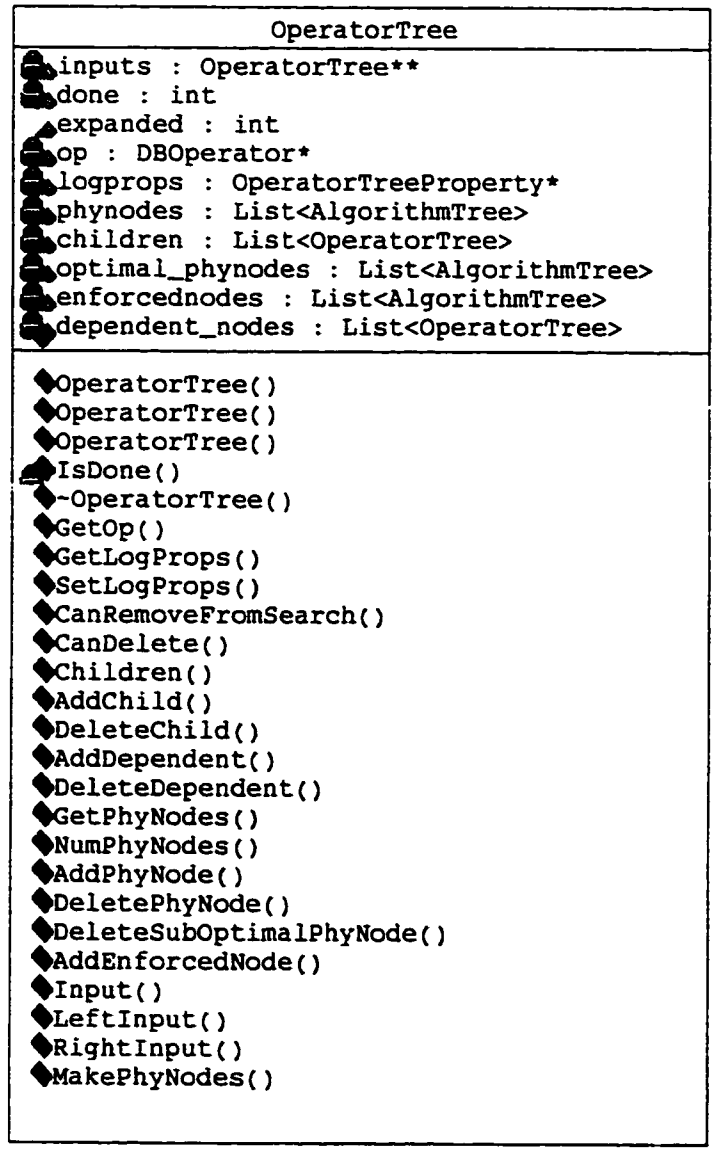

Figure 24: The OPERATORTREE Class

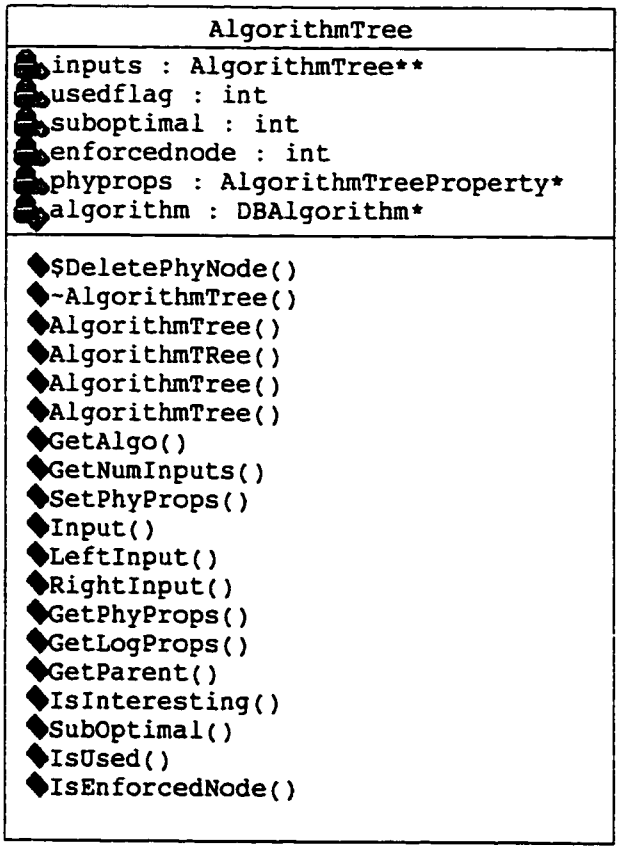

Figure 25: The ALGORITHMTREE Class

The OPERATORTREE class is used to represent logical query plans, and the ALGORITHMTREE class is used to represent physical query plans associated with some logical query plans. An OPERATORTREE instance consists of a list of inputs (instances of the OPERATORTREE), and a list of associated physical query plans (instances of the ALGORITHMTREE). An ALGORITHMTREE instance consists of a list of inputs (instances of the ALGORITHMTREE), and a reference to its parent (an instance of the OPERATORTREE).

4.2.2 Design Patterns Used

There are three patterns used in the Search Strategy component.

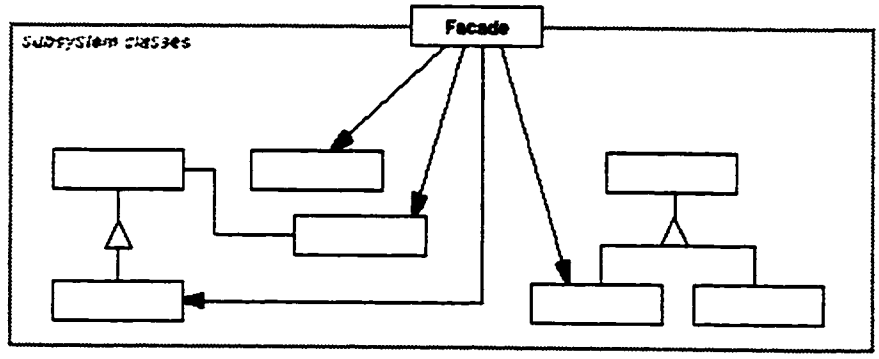

Figure 26: Facade Design Pattern

4.2.2.1 The Facade Design Pattern

Figure 26 shows the standard structure of the Facade pattern [5]. In the query optimization framework, the QUERYOPTIMIZERFACADE and its subclasses play the role of Facade of the standard structure.

There are three reasons behind the use of the facade pattern:

- Query optimization in the simplest sense is to give a parsed query and ask for an optimal access plan. The use of facade simplifies the interface of the complex optimization system. It provides the default view of this system and the clients are not required to understand what search strategies should be used and how they work.

- There exist dependencies between the clients and the system. For example, the clients need to prepare for some inputs and initialize some environment variables that may relate to some other subsystems. The use of facade decouples the system from clients and other subsystems because all interactions are centralized in the facade class.

- The facade class defines an entry point to the query optimization system. Its use promotes layering of the DBMS system because different layers can communicate only through their facades.

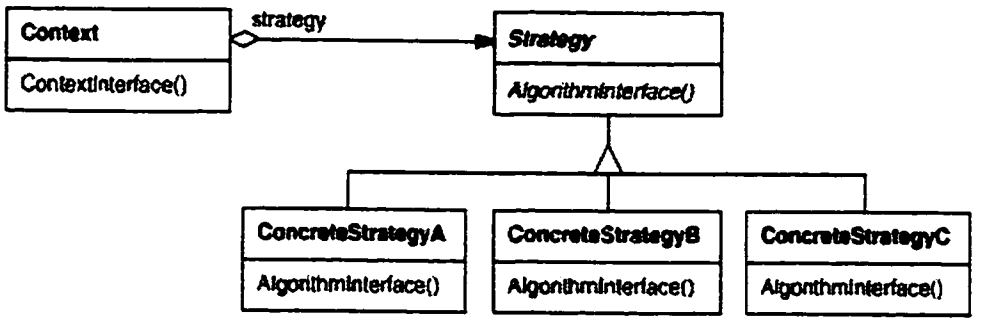

Figure 27: Strategy Design Pattern

4.2.2.2 The Strategy Design Pattern

Figure 27 shows the standard structure of the Strategy pattern [5]. In this framework, the QUERYOPTIMIZERFACADE class plays the role of Context, the SEARCHSTRATEGY class plays the role of Strategy, and the BOTTOMUPSEARCHSTRATEGY and the TRANSFORMATIVESEARCHSTRATEGY play the role of Concretestrategy in the standard structure.

There are two reasons behind the use of the Strategy design pattern:

- Different search strategy approaches that are used in query optimization only differ in their behavior. With the use of the Strategy pattern we can configure one of these approaches at run time in the query optimization system.
- Different search strategy approaches have tradeoffs. The Strategy pattern provides a way to offer various choices to the client.

4.2.2.3 The Factory Method Design Pattern

Figure 28 shows the standard structure of the Factory Method design pattern [5]. In this framework, the SEARCHSTRATEGY class plays the role of Creator, the BOTTOMUPSEARCHSTRATEGY class and the TRANSFORMATIVESEARCHSTRATEGY play the role of ConcreteCreator, the SEARCHTREE class plays the role of Product, and the EXPANDEDSEARCHTREE and the TRANSFORMEDSEARCHTREE play the role of ConcreteProduct in the standard structure.

The rationale behind the use of the Factory Method design pattern is that the SEARCHSTRATEGY abstract class does not know which search tree to use so it needs

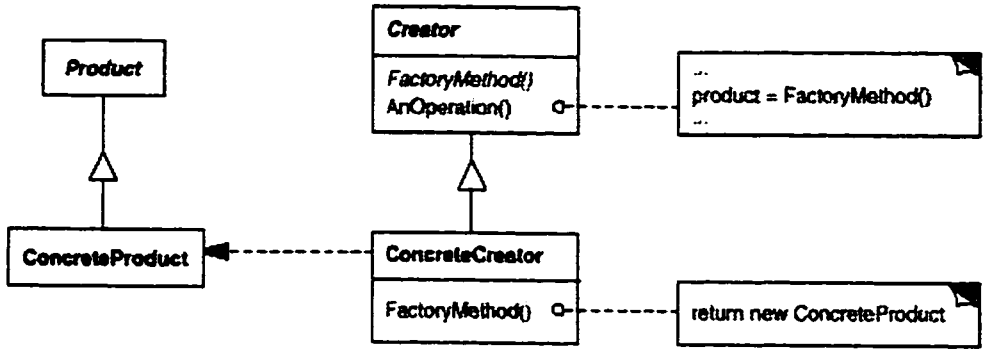

Figure 28: Factory Method Design Pattern

to defer their creation to its concrete subclasses.

4.2.3 Separation of the Search Strategy and the Search Tree

In the query optimization framework, we make a clear separation of the search strategy and the search tree. The former represents the interface for all search strategies, and the latter is the implementation of this interface.

We think this separation is necessary and important in the reusable object-oriented design in order to achieve a reusable object-oriented query optimization framework. There are many advantages for this separation. First, the separation avoids exposing complex data structures that are used in the implementation to the client, thus promoting information encapsulation. Second, the implementation of a search strategy is volatile because it will be improved over time. The separation of the interface from its implementation maintains a stable interface to the client and allows its implementation to be changed without re-compiling the client code. Third, with this separation, the search strategy becomes a logical concept, thus leading to a good understanding of the query optimization system and allowing different implementation algorithms to be experimented for each search strategy.

4.2.4 Description of the Major Classes

4.2.4.1 The Facade Classes

The QUERYOPTIMIZERFACADE Class

The QUERYOPTIMIZERFACADE is an abstract class. It provides an entry point to the query optimization system with two visible methods:

Optimize — optimizes a user-written query written in a query language and returns the physical query plan that has least execution cost and that corresponds to the complete query. By default, the Bottom-Up Search Strategy is used to perform the search.

SetSearchStrategy — provides the sophisticated user with a choice to change the default search strategy. The argument search strategy will be used as the new search strategy for search.

One important method defined in this class is the *PreprocessQuery*. It is a protected abstract method and should be implemented by the immediate concrete subclasses. This method is defined to resolve the user conflicts and provides two means for the user to access to this system. A major functionality of it is to perform semantic checking on the input query, which includes:

1. Check relation uses. Any relation involved in the query must be a relation or view in the system catalog.

2. Check and resolve attribute uses. Attributes mentioned in the query should exist in some relations in the current scope. Also, if the relation name an attribute refers to is absent in the present of this attribute, it is attached to this attribute.

3. Check types. All attributes are of appropriate types to their use. For example, if, in the system catalog, the attribute name of relation Persons is of type string, then **Persons.name = john** is invalid while **Persons.name = 'john'** should be accepted (supposing the syntax for string is quoted within a single quote).

The QUERYOPTIMIZERFACADEWITHFORMATEDFILE Class

The QUERYOPTIMIZERFACADEWITHFORMATEDFILE class is a subclass of the

QUERYOPTIMIZERFACADE class. It implements the method *PreprocessQuery*. It reads the parsed query from a file named **.parsedquery** which is written in a certain format specified in the framework and then performs semantic checking.

The QUERYOPTIMIZERFACADEWITHPARSER Class

The QUERYOPTIMIZERFACADEWITHPARSER class is a subclass of the QUERYOPTIMIZERFACADE class. It implements the method *PreprocessQuery* that incorporates a query parser developed from the Yacc framework. The incorporation of a query parser makes the query optimization system standalone, which is good for experiment purpose. This method accepts user input queries from the text mode interface, interprets them, parses them, and performs semantic checking on them. This class is isolated and may allow future extension to incorporate a query parser with graphical interface without affecting the rest of the system.

4.2.4.2 The Search Strategy Classes

The SEARCHSTRATEGY Class

The SEARCHSTRATEGY class is an abstract class. It defines the interfaces for all possible search strategy approaches used in query optimization. It consists of a search tree (an instance of the SEARCHTREE class) and defines three methods:

Optimize — a public method to optimize the user query and returns the physical query plan that has least execution cost and that corresponds to the complete query. It performs its work by delegating to the searchTree object it contains.

GetSearchTree — a public method that returns its attribute searchTree.

CreateSearchTree — a protected abstract method. It will be implemented by the subclasses.

searchTree — a private attribute. It represents the actual search tree that is used to perform the search.

The BOTTOMUPSEARCHSTRATEGY Class

The BOTTOMUPSEARCHSTRATEGY class is a subclass of the SEARCHSTRATEGY abstract class. It implements the *CreateSearchTree* method and returns a newly created object of the EXPANDEDSEARCHTREE class.

The TRANSFORMATIVESEARCHSTRATEGY Class

The TRANSFORMATIVESEARCHSTRATEGY class is a subclass of the SEARCHSTRATEGY abstract class. It implements the *CreateSearchTree* method and returns a newly created object of the TRANSFORMEDSEARCHTREE class.

The classes TRANSFORMATIVESEARCHSTRATEGY and BOTTOMUPSEARCHSTRATEGY differ in the way they create and use different search trees for search.

4.2.4.3 The Search Tree Classes

The SEARCHTREE Class

The SEARCHTREE class is an abstract class. It represents a search tree that is used to explore the search space in order to locate the "best" physical plan.

The following shows some major methods and attributes defined in this class:

Prune — a static method that is used for cost pruning on all physical query plans.

MakeInitialTree — a public method with empty implementation. It will be overridden in the subclasses to create initial operator tree(s).

DoSearch — a public method with empty implementation. It will be overridden in the subclasses to perform the search on the search tree.

NewNode — a public method to add a newly created operator tree to the search tree for later processing.

ExpandNode — a public method to expand an operator tree. An operator tree is expanded when a logical operator is applied to it and a bigger operator tree is built.

Initialize — a protected method to initialize the environment for search.

CleanUp — a protected method to clean up the search environment when the optimal physical query plan is found.

listofunexpandednodes — a protected attribute that contains lists of unexpanded operator trees that will be expanded in an increasing order of number of operations they contains.

listofrootnodes — a protected attribute that contains list of atomic operator trees that may be used to combine with other operator trees to

build bigger operator trees.

unexpanded_nodes_index — a protected attribute that represents an index for the position in the listofunexpandednodes where the unexpanded operator trees are under expansion.

The EXPANDEDSEARCHTREE Class

The EXPANDEDSEARCHTREE class is a subclass of the SEARCHTREE class. In this class, the methods *MakeInitialTree* and *DoSearch* are re-written. The former creates a set of initial operator trees according to the relations mentioned in the query. For example, for the following query, the *MakeInitialTree* method will create an initial operator tree that consists of one node, that is, the relation Students.

select name **from** Students **where** id = 123;

The *DoSearch* method controls the search and continuously expands the operator trees until a logical query plan representing the complete plan is reached and there is no need to expand it. A logical query plan (an operator tree) is considered complete when its logical properties contain all the expressions (including predicates) in the query.

The TRANSFORMEDSEARCHTREE Class

The TRANSFORMEDSEARCHTREE class is a subclass of the SEARCHTREE class. In this class, the methods *MakeInitialTree*, *DoSearch*, *NewNode*, and *ExpandNode* are re-written.

MakeInitialTree — a public method that creates a complete logical query plan that represents the complete query, and converts this logical query plan to its corresponding physical query plans. For example, for the query

select name **from** Students **where** id = 123;

the *MakeInitialTree* method will create an initial operator tree as seen in Figure 29 and the physical plans associated with it.

DoSearch — a public method that continuously transforms the initial logical query plan to its equivalents, generates corresponding physical

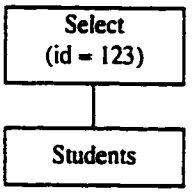

Figure 29: An Initial Tree

query plans, performs cost pruning, and finally returns the physical query plan with least cost.

Methods *NewNode* and *ExpandNode* are re-written to limit the possible combinations of the logical operators when the initial logical query plan is built. In the class EXPANDEDSEARCHTREE, all logical operators are applied to explore all possible combinations to build new logical query plans. But in this class, only one logical operator is applied at a time. Attribute isTryNext is the associated attribute that is used for the same purpose. It limits the number of operators that can be applied to expand an operator tree. If one operator is already applied to build a new operator tree, then isTryNext helps to stop trying all the other operators that can be applied to this tree to generate alternative operator trees. For example, when the query

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

is optimized, the initial operator trees are in Figure 30.

Figure 30: Initial Trees

When the operator tree t2 is expanded, there are two alternatives: If the Select operator is applied, operator tree t3 will be generated; If the Join operator is applied, then t4 and t5 will be generated (see Figure 31). In the class EXPANDEDSEARCHTREE, all the trees t3, t4, t5 are generated for further optimization. But in this class, either t3 or t4 and t5 are generated depending on the order of applying these operators. That is, if operator Select is applied before the operator Join, then only t3 is generated. Otherwise, both t4 and t5 are generated.

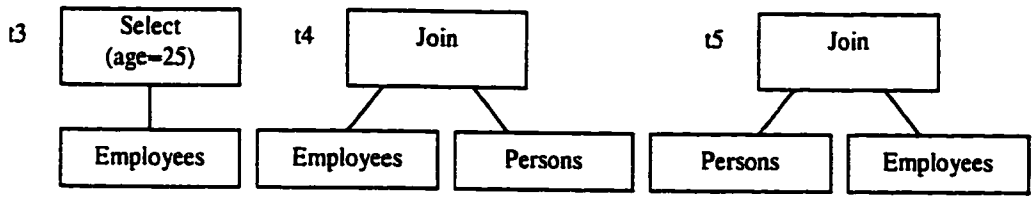

Figure 31: Expanded Trees

The OPERATORTREE Class

The OPERATORTREE class represents the logical query plan in query optimization.

op — a private attribute which is a pointer instance of the class DBOPERATOR. The operator tree is built by applying the logical operator op to the tree's inputs. The method associated with it is *GetOp*, which returns this object.

inputs — a private attribute representing the inputs (operator trees) of this tree. The current tree is the result of applying the attribute op to the input trees. Related methods include *Input*, *LeftInput*, and *RightInput*. *Input* returns the Nth input if N is the provided parameter. *LeftInput* returns the first input, and *RightInput* returns the second input.

phynodes — a private attribute representing a list of physical query plans associated with this operator tree. Related methods include *MakePhyNodes*, *GetPhyNodes*, *NumPhyNodes*, *AddPhyNode*, and *DeletePhyNode*. *MakePhyNodes* dispatches to the search space to convert the current operator tree to corresponding physical query plans. When a logical query plan is built, all its corresponding physical plans are generated and used for cost pruning. *GetPhyNodes* returns the object. *NumPhyNodes* returns the number of physical query plans in the phynodes. *AddPhyNode* adds a newly created physical query plan that associates with this operator tree to the phynodes. *DeletePhyNode* deletes a physical query plan from the list of phynodes because it is suboptimal and is not used by others.

logprops — a private attribute representing the logical properties of the current operator tree. It is an instance of class OPERATORTREEPROPERTY and contains information such as a set of relations and predicates

involved in this tree, and estimated cardinality of the output, etc.

Figure 32 is an example that shows the logical properties associated with each tree node in an operator tree for the query:

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

Suppose operators DBRelation, Select, and Join are defined in the database. DBRelation represents an operator that has no inputs and serves as leaves in the operator tree and refers to the relations stored in the database. The meanings of the other two operators are straightforward.

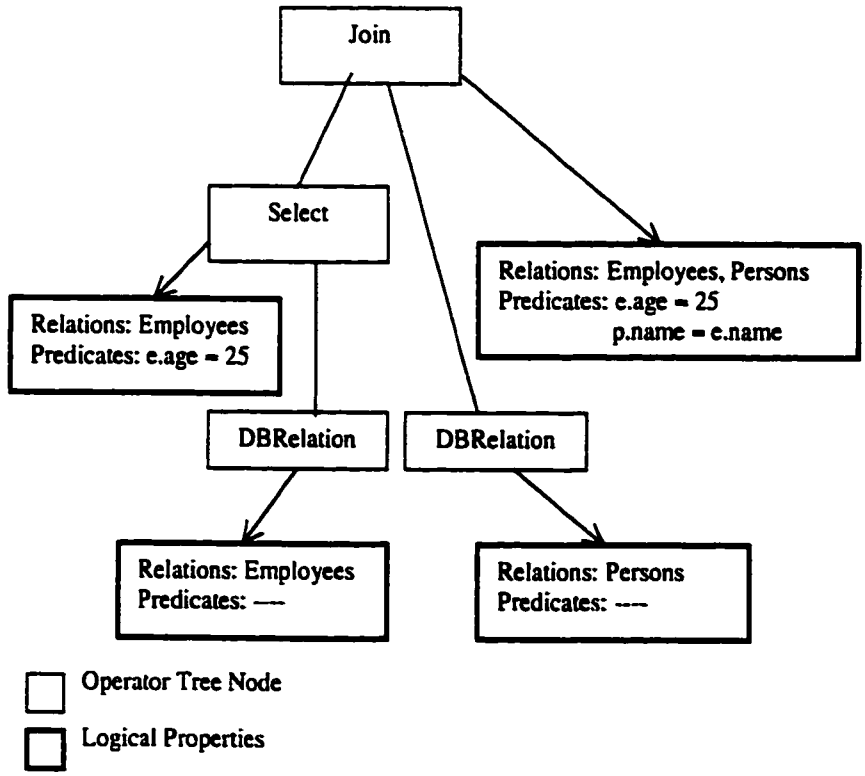

Figure 32: Operator Tree and Logical Properties

Each node in the operator tree has some logical properties. These properties are associated with the tree that rooted at that node. Nodes in the operator tree contain references to their properties. For instance, in Figure 32, the Select tree node represents the part of the operator tree rooted at it. It is actually an operator tree

too. The logical properties of this tree are that: it contains relation Employees and the predicate e.age=25.

There are two methods related to this attribute. *SetLogProps* sets the logical property object of the current tree to the supplied parameter object. *GetLogProps* returns the logical property object logprops.

children — a private attribute representing logical query plans generated from this operator tree. This attribute maintains a list of all the logical query plans that are built from the current tree. There are three methods related to this attribute. *Children* returns the children object. *AddChild* inserts a child into it when a logical query plan is built from the current tree. *DeleteChild* removes a child from it.

dependent_nodes — a private attribute representing a list of logical query plans that must be deleted when the current tree is deleted. Related methods *AddDependent* inserts a dependent logical query plan into it and *DeleteDependent* removes a dependent logical query plan from it.

suboptimal_phynodes — a private attribute representing physical query plans that associated with the current tree and who are sub-optimal but they can not be deleted because they are used as inputs to other physical query plans. Its related method *DeleteSubOptimalPhyNode* deletes a sub-optimal physical query plan if it is no longer used as inputs to other physical query plans.

enforcednodes — a private attribute representing a list of physical query plans that are created by being applying enforcers to the phynodes of the current tree. Its related method *AddEnforcedNode* adds a new physical query plan to it. This new physical query plan is the result of applying an enforcer to one of the phynodes of the current tree.

The OPERATORTREEPROPERTY Class

The OPERATORTREEPROPERTY contains the logical properties of an operator tree (logical query plan). Figure 33 shows its structure.

Major methods and attributes include:

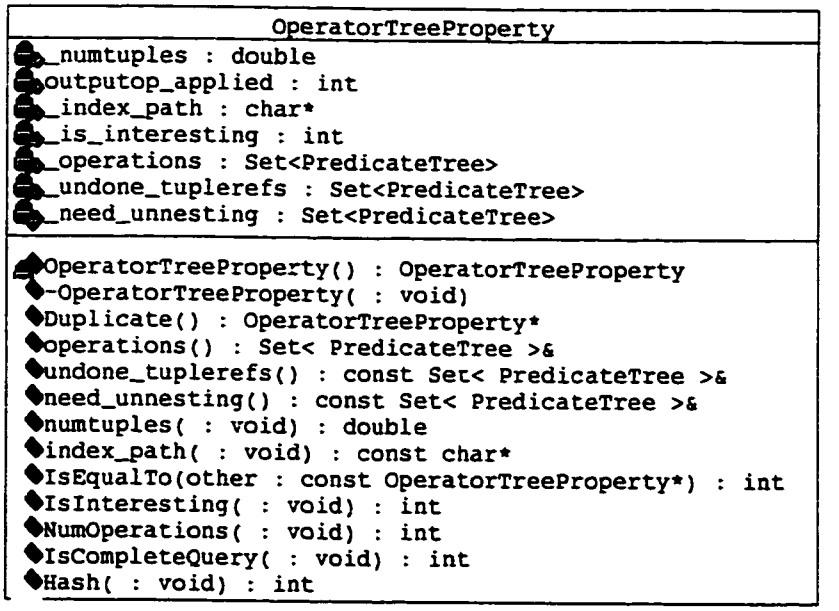

Figure 33: Class OPERATORTREEPROPERTY

IsEqualTo — a public method that compares the logical properties of two operator trees, and returns true if they are the same.

IsCompleteQuery — a public method that returns true if the current operator tree is complete. An operator tree is considered complete when its logical properties contain all the expressions (including predicates) in the query.

IsInteresting — returns true if the operator tree it refers to contains some interesting logical properties. Which logical property is interesting is up to the optimizer-implementor. Normally, if the current operator tree has a non-empty index path name, it is interesting.

Hash — a public method that hashes the current operator tree to a specific number. If two operator trees are deemed to equivalent, their hash numbers should be the same; otherwise, they should not.

_numtuples — a private attribute representing estimated number of tuples (i.e. cardinality) in the output of the operator tree. Its related method *numtuples* returns this attribute.

_operations - - a private attribute representing operations that are applied to the current tree so far. It is accumulated when an operator tree is

built from bottom-up. Related methods include:

NumOperations — returns the number of operations applied so far.

operations — returns this object.

_undone_tuplerefs — a private attribute representing pointer join predicates which should have applied but haven't yet been. It has one related method *undone_tuplerefs* that returns this object.

_need_unnesting — a private attribute representing set-valued attributes that haven't been unnested. It has one related method *undone_tuplerefs* that returns this object.

_index_path — a private attribute holding the index path name which can be used for an index scan. It has one related method *index_path* that returns this object.

The optimizer-implementor can add other logical properties to the class OPERATORTREEPROPERTY or change definitions for interesting properties and equivalent logical query plans. The attributes _undone_tuplerefs and _need_unnesting are defined for uses of pointer joins and set-valued attributes. They do no harm to any query optimization system that does not use them.

The ALGORITHMTREE Class

The ALGORITHMTREE class represents the physical query plan (access plan) in query optimization.

algorithm — a private attribute which is a pointer instance of the class DBALGORITHM and represents that this tree is built by applying it to the tree's inputs. It has one associated method *GetAlgo* that returns this object.

inputs — a private attribute representing the inputs (algorithm trees) of this tree. The current tree is the result of applying the attribute algorithm to the inputs. There are three related methods related to this attribute. *Input* returns the Nth input if N is the provided parameter. *LeftInput* returns the first input. *RightInput* returns the second input.

parent — a private attribute representing a reference to the operator tree that associates with this algorithm tree. In other words, the current algorithm tree is one of the physical query plans of its parent. Its related method *GetParent* returns this object (an instance of the OPERATORTREE class).

phyprops — a private attribute representing the physical properties of the current algorithm tree. It is an instance of class ALGORITHMTREEPROPERTY and contains information such as estimated execution cost, sort-order, etc.

Figure 34 is an example that shows the physical properties associated with each tree node in an algorithm tree for the query:

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

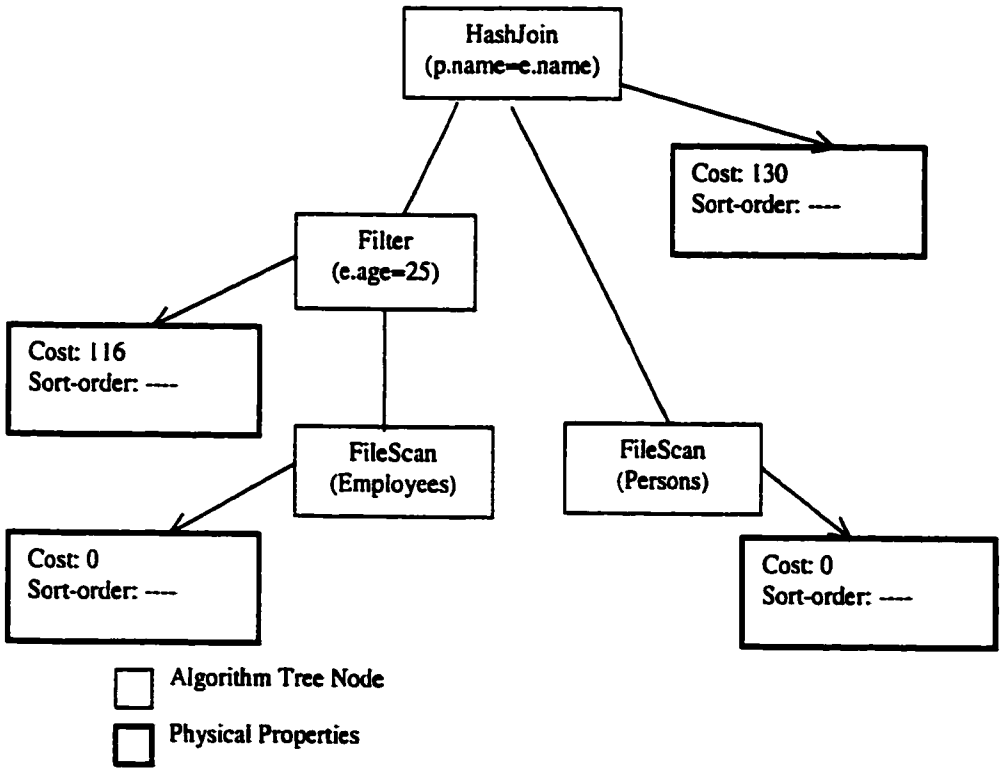

Figure 34: Algorithm Tree and Physical Properties

Suppose execution algorithms FileScan, Filter, and HashJoin are defined in the database. FileScan represents the execution algorithm for the logical operator DBRelation, which is an operator that has no inputs and serves as leaves in the operator tree and refers to a relation stored in the database. Filter represents the execution algorithm for the logical operator Select. HashJoin is one execution algorithm for the logical operator Join.

Each node in the algorithm tree has some physical properties. These properties are associated with the tree that rooted at that node. Nodes in the algorithm tree contain references to their physical properties. For instance, in Figure 34, the Filter tree node represents the part of the algorithm tree rooted at it. It is actually an algorithm tree too. The physical properties of this tree are that its execution cost is 116 and its output is not in any sort order.

The two methods related to attribute phyprops are *SetPhyProps* and *GetPhyProps*. *SetPhyProps* sets the physical property object of the current tree to the supplied parameter object. *GetPhyProps* returns the physical property object phyprops.

suboptimal — a private boolean attribute whose value is true if the current algorithm tree is sub-optimal because it has higher estimated execution cost than its equivalents but it can not be deleted because it is used as an input to other physical query plans. An algorithm tree that is marked as sub-optimal should not be considered as input for other algorithm trees. Its relate method *SubOptimal* return the value of the suboptimal.

enforcednode — a private boolean attribute whose value is true if the current algorithm tree is the result of the application of an enforcer. Its related method *IsEnforcedNode* returns the value of the enforcednode.

The ALGORITHMTREEPROPERTY Class

The ALGORITHMTREEPROPERTY contains the physical properties for an algorithm tree (physical query plan). Figure 35 shows its structure.

Major methods and attributes include:

IsEqualTo— a public method that compares the physical properties of two algorithm trees, and returns true if they are the same.

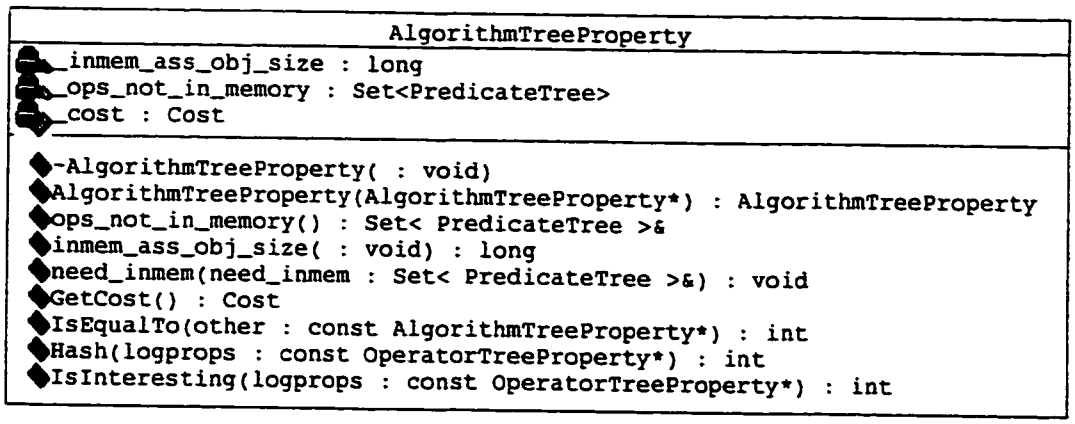

Figure 35: Class ALGORITHMTREEPROPERTY

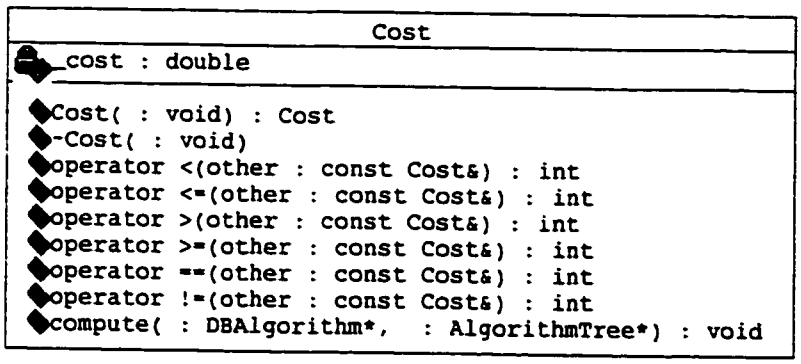

Figure 36: Class COST

IsInteresting — returns true if the algorithm tree it refers to contains some interesting physical properties. Which physical property is interesting is up to the optimizer-implementor. Normally, if the result of the current algorithm tree is in sort order, it is interesting.

Hash — a public method that hashes the current algorithm tree to a specific number. If two algorithm trees are deemed to equivalent, their hash numbers should be the same; otherwise, they should not.

_cost — a private attribute that is an object of class COST and that contains the estimated execution cost for the current algorithm tree. It has an associated method *GetCost*, which returns this object.

Figure 36 shows the structure for the class COST. The COST class has an attribute _cost that contains the estimated execution cost value. It also defines a couple of methods to compute the cost values and compare the cost values between two COST

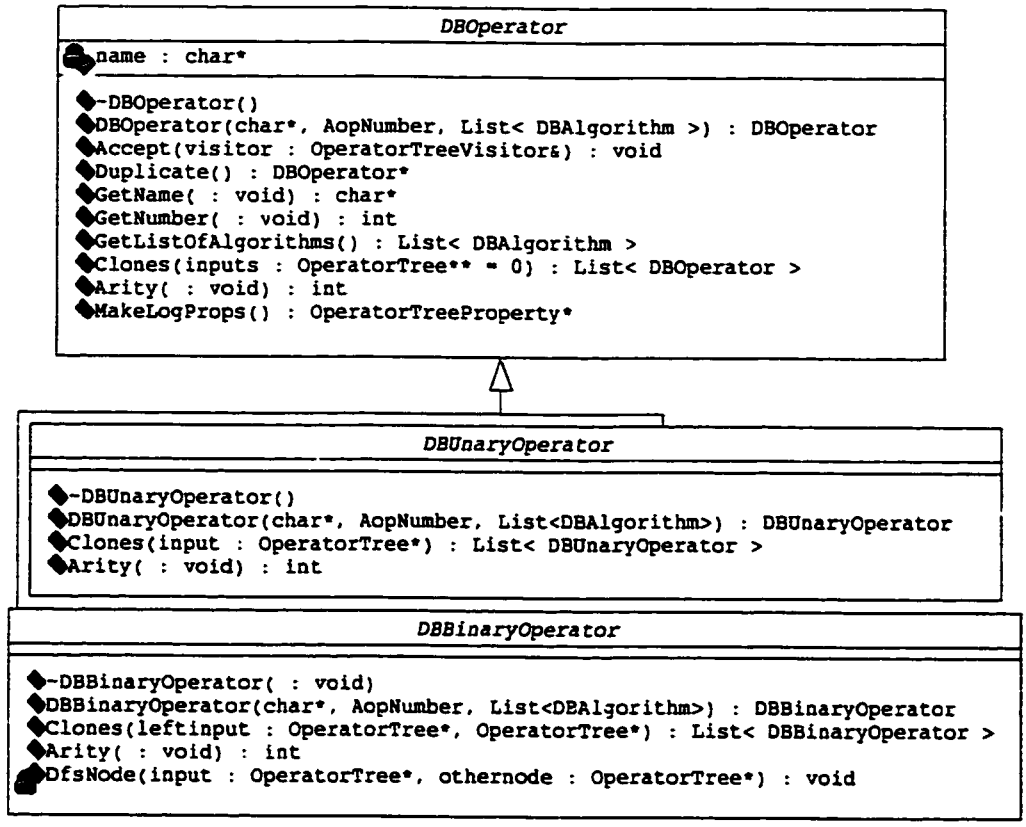

Figure 37: The Operator Hierarchy

objects.

The optimizer-implementor can add other physical properties to the class ALGORITHMTREEPROPERTY or change definitions for interesting properties and equivalent physical query plans.

4.3 The Algebra Component

4.3.1 Structure

Figure 37, Figure 38, and Figure 39 show the class diagrams in the Algebra component. This component is composed of the logical operators and the physical operators in the database. The DBOPERATOR class is the representative of the logical operators. Operators such as Select and Join are common logical operators that are found in the relational database. Each node in an operator tree is created by applying a logical operator to its input trees. The DBALGORITHM class is the representative of

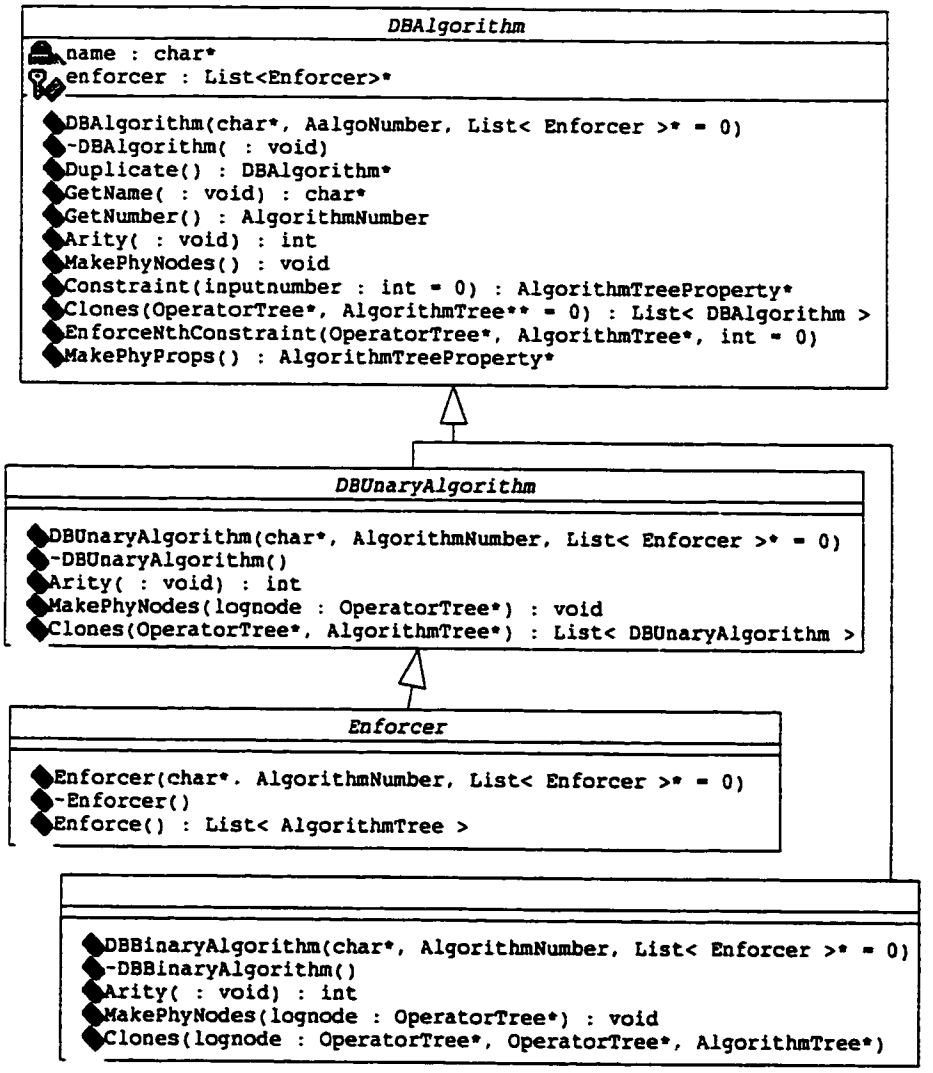

Figure 38: The Algorithm Hierarchy

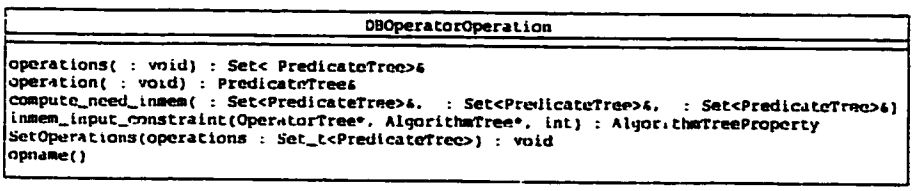

Figure 39: The DBOperatorOperation Class

the physical operators. Operators such as File Scan, Index Scan, and Nested Loop Join are common physical operators that are found in the relational database. Each node in the algorithm tree is created by applying a physical operator to its input trees.

Each logical operator can have more than one execution algorithm associated with it. For example, for the Join operator in the relational database, there are several algorithms associated with it such as Nested Loop Join, Merge Join, Hash Join, and so on. An algorithm tree can be built by replacing the logical operators in an operator tree with their associated physical operators. Since one logical operator may have more than one corresponding physical operator, an operator tree may have more than one algorithm tree associated with it.

There is a weak dependency between the logical operator and the physical operator. Any logical operator needs to keep track of a set of physical operators associated with it. So when a new operator tree is built, these associated physical operators are accessed and used to build the corresponding algorithm trees.

Since operators may differ in their arity and same as algorithms, the classes DBUnaryOperator, DBBinaryOperator, DBUnaryAlgorithm, and DBBinaryAlgorithm are defined to represent unary logical operators, binary logical operators, unary physical operators, and binary physical operators respectively.

4.3.2 Major Design Issue and Design Decision

The major design issue in the design of the Algebra component is that logical and physical operators are basic elements making up of logical and physical query plans that are used in the search. The query optimization is actually performed on the database algebra. That means in query optimization the search strategy is inherently non-extensible with respect to the database algebra. The question is: How can the search strategy perform without any knowledge of the database algebra in the query optimization framework?

OPT++ [20] solves this problem by making specific assumptions about the kinds of manipulations that are allowed on the operator trees and the algorithm trees. The design idea is to define abstract classes for the logical operator and the physical operator, put assumptions of the kinds of operations as abstract methods in these abstract classes, and defer the implementation to their concrete subclasses that will

be defined by the optimizer-implementor. With the inheritance and run time binding in the C++, the actual logical and physical operators of the database algebra are called at run time.

We reuse the design and code of the Algebra component in OPT++ and make some improvement to them. Kabra claims that he separates the Algebra and the Search Space in his design [20]. But in his implementation code, these two components are actually coupled. We solve this problem by introducing an *Accept* method in the logical operator hierarchy. Operations on the database algebra are represented as visitors and are moved to the Search Space component. Then the database algebra can perform any operation on it by accepting an appropriate visitor. The net effect is that the Algebra component is only the representation of the database algebra, while how the elements in the algebra are used to build what shapes of operator trees and algorithm trees is the responsibility of the Search Space component. We thus physically separate the Algebra component and the Search Space component and allow the Search Space to be experimented with different implementations without affecting the Algebra component.

Logical and physical operators in the Algebra are further divided into unary and binary. There are some slight differences in behavior between the unary operator and the binary operator. This division allows common code to share by the concrete logical and physical operators if they belong to the same category.

4.3.3 Class Descriptions

The DBOperator Class

The DBOperator class is the base class in the logical operator hierarchy. It is an abstract class and defines three abstract methods:

Arity — a logical operator can be unary, binary, etc.

Duplicate — enables a logical operator to copy itself. It provides an easy way to initialize a newly created logical operator. For example, the set of execution algorithms associated with a logical operator is kept unchanged when copy.

MakeLogProps — sets up the logical properties. The logical properties associated with a logical operator may include the set of relations and

the predicates used.

Other major methods defined in the class DBOPERATOR include:

Accept — an important method that makes the logical operator structure flexible. By giving an argument visitor that represents one kind of operations on this logical operator, this method allows various operations to be performed on itself without changing any definition in this class.

GetListOfAlgorithms — returns a list of execution algorithms associated with this logical operator. Execution algorithms that associated with this operator are defined by the optimizer-implementor, and they are initialized when the system starts.

Clones — an important method to produce a list of operator instances representing different ways of applying this operator to the inputs with different parameters. The default implementation is one way to apply this operator. The optimizer-implementor can rewrite this method if needed.

Each logical operator has a name (attribute) and a number (an enumeration type). The number associated with a logical operator is initialized when the system starts. The associated methods with these attributes are *GetName* and *GetNumber*, which return the name and number for this logical operator respectively.

The DBUNARYOPERATOR Class

The DBUNARYOPERATOR is a subclass of the DBOPERATOR class. It is also an abstract class. It only implements the method *Arity* with a return value one and leaves the implementation of methods *Duplicate* and *MakeLogProps* to its concrete unary logical operator classes.

The DBBINARYOPERATOR Class

The DBBINARYOPERATOR is a subclass of the DBOPERATOR class. It is also an abstract class. It only implements the method *Arity* with a return value two and leaves the implementation of methods *Duplicate* and *MakeLogProps* to its concrete binary logical operator classes.

The DBALGORITHM Class

The DBALGORITHM class is the base class in the physical operator hierarchy. It is an abstract class and defines four abstract methods:

Arity — a physical operator can be unary, binary, etc.

Duplicate — enables a physical operator to copy itself. It provides an easy way to initialize a newly created physical operator. For example, the name and the number associated with a physical operator are kept unchanged when copy.

MakePhyProps — sets up the physical properties for the algorithm tree rooted at this physical operator. These physical properties may include the estimated execution cost, sort order, etc.

MakePhyNodes — creates all physical query plans as a result of applying this algorithm instance to its input operator trees.

Other major methods defined in the class DBOPERATOR include:

Constraint — each algorithm has constraints on its inputs. This method takes an algorithm tree that will serve as the Nth input as input, and returns the physical properties that are required for that tree if it is to serve as the Nth input.

Clones — an important method to produce a list of algorithm instances representing different ways of applying this algorithm to its inputs with different parameters. The default implementation is one way to apply this algorithm. The optimizer-implementor can rewrite this method if needed.

EnforceNthConstraint — makes sure that the Nth input satisfies the required constraints. It first calls method *Constraint* to find out what physical properties are required. It then calls each of the enforcers associated with the Nth input and applies them to the Nth input tree. Finally it returns a list of algorithm instances which might result from the application of these enforcers to the Nth input tree. The resultants will serve as the actual inputs of this algorithm.

Each physical operator has a name (attribute) and a number (an enumeration type). The number associated with a physical operator is initialized when the system starts. The associated methods with these attributes are *GetName* and *GetNumber*, which return the name and number for this physical operator respectively.

The DBUnaryAlgorithm Class

The DBUnaryAlgorithm is a subclass of the DBAlgorithm class. It is also an abstract class. It implements the method *Arity* with a return value one and the method *MakePhyNods* by delegating it to the Search Space, and leaves the implementation of the method *MakePhyProps* to its concrete unary physical operator classes. In addition, it overrides the method *Clones* to perform some operations that relate to a unary operator.

The DBBinaryAlgorithm Class

The DBBinaryAlgorithm is a subclass of the DBAlgorithm class. It is also an abstract class. It implements the method *Arity* with a return value two and the method *MakePhyNods* by delegating it to the Search Space, and leaves the implementation of the method *MakePhyProps* to its concrete binary physical operator classes. In addition, it overrides the method *Clones* to perform some operations that relate to a binary operator.

The Enforcer Class

A database system might have special execution algorithms that do not correspond to any operator in the logical algebra. The purpose of these algorithms is not to perform any logical data manipulation but to enforce physical properties in their outputs that are required for subsequent query processing algorithms. We call these algorithms enforcers [20]. An example of the enforcers is the sort algorithm. It can be used to ensure the inputs of the merge join are sorted on the join attributes.

The Enforcer class is defined to represent enforcers. It is an abstract class and only defines one abstract method *Enforce*, which takes an algorithm tree and proposed physical properties as inputs and outputs a list of algorithm trees that are results by forcing the proposed physical properties to be added to the input algorithm tree.

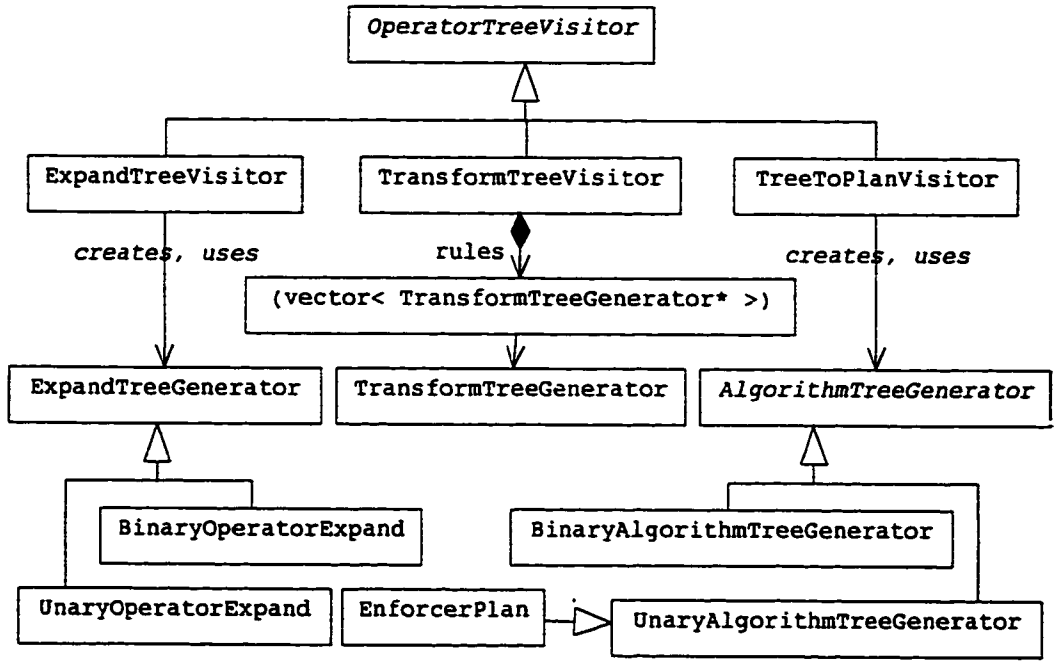

Figure 40: The Search Space Component

4.4 The Search Space Component

4.4.1 Structure

Figure 40 shows the main class diagram in the Search Space component. The class details are shown in Figure 41 and Figure 42. There are two hierarchies in this component. One is the visitor hierarchy, the other is the generator hierarchy.

The visitor hierarchy performs operations on the logical algebra. The root class OPERATORTREEVISITOR is an abstract class and is used in places where polymorphism is needed. The EXPANDTREEVISITOR subclass is defined to expand an operator tree in a bottom-up fashion. The TRANSFORMTREEVISITOR subclass is designed to transform an operator tree to its equivalents. Two operator trees are equivalent if all of their logical properties are the same. Relations and predicates are two examples of the logical properties of an operator tree. The TREETOPLANVISITOR subclass is designed to convert an operator tree to its corresponding algorithm trees. One operator tree may have more than one corresponding algorithm tree if any operator in this operator tree has more than one execution algorithm. The visitor classes dispatch their responsibility to the corresponding generator classes.

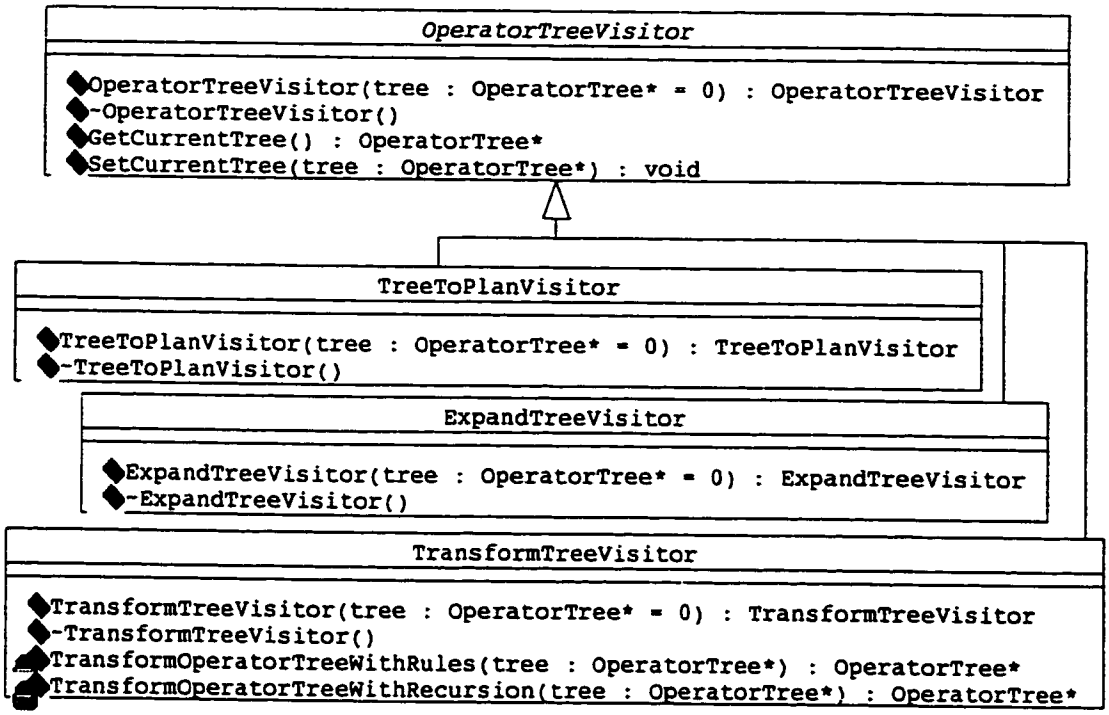

Figure 41: The Visitor Hierarchy

The generator hierarchy implements the major functionality in the Search Space component. There are three generator classes, each is created and used by a corresponding OPERATORTREEVISITOR subclass. How the search space is shaped in query optimization depends on how these generator classes are implemented.

4.4.2 Design Patterns Used

There is one design pattern used in the Search Space component. That is the Visitor design pattern. Figure 43 shows its standard structure [5].

In our design, the OPERATORTREEVISITOR plays the role of Visitor in the standard structure. EXPANDTREEVISITOR, TRANSFORMTREEVISITOR, TREETOPLANVISITOR play the role of ConcreteVisitor. Logical algebra plays the role of ObjectStructure. In addition, the DBOPERATOR, DBUNARYOPERATOR, and DBBINARYOPERATOR play the role of Element. Any concrete logical operator defined by the optimizer-implementor plays the role of ConcreteElement of the standard structure.

The use of the Visitor design pattern in this framework has significant effect on

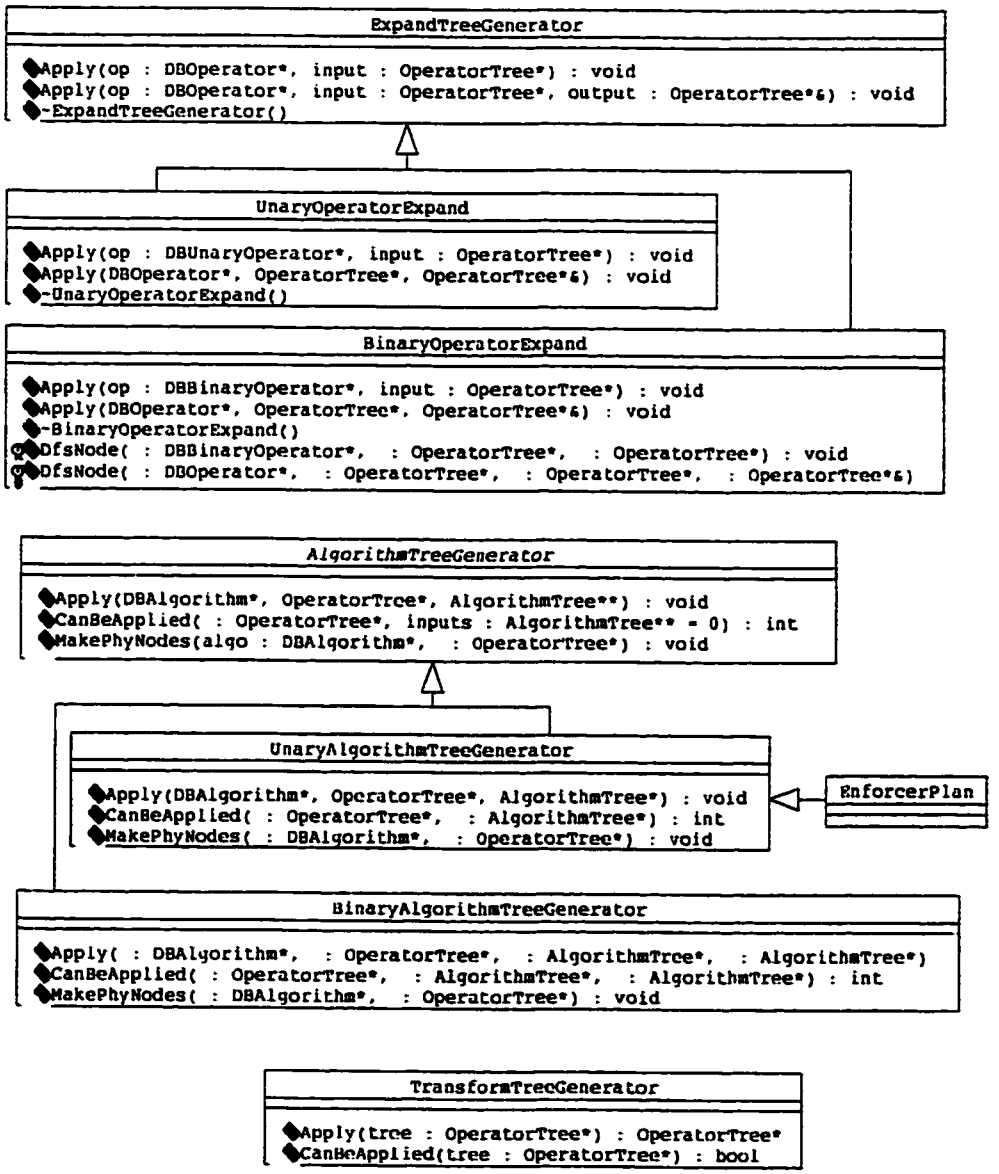

Figure 42: The Generator Hierarchy

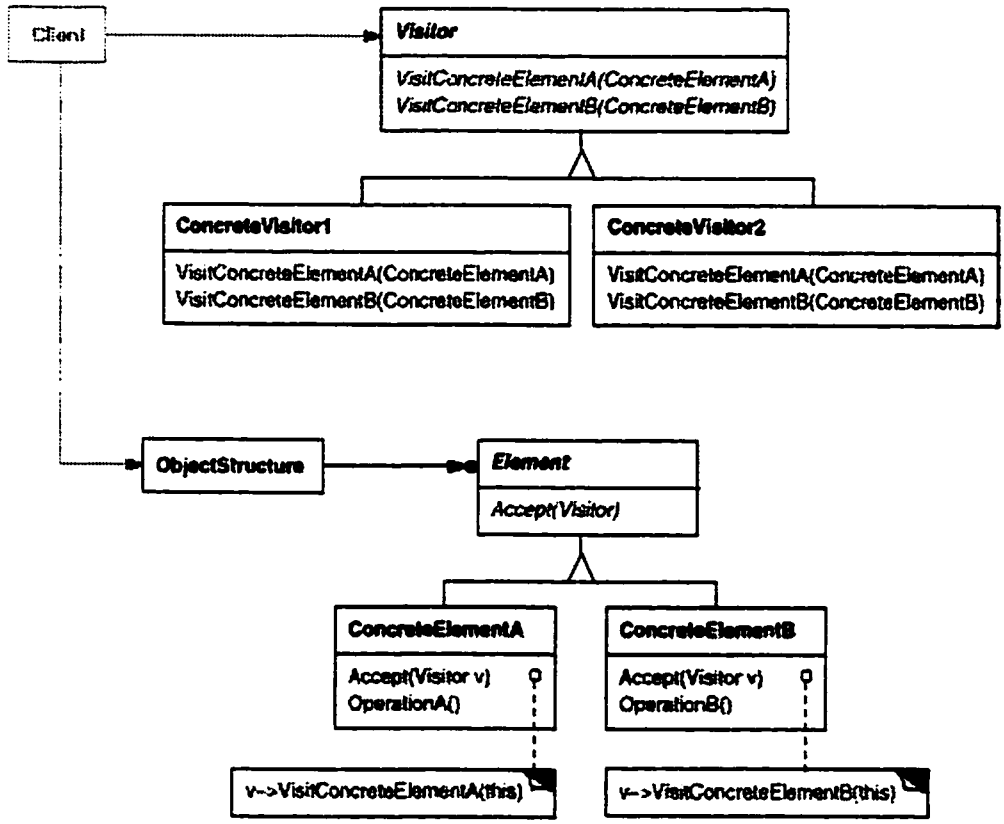

Figure 43: Visitor Design Pattern

the extensibility of the query optimization system. The interface of the logical algebra is relatively stable because what specific algebra to use is known by the optimizer-implementor or it can be defined by making specific assumptions on the kinds of operations that are allowed to perform as that in OPT++. But how the search space is shaped in query optimization is subject to change because that is still a research topic. The use of the Visitor design pattern helps to move the volatile implementation of the operations on the logical algebra far from it and allows algorithms to be experimented in the Search Space independently. Additional operations that may perform on the logical algebra can be easily added at any time without the need to re-compile the Algebra component.

4.4.3 The Need for the Generator Hierarchy

A tricky question is: Why introduce the generator hierarchy?

According to the design of the Visitor design pattern, the visitors not only represent the operations on the object structure, they actually implement these operations. While in our design, the visitors invoke appropriate generators and delegate their implementation to them.

Of course, without the generators, we can put the implementation code for the operations inside the visitors, and the design still works. But since the visitors represent operations that perform on the logical algebra, for each logical operator XX, there must be a *VisitXX* method to be defined in each visitor class. In a complex algebra system, there will be a long list of *VisitXX* methods needed to be defined in the Search Space component. While at the same time, the implementation of each of these methods is also complex — it truly is! Obviously such a complex structure is hard to understand and maintain. The advantages of introducing the generator classes and moving the implementation of the operations that a normal visitor needs to these generator classes are:

- Both the visitor hierarchy and the generator hierarchy are simple to handle. The visitor classes may have complex interfaces (due to a long list of *Visit* methods), but the implementation of these interfaces is simple because they simply delegate to appropriate generators. The generator classes have complicated implementation but their interfaces are simple because each concrete generator class only implements one operation that may perform on a logical

operator. So the classes in the Search Space component have a well-balanced responsibility.

- We follow the design convention to separate interface from implementation. The visitor classes define interfaces of these operations and the generator classes implement these interfaces. On one hand, it promotes information hiding. On the other hand, it protects the client code from changing while allows different implementations to be experimented.

Generators in the Search Space component are further divided into unary and binary. There are some slight differences in behavior between the unary generator and the binary generator. The division allows common code to share by the concrete generators if they belong to the same category.

4.4.4 Class Descriptions

The Visitor Class Hierarchy

The OPERATORTREEVISITOR is the root class in the visitor hierarchy. It is partially designed in the query optimization framework. It defines an attribute currentTree that represents the current operator tree to be manipulated and avoids the overhead of parameter passing. This attribute has two associated methods: *GetCurrentTree* returns the current operator tree especially after tree manipulation. *SetCurrentTree* resets the current operator tree to the parameter one. The OPERATORTREEVISITOR is an abstract class and leaves a series of the *VisitXX* methods to be defined by the optimizer-implementor. XX refers to a logical operator in the database algebra. These methods should be defined as abstract and leaves their implementation to the concrete subclasses.

The EXPANDTREEVISITOR is a subclass of the OPERATORTREEVISITOR. It is used to expand an operator tree in a bottom-up fashion. This class is very useful in the Bottom-Up Search Strategy. It is also used by the Transformative Search Strategy to build an initial operator tree from bottom-up. This class must implement a series of the *VisitXX* methods and simply dispatch implementation to appropriate concrete generator classes.

The TRANSFORMTREEVISITOR is a subclass of the OPERATORTREEVISITOR. It is used to transform an operator tree to its equivalents. Two operator trees are

equivalent if all of their logical properties are the same. This class is very useful in the Transformative Search Strategy to transform an operator tree to another using algebraic laws. This class must implement a series of the *VisitXX* methods and simply dispatch implementation to appropriate concrete generator classes.

The TREETOPLANVISITOR is a subclass of the OPERATORTREEVISITOR. It is used to convert an operator tree (logical query plan) to an algorithm tree (physical query plan). This class is very useful in all search strategies because the conversion from an operator tree to the corresponding algorithm trees is fundamental in query optimization. This class must implement a series of the *VisitXX* methods and also simply dispatch implementation to appropriate concrete generator classes.

The Generator Hierarchy

The generator hierarchy does not have a common root class, and it is unnecessary to define one. Instead, we define three top-level generator classes, each corresponds to one immediate subclass of the OPERATORTREEVISITOR class.

The first top-level generator class is the EXPANDTREEGENERATOR class. It is designed for tree expansion. This class defines a major method *Apply* that is used to create operator trees corresponding to the application of the given logical operator to (all possible) sets of inputs, one of which is the given operator tree.

The UNARYOPERATOREXPAND is a subclass of the EXPANDTREEGENERATOR. It overrides the *Apply* method to apply the given logical operator to the given operator tree to create one new operator tree. It represents the common behavior in all unary logical operators.

The BINARYOPERATOREXPAND a subclass of the EXPANDTREEGENERATOR. It overrides the *Apply* method to apply the given logical operator to the given operator tree to create new operator trees. The given operator tree will serve as one of the inputs for the given operator. The other inputs will have to be found in the search tree. This class also defines a private method *DfsNode* that is used by the *Apply* to perform a depth first search of the search tree for suitable nodes that can be paired with the given operator tree input. The BINARYOPERATOREXPAND class defines methods representing the common behavior in all binary logical operators.

The second top-level generator class is the TRANSFORMTREEGENERATOR class. It is designed for equivalent tree transformation. This class defines two major methods:

CanBeApplied — returns true if the current generator can apply to the given operator tree.

Apply — applies the current generator to the given operator tree and restructures the given operator tree into a new operator tree. The new operator tree must be equivalent to the given operator tree.

The optimizer-implementor has to define concrete subclasses of the TRANSFORMTREEGENERATOR, each represents an algebraic law that performs on some logical operators. The newly created subclasses must overwrite the default behaviors of the methods *CanBeApplied* and *Apply* in the base class.

The third top-level generator class is the ALGORITHMTREEGENERATOR class. It is designed to convert an operator tree to its algorithm trees. It is an abstract class and defines three methods:

MakePhyNodes — an abstract method that is supposed to control the creation of the algorithm trees by applying the given physical operator to the given algorithm tree inputs. Only non sub-optimal inputs are considered to be used to build the new trees. The implementation of this method is left to the concrete subclasses.

CanBeApplied — returns true if the given physical operator can apply to the given algorithm trees.

Apply — constructs all algorithm trees for the given operator tree by applying the given physical operator to the given algorithm trees.

The UNARYALGORITHMTREEGENERATOR and the BINARYALGORITHMTREEGENERATOR are two subclasses of the ALGORITHMTREEGENERATOR. In these two classes, method *MakePhyNodes* is implemented and methods *CanBeApplied* and *Apply* are overridden. These two classes only differ in the way the given physical operator applies to the given inputs. In the class UNARYALGORITHMTREEGENERATOR, a unary physical operator is applied. While in the class BINARYALGORITHMTREEGENERATOR, a binary physical operator is applied.

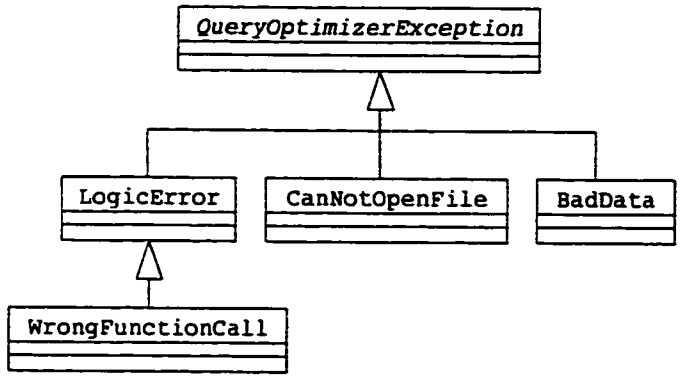

Figure 44: Class Diagram for the Exception Handling

4.5 Exception Handling

An exception handler can centralize the processing of all exceptions no matter where they occur. It is difficult to incorporate an exception handler into a system if the system has been completed. Although a query optimization system is not intended to be directly interact with the user and the exceptions that may occur are not significant, we still consider it necessary to incorporate an exception handling mechanism.

Figure 44 shows the exception handling hierarchy in the query optimization framework. The root class of the exception hierarchy is the QUERYOPTIMIZEREXCEPTION class. It is an abstract class and defines one public method *What* refers to what the exception is. Since the root class does not know what exception it will be, it defers that to its concrete subclasses. The advantage of using a common root exception class is that a polymorphic exception variable can be caught and be processed according to its run-time type. For example, a C++ statement *catch (QueryOptimizerException& e)* can be placed in the main control program and a call to *e.What()* will display what specific exception is. If e is a logic error, then *e.What()* will display this logic error. If e is an error that a file can not be opened, then *e.What()* will display the name of the file which could not be opened.

The LOGICERROR is a subclass of the QUERYOPTIMIZEREXCEPTION that is used to refer to logic errors in the system. It defines a private attribute errorMessage that records what logic error it is. The *What* method is implemented to display this logic error message. These logic errors are not intended to be understood by the user of this system. Instead, they are designed to help the system maintainer to fix the system errors. Wrong function call is one kind of logic error and class

WRONGFUNCTIONCALL is defined for this purpose. It refers to the kind of errors that a function should not be called but it is mistakely called. The *What* method is overridden to display which function should not be called.

The BADDATA is a subclass of the QUERYOPTIMIZEREXCEPTION that is used to refer to errors that data is not in the right format and could not be processed. A private attribute errorMessage is defined to record this kind of errors. The *What* method is implemented to return the error message.

The CANNOTOPENFILE is another subclass of the QUERYOPTIMIZEREXCEPTION that is used to refer to the common I/O errors: "Can not open files". A private attribute filename is defined to records the name of the file that can not be opened, and the *What* method is implemented to return the error message.

4.6 Dynamic Behavior

This section shows the dynamic behaviors of the query optimization system.

4.6.1 Scenario of Invoking Query Optimization

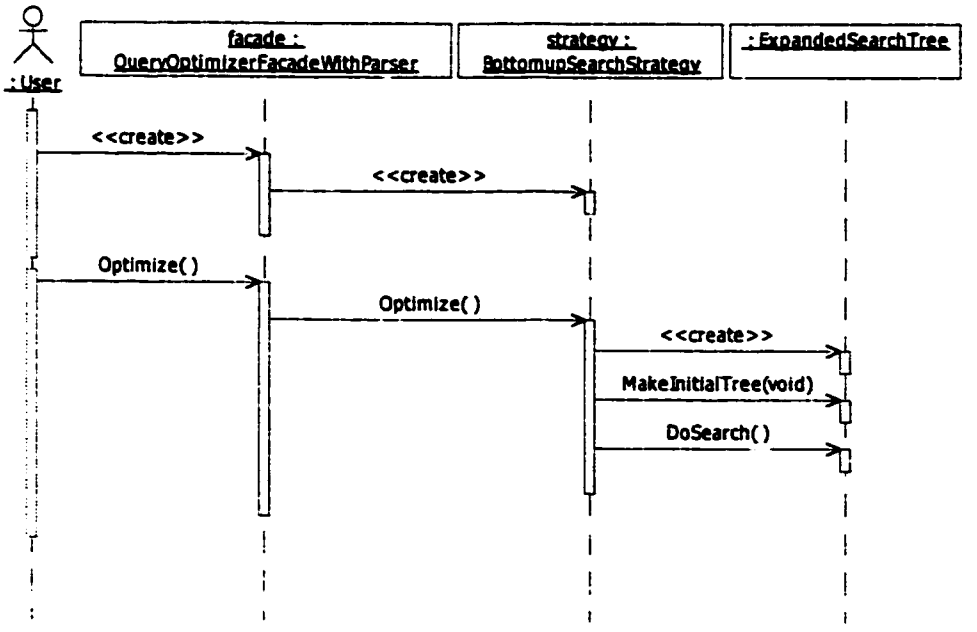

Figure 45: Sequence Diagram for Bottom-Up Optimization

Figure 45 shows one scenario of how the user interacts with the system and how the system reacts to the user calls. It demonstrates the default view of this system, that is query optimization from bottom-up. This scenario illustrates:

1. The user creates a QUERYOPTIMIZERFACADEWITHPARSER object facade.
2. A BOTTOMUPSEARCHSTRATEGY object is created and will be used for search.
3. The user invokes method *Optimize* on the facade object.
4. The facade delegates the request to the BOTTOMUPSEARCHSTRATEGY object.
5. The BOTTOMUPSEARCHSTRATEGY object creates an EXPANDEDSEARCHTREE object search tree which will be used to perform search.
6. The *MakeInitialTree* is called on the search tree. Initial logical query plans are created according to the relations involved in the query. Their corresponding physical query plans are also created.
7. The *DoSearch* is called on the search tree. Logical query plans are expanded continuously. For each logical query plan, all its corresponding physical query plans are also created, execution costs of these physical plans are estimated, and cost pruning is performed. Equivalent plans that have higher estimated costs are pruned out.
8. Repeats step 7 until no logical query plans can be further expanded.
9. The physical query plan that relates to the complete query and that has the least cost is returned as a result of *Optimize*.

4.6.2 Scenario of Invoking Transformation Query Optimization

Figure 46 shows another scenario of how the user interacts with the system and how the system reacts to the user calls. It demonstrates another view of this system, that is query optimization based on transformation. This scenario illustrates:

1. The user creates a QUERYOPTIMIZERFACADEWITHPARSER object facade.

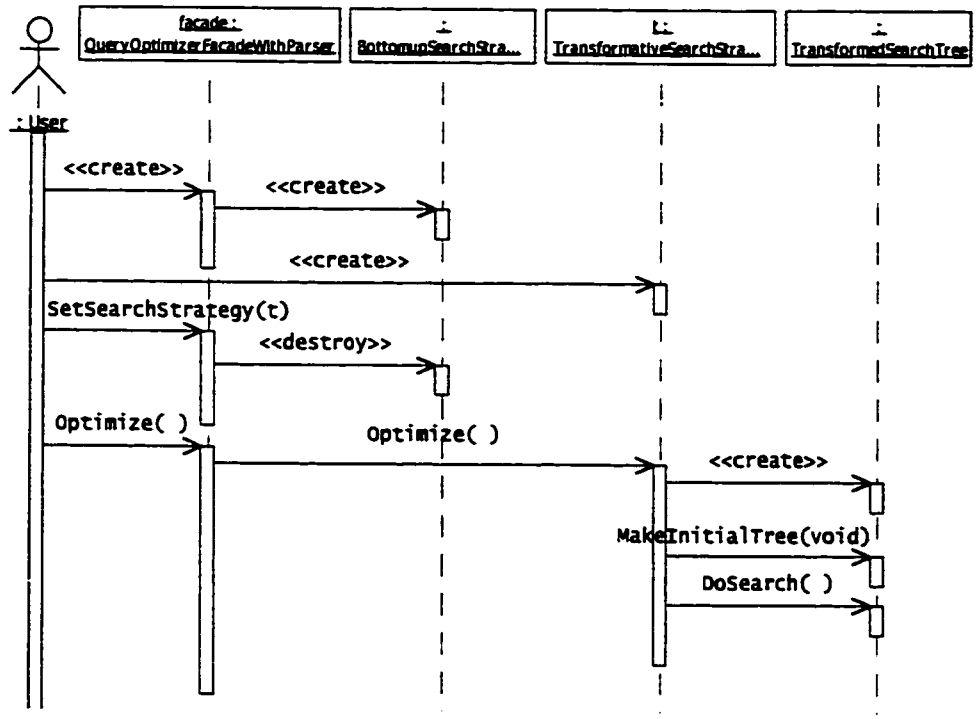

Figure 46: Sequence Diagram for Transformation Optimization

2. A BOTTOMUPSEARCHSTRATEGY object is created and will be used for search.

3. The user creates a TRANSFORMATIVESEARCHSTRATEGY object.

4. The user invokes the *SetSearchStrategy* on the facade object. The default BOTTOMUPSEARCHSTRATEGY object is substituted by the TRANSFORMATIVESEARCHSTRATEGY object, which will be used for search.

5. The user invokes method *Optimize* on the facade object.

6. The facade object delegates the request to the TRANSFORMATIVESEARCHSTRATEGY object.

7. The TRANSFORMATIVESEARCHSTRATEGY object instantiates a search tree object with type TRANSFORMEDSEARCHTREE which will be used to perform search.

8. The *MakeInitialTree* is called on the search tree. An initial logical query plan that represents the complete query is created. All its corresponding physical

query plans are also created.

9. The *DoSearch* is called on the search tree. The current logical query plan is transformed into equivalent logical query plans using pre-defined algebraic laws. For each logical query plan, all its corresponding physical query plans are also created, execution costs of these physical plans are estimated, and cost pruning is performed. Equivalent plans that have higher estimated costs are pruned out.

10. Repeats step 9 until no logical query plans can be transformed with pre-defined algebraic laws.

11. The physical query plan that has the least cost is returned as a result of *Optimize*.

4.6.3 The Search Strategies

Two search strategies that are used for search are defined in the query optimization framework: One is the Bottom-Up Search Strategy, the other is the Transformative Search Strategy. Please refer to Section 2.4.1.3 and Section 2.4.2.2 for the descriptions of the algorithms Bottom-Up Search Strategy and Transformative Search Strategy, respectively.

4.6.4 Scenario for Bottom-Up Optimization

Figure 47 shows the scenario of optimizing the following query from bottom-up :

select e.name, p.age **from** Persons p, Employees e **where** e.name=p.name;

Suppose the database algebra defines three logical operators: DBRelation, Select, and Join. DBRelation represents an operator that has no inputs and serves as a leaf in the operator tree and refers to a relation stored in the database. The meanings of the other two operators are straightforward. Also suppose for each logical operator, there is a physical operator associated with it, which represents the execution algorithm for this logical operator. There are three physical operators defined in the database algebra:

File Scan -- an execution algorithm for the DBRelation operator.

Filter — an execution algorithm for the Select operator.

Hash Join — an execution algorithm for the Join operator.

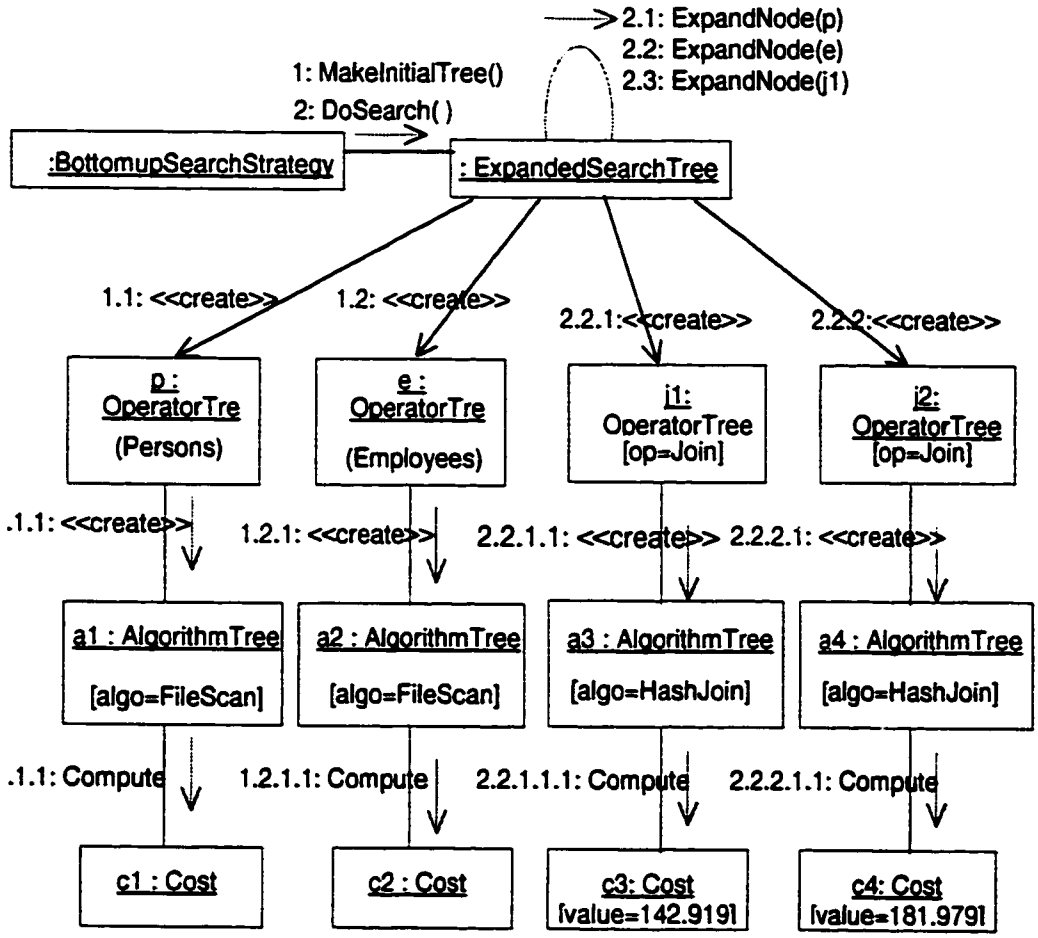

Figure 47: Collaboration Diagram for Bottom-Up Optimization

The bottom-Up optimization begins with a call of method *MakeInitialTree* on the EXPANDEDSEARCHTREE, creating initial logical query plans according to the relations involved in the query. In Figure 47, two initial logical query plans p and e are generated representing relations Persons and Employees respectively. When a logical query plan is created, all its physical query plans must be generated accordingly. Objects a1 and a2 are corresponding physical query plans of p and e. One important thing in building a physical query plan is to compute its execution cost. Objects c1 and c2 are estimated execution costs for physical query plans a1 and a2. Since there are no equivalent physical plans for them, they are kept for further optimization.

DoSearch is then called on the EXPANDEDSEARCHTREE to perform the search based on the initial plans. Logical query plan p is expanded first. Since at this time the plan e has not been processed, e is not visible to the tree expansion process. Plan p is bypassed (since no logical operators can apply to it) and stored away for future use. Logical query plan e is then expanded. Join operator is applied to both logical query plans p and e, and two new logical query plans j1 and j2 are created. Their structures are shown in Figure 48.

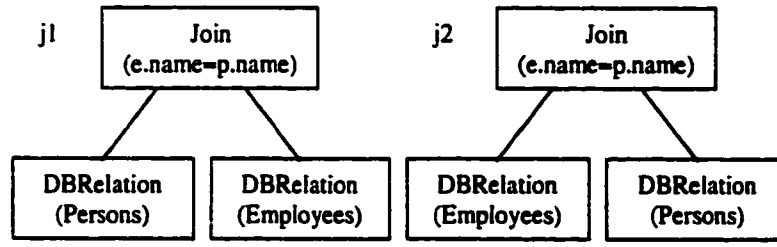

Figure 48: Operator Trees j1 and j2

All corresponding physical query plans for j1 and j2 are created. Since each operator has only one algorithm associated with it, there is one physical query plan for j1 and j2 respectively. They are a3 and a4 in Figure 47. Their structures are shown in Figure 49.

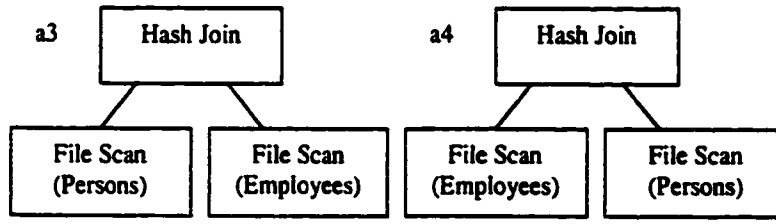

Figure 49: Algorithm Trees a3 and a4

Execution costs for physical query plan a3 and a4 are computed. They are c3 and c4 in Figure 47. Since j1 and j2 are equivalent (Both represents the complete query) and a4 is more expensive than a3, a4 is pruned out. Plan a3 is kept for further optimization.

Logical query plan j1 represents the complete query and it could not be further expanded. Its physical query plan a3 is returned as the optimal plan with least cost.

4.6.5 Scenario for Transformation Optimization

We make the same assumptions about the database algebra as that in Section 4.6.4. For simplicity, we only define one algebraic law Select Push Down in tree transformation in the following example. Select Push Down means putting the Select operator as close as possible to the leaves in an operator tree where its associated relations reside. It is a typical algebra law that is frequently used in the transformation optimization. The advantage of this rule is that it greatly reduces the cardinality of its output thus reducing the execution cost for further processing if its output will serve as some inputs in later processing.

Figure 50(a) shows the initial logical query plan to be transformed. Figure 50(b) shows the logical query plan after Select Push Down has performed on tree Figure 50(a).

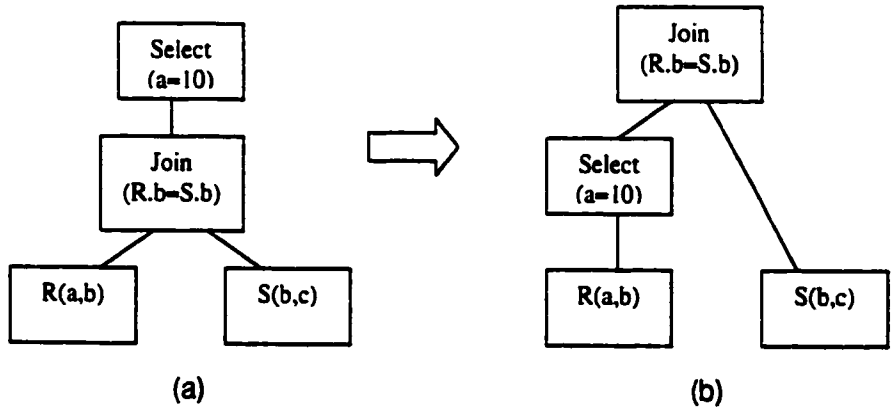

Figure 50: Source Operator Tree and Destination Operator Tree

Figure 51 shows the scenario of transforming the logical query plan in Figure 50(a) to Figure 50(b). The transformation optimization begins with a call of method *MakeInitialTree* on the TRANSFORMEDSEARCHTREE, creating an initial logical plan t1 that represents the complete query. A logical query plan is considered to be complete if it encompasses all the expressions (including predicates) in the query. Logical query plan t1 has a structure like that in Figure 50(a). When a logical query plan is created, all its physical query plans are created accordingly. There is only one physical query plan p1 for t1 in Figure 51 because each operator in t1 has only one execution algorithm. One important thing in building a physical query plan is to

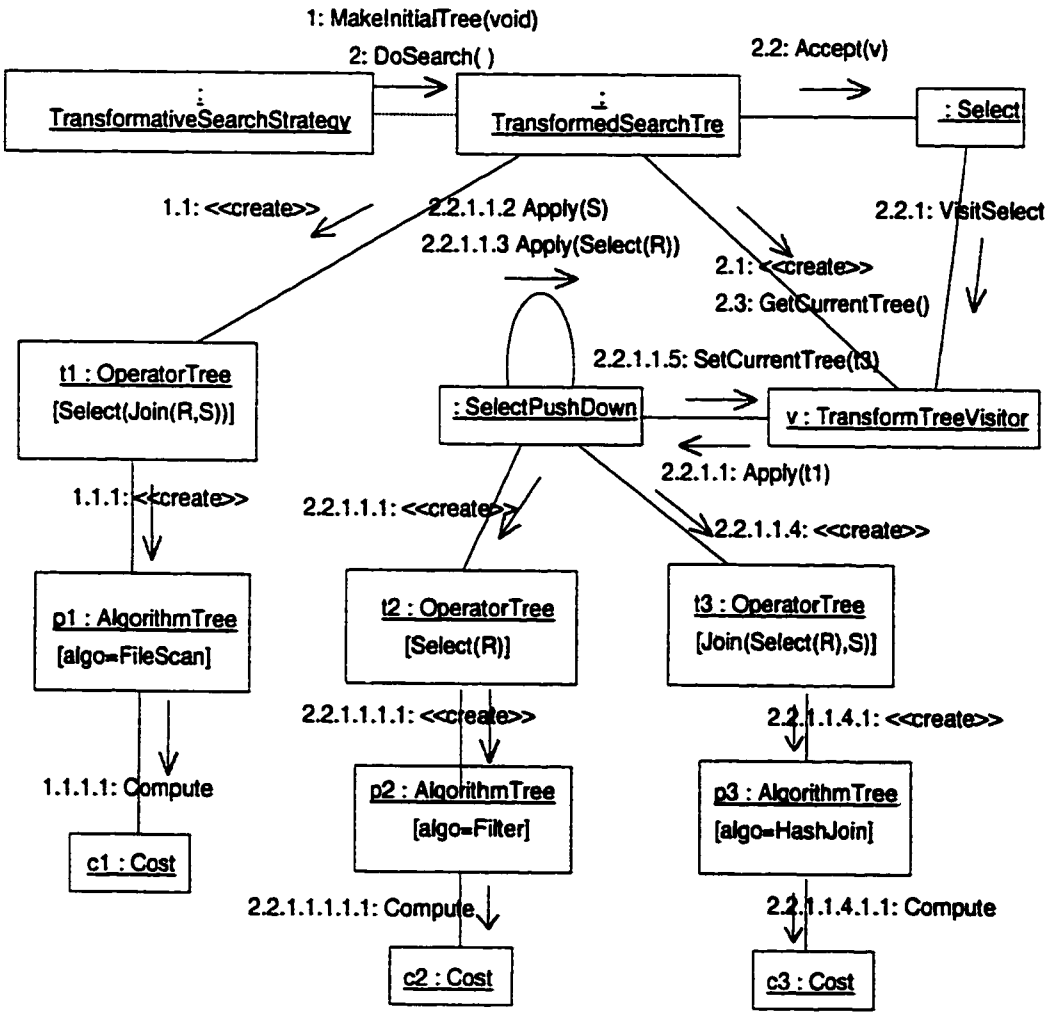

Figure 51: Collaboration Diagram for Transformation Optimization

compute its execution cost. In Figure 51, c1 is the estimated execution cost for the physical query plan p1. Since there are no equivalent physical plans for p1, it is kept for further optimization.

DoSearch is then called on the TRANSFORMEDSEARCHTREE to perform the search based on the initial logical plan t1. First, a TRANSFORMTREEVISITOR object v is created. It is then accepted by the root operator (Select) of the initial logical query plan t1. When the Select operator is visited, Select Push Down rule is applied. There are several steps in applying the Select Push Down rule to tree t1:

1. Select operator is applied to both inputs of the Join operator (child of the Select operator) in Figure 50(a), which are relations R and S.

2. Since Select operator can not apply to relation S, S is kept and will serve as one of the new inputs.

3. Select operator applies to relation R. A new logical query plan t2 is created. Its physical query plan p2 is also created and execution cost c2 is computed. Since there are no equivalent plans for p2, it is kept for further optimization.

4. A new logical query plan t3 rooted at the Join operator is created. Relation S serves as its left input, and new logical query plan t2 serves as its right input. Logical query plan c has a structure like Figure 50(b). Physical query plan p3 for t3 is created. Execution cost c3 for p3 is computed.

5. Since physical query plan p1 and p3 are equivalent while p3 is less expensive, p1 is pruned out. Plan p3 is kept for further optimization.

6. Since no algebra laws can be further applied to transform the logical query plan t3, t3 is the optimal logical query plan (a logical query plan whose physical query plan has least cost) and is stored in the TRANSFORMTREEVISITOR object.

7. The TRANSFORMEDSEARCHTREE object calls the method *GetCurrentTree* on the TRANSFORMTREEVISITOR object v to withdraw the optimal logical query plan t3, from which the physical query plan p3 is extracted and is returned as the result of tree transformation.

4.7 Cost Evaluation

The cost evaluation implements the core function of query optimization. It encapsulates the cost model and the cost pruning functionality. Each physical query plan, no matter if it is supposed to be implementing a partial or complete logical query plan, is compared to the existing physical query plans. Equivalent physical query plans that have higher costs are pruned out.

4.7.1 Cost Model

In the query optimization framework, the cost model is a hot spot open for customization. Information used in estimating the execution cost of a database algorithm (a physical operator) may include:

- Estimates of the sizes of the inputs and the output. This information is stored in the OPERATORTREEPROPERTY instance associated with that operator tree.
- Selectivities of the predicates applied by the algorithm. This information is computed by the *Selectivity* method of the PREDICATETREE class.
- Estimates of the number of CPU instructions required to execute the predicates and expressions evaluated by the algorithm.
- Miscellaneous information likes size of memory available, page-size, etc.

Here is an example showing the cost model for a File Scan algorithm:

$$\text{Cost}_{FileScan} = \text{number of pages} \times \text{I/O cost per page} + \text{number of pages} \times \text{number of instructions per page} \times \text{execution cost per instruction}$$

The estimated execution cost for the File Scan algorithm is the sum of the disk I/O cost to bring the corresponding relation into the memory and the execution cost to scan all the tuples in that relation. Amongst these parameters, the number of pages is computed at run time, and the others are estimated values associated with the machine that is used for query optimization.

There are some alternative cost models proposed in the query optimization literature. The most frequently used one is to use the disk I/O to predict the cost. Although this one is good enough in the general case, it is not suitable in cases where

queries are evaluated on a parallel machine or collection of interconnected machines. Therefore, it is good to predict the execution cost as precisely as possible so that it is not only suitable for the case that the disk I/O dominates but also suitable for the cases that does not.

4.7.2 Case Study

There are three cases when a physical query plan is evaluated:

Exact Match

1. A new physical query plan is created.
2. There exists a physical query plan in the search tree, which produces the same output as the new one. That is, all of their logical properties and physical properties are the same. This old physical query plan is an exact match of the new one.
3. Compare the estimated cost of the new physical query plan to its exact match. If the new one is more expensive that the old one, it is deleted. Otherwise, the new one replaces the old one and the old one is deleted.

Replacer and Replacee

1. A new physical query plan is created.
2. There not exists an exact match of the new physical query plan. But there exists a physical query plan whose logical properties are the same as the new one, but its physical properties are not, and its physical properties are not interesting.
3. Compare the estimated cost of the new physical query plan to the old one. If the old one is more expensive, it is replaced by the new one because the new plan provides everything that the old one provides at a less cost. The old one is deleted. We call the new plan a Replacer and the old one a Replacee.
4. The situation reverses if the old plan has interesting physical properties and it has less cost. The new plan is deleted.

Others

1. A new physical query plan is created.
2. There not exists any physical query plan in the search tree that has the same logical properties as the new one.
3. The new plan is kept around to be considered for further optimization.

4.7.3 Structure

Figure 52 shows the class diagram for hashing. They are auxiliary classes for cost pruning. The major classes include HASHID, HASHTABLE, and HASHNODE. The HASHNODE is the basic element for pruning. It contains references to both the logical properties and physical properties for fast access, as well as a reference to the physical query plan it refers to and references to a list of other hash nodes that can be used to replace itself. The HASHTABLE maintains a hash table for all the hash nodes that are used in pruning. It is responsible for rehashing, finding the exact match for the physical query plan it refers to, and finding a replacee that can be replaced by the physical query plan it refers to. HASHID is the control class in hashing. It contains a reference to the physical query plan it refers to, a reference to a replacee that can be replaced by itself, a reference to an exact match hash node that has the same logical and physical properties as itself, and a list references to some other hash nodes that can be used to replace itself. The HASHID class centralizes the control for locating the exact match node, a replacee, and a replacer, replacing a node and/or enabling a node to delete itself. It dispatches some of its work to other classes.

The logical properties and the physical properties that are used in hashing can be fast accessed through the HASHNODE class. The estimated execution cost is stored in the ALGORITHMTREEPROPERTY object that can also be easily accessed through the HASHNODE class. Also, for fast access purpose, both classes of HASHID and HASHNODE maintains a reference to the corresponding physical query plan they refer to.

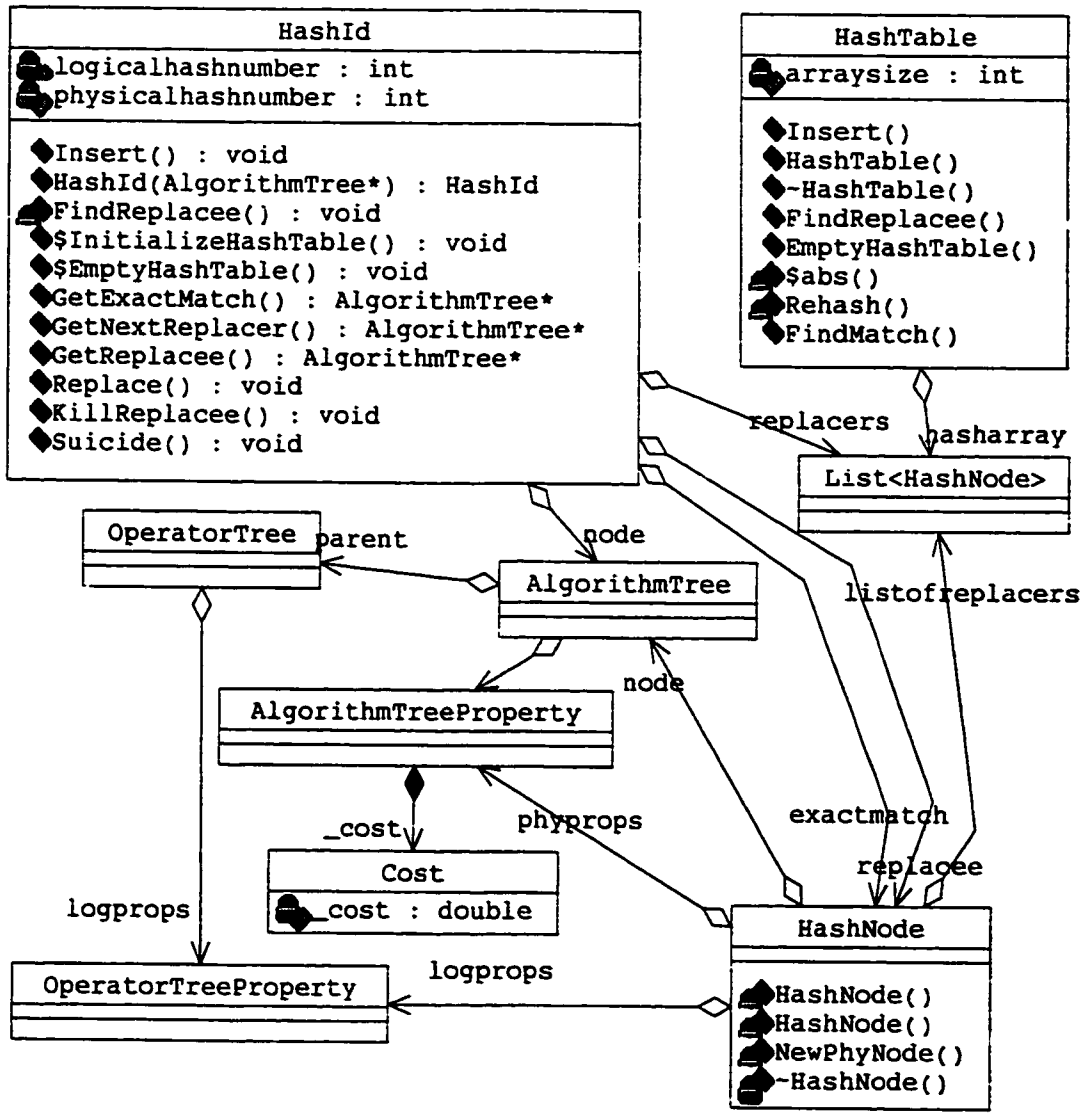

Figure 52: Auxiliary Classes for Cost Pruning

4.7.4 Class Descriptions

4.7.4.1 The HASHID Class

The HASHID class is responsible for searching amongst a list of hash nodes, locating the exact match node, a replacee node, the replacer nodes, and performing the replacing and/or suicide operations to prune out the sub-optimal physical query plans.

The major methods and attributes are:

Insert — a public method that is used to insert a physical query plan into the hash table.

Replace — a public method that is used to replace a sub-optimal physical query plan in the search tree with the current one.

Suicide — a public method that is called when a physical query plan is pruned out of the search tree. It deletes itself by committing suicide.

InitializedHashTable — a static method that is used to initialize the hash table before the cost pruning begins.

EmptyHashTable — a static method that is used to clean up the hash table when the cost pruning ends.

node — a private attribute representing the physical query plan it refers to.

exactmatch — a private attribute representing a physical query plan that exists in the search tree who has the same logical and physical properties with the physical query plan associated with itself. It has an associated method *GetExactMatch* that returns this object.

replacee — a private attribute representing a physical query plan in the search tree that can be replaced by the current one. It has two associated methods: *GetReplacee* returns this object and *KillReplacee* deletes this object.

replacers — a private attribute representing a list of physical query plans that can be used to replace itself. It has an associated method *GetNextReplacer* that returns the next replacer in the list.

Class HashId	Collaborators + AlgorithmTree + HashNode
Responsibility + Finds an exact match. + Finds a replacee. + Finds replacers. + Replaces sub-optimal plans and commits suicide.	

Table 4: CRC Card for the Class HASHID

logicalhashnumber — a private attribute whose value is computed according to the logical properties of the corresponding logical query plan.

physicalhashnumber — a private attribute whose value is computed according to the physical properties of the corresponding physical query plan.

Table 4 shows the CRC card [29] for the class HASHID. The HASHID class collaborates with the ALGORITHMTREE class and the HASHNODE class. The first class helps it to calculate the hash values. The latter serves as the basic element for hashing.

4.7.4.2 The HASHTABLE Class

In order to find the Exact Match, a Replacer, and a Replacee efficiently, a hash table is used as the major data structure to facilitate the cost pruning. A node can go to two possible buckets in the hash table. The first hash value is computed according to the physicalhashnumber provided by the HASHID class if the physical query plan is interesting. The bucket this kind of hash nodes go to will be used for looking for the Exact Match of interesting nodes. The other hash value is computed according to the logicalhashnumber that is also provided by the HASHID class. The bucket this kinds of hash nodes go to is used to store nodes that don't have interesting physical properties. It is a place for looking for Replacees.

When a new node is created and inserted in the hash table, there are two possibilities:

1. It is not interesting. In this case, the logical properties are used to calculate the hash value and the node is inserted in the corresponding bucket.

Class	Collaborators
HashTable	+ HashNode
Responsibility	
+ Maintains a list of hash nodes.	
+ Locates an exact match.	
+ Finds replacers.	
+ Locates a replacee.	

Table 5: CRC Card for the Class HASHTABLE

2. It is interesting. In this case, the physical properties are used to calculate the hash value and the node is inserted in the corresponding bucket.

In addition, the logical properties are also used to calculate the hash values and the nodes are inserted in the list of replacers that are maintained at that location.

Major methods and attributes of the HASHTABLE class include:

Insert — a public method that is used to insert a hash node into the hash table for hashing. It is used when no exact match can be found for this hash node.

Rehash — a public method that is used to re-hash the hash table.

FindMatch — a public method that is used to locate the exact match hash node. Two hash nodes are exact matches if they have the same logical properties and physical properties.

FindReplacee — a public method that is used to locate a hash node, which can be replaced by the current one.

hasharray — a private attribute that is a list, each element is a list of hash nodes.

Table 5 shows the CRC card [29] for the class HASHTABLE. The HASHTABLE class collaborates with the HASHNODE class which provides the basic element in the hash table.

4.7.4.3 The HASHNODE Class

The HASHNODE class represents the basic element for hashing.

Major methods and attributes in the HASHNODE class include:

Class HashNode	Collaborators + AlgorithmTree + HashId + OperatorTreeProperty + AlgorithmTreeProperty
Responsibility + A basic element for hashing. + Facilitates easy access to logical properties and physical properties.	

Table 6: CRC Card for the Class HASHNODE

NewPhyNode — a public method that is used to initialize a reference to a physical query plan this hash node associates with, to initialize a reference to its logical properties, and to initialize a reference to its physical properties.

node — a private attribute representing the physical query plan this hash node refers to.

logprops — a private attribute referring to the logical properties (an instance of the OPERATORTREEPROPERTY) associated with this hash node.

phyprops — a private attribute referring to the physical properties (an instance of the ALGORITHMTREEPROPERTY) associated with this hash node.

listofreplacers — a private attribute representing a list of hash nodes that can replace the current one.

Table 6 shows the CRC card [29] for the class HASHNODE. The HASHNODE class collaborates with four classes: ALGORITHMTREE, ALGORITHMTREEPROPERTY, OPERATORTREEPROPERTY, and the HASHID. It provides fast access to the associated physical query plan, its physical and logical properties.

4.7.5 Dynamic Cost Pruning

Figure 53 shows the algorithm of the cost pruning in an activity diagram. The cost pruning begins with a search of an exact match (a node with the same logical and physical properties). If it is not found, the current node is inserted into the hash

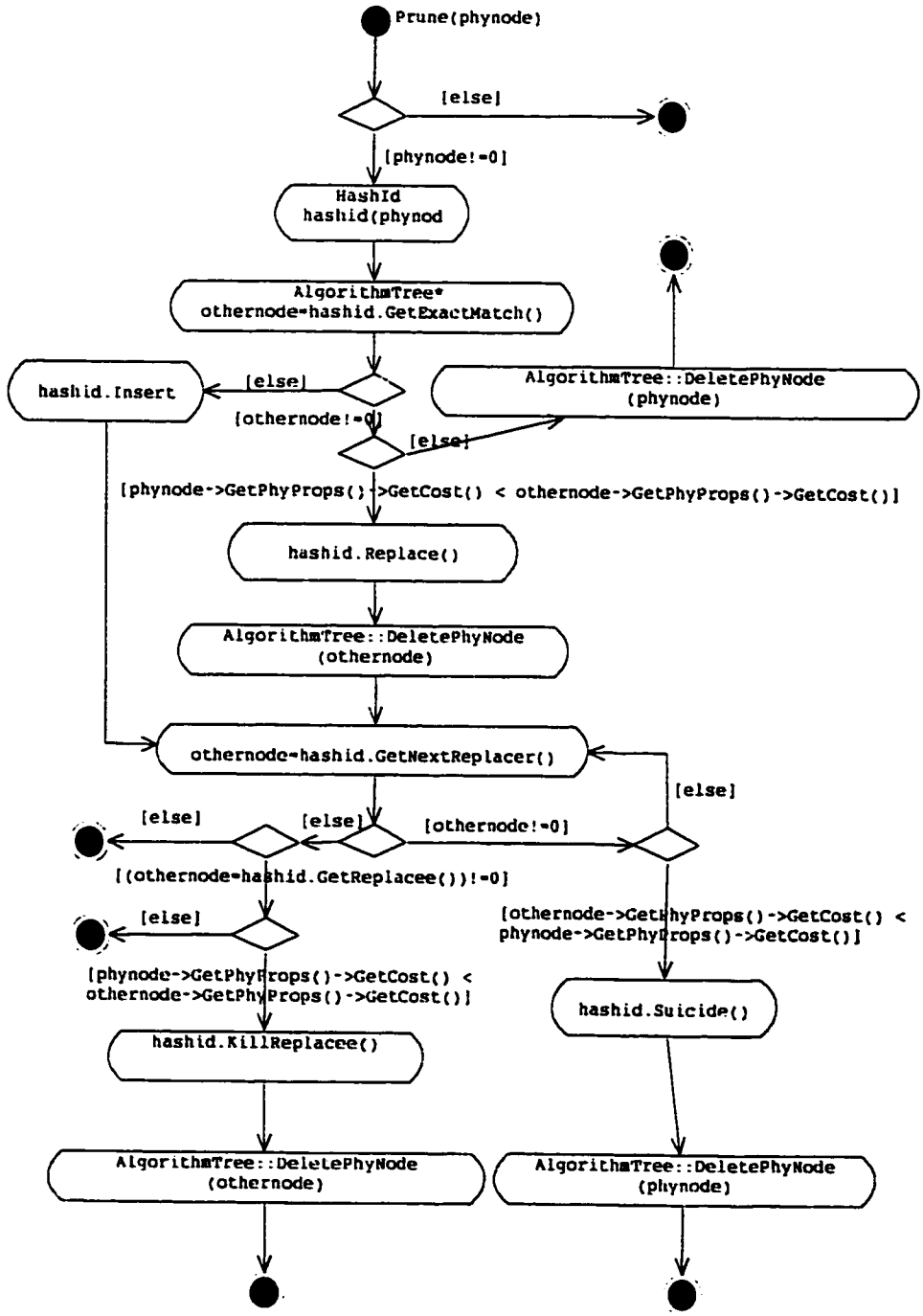

Figure 53: Activity Diagram for Cost Pruning

table for further evaluation. If an exact match is found and it has less cost than the current node, the current node is deleted and the process terminates. Otherwise, the current node replaces the exact match in the hash table. The exact match is deleted.

The cost pruning process continues to locate a replacer for the current node. If any one in the list of replacers has less cost than the current node, the current node is deleted and the process terminates.

If none of the replacers has less cost than the current node, The cost pruning process continues to locate the replacee for the current node. If it is found and it has higher cost than the current node, the replacee is deleted. Otherwise, the cost pruning process terminates.

4.7.6 Examples

This section shows two cost pruning examples in optimizing a query:

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

One is based on the Bottom-Up Search Strategy and the other is based on the Transformative Search Strategy.

We assume each database operator has only one execution algorithm. So any operator tree will have only one corresponding algorithm tree. Since we know algorithm trees have the same tree structures as their operator trees, we only show the operator trees in this example for simplicity purpose. The execution costs are gathered as measurement results from the query optimization system built from this framework. Please note that these costs are associated with the algorithm trees, not the operator trees. We put them together with the operator trees in order to show the estimated execution costs for the corresponding algorithm trees associated with these operator trees.

4.7.6.1 Bottom-Up Query Optimization

Steps:

1. Make Initial Trees

   We get two initial operator trees in Figure 54. They represent the relations involved in the query. They are kept around for further optimization.

Figure 54: Initial Trees

2. Expand Trees

   (a) Expand Tree t1

   Since at this moment tree t2 has not been processed, it is invisible. No operator can be applied to tree t1 and it is added to the list of atomic nodes for later processing.

   (b) Expand Tree t2

   Two operators can be applied to tree t2. One is the Select operator, the other is the Join operator. When the Select operator is applied, tree t3 is created. Its physical query plan is generated, whose estimated execution cost is computed, that is 115.553.

   When the Join operator is applied, tree t4 and t5 are generated. The difference between tree t4 and t5 is that their inputs are in reversed order. The physical query plans of tree t4 and t5 are generated, whose estimated costs are 181.979 and 142.919 respectively (see Figure 55).

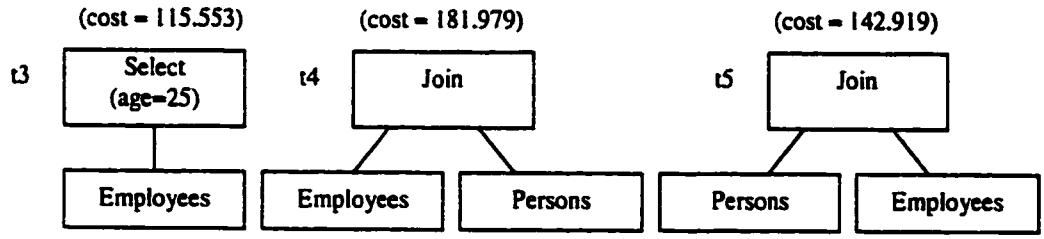

Figure 55: Resultant Trees After Expanding Tree t2

Since t4 and t5 are equivalent (they have the same logical and physical properties) and t4 is more expensive than tree t5, t4 is pruned out and deleted. Trees t5 and t3 are kept around for further optimization.

   (c) Expand Tree t3

   Only one operator can be applied to tree t3 for expansion, that is the Join operator.

When the Join operator is applied, two operator trees are created. They differ in the reversed order of their left and right inputs of the Join operator. Physical query plans for tree t6 and t7 are generated, whose execution costs are estimated, they are 129.745 and 129.349 respectively (see Figure 56).

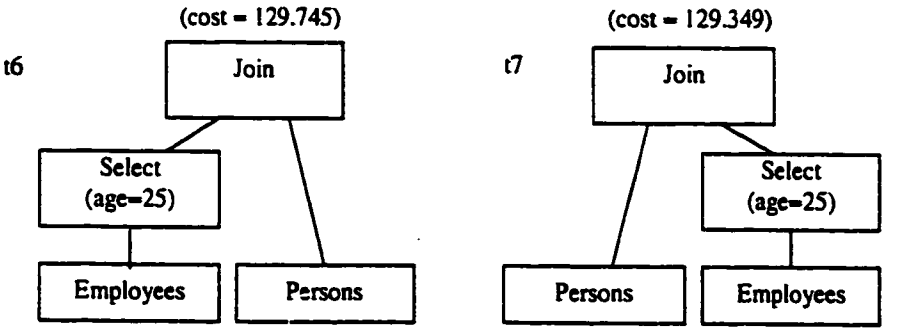

Figure 56: Resultant Trees After Expanding Tree t3

Since tree t6 and t7 are equivalent but t6 is more expensive than t7, t6 is pruned out and deleted. Tree t7 is kept around for further optimization.

(d) Expand Tree t5

Only operator Select can be applied to tree t5. The new operator tree is tree t8. Its physical query plan is generated, whose execution cost is estimated, that is 142.944 (see Figure 57).

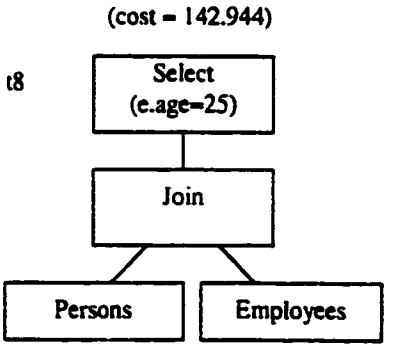

Figure 57: Resultant Tree After Expanding Tree t5

Since trees t8 and t7 both represents the same query, they are equivalent. But t8 is more expensive than t7. It is pruned out and deleted.

(e) Expand Tree t7

Tree t7 represents the complete query and no other operators can be applied to it for further expansion. Physical query plan of tree t7 is returned as the optimal plan that has least cost.

4.7.6.2 Transformation Query Optimization

Two steps are involved in Transformation Optimization:

1. Make an initial tree. An operator tree that represents the complete query is created.

2. Transform tree. All tree transformation rules are applied to transform one operator tree to another.

Make An Initial Tree

1. Create Initial Trees

   Two initial operator trees are created according to the relations involved in the query. They are tree t1 and t2 in Figure 58.

Figure 58: Initial Trees

2. Expand Tree t1

   Since at this moment tree t2 has not been processed, it is invisible. No operator can be applied to tree t1 and it is added to the list of atomic nodes for later processing.

3. Expand Tree t2

   If one operator can apply to an operator tree in the process of building the initial operator tree that represents the complete query, we stop trying other alternatives that may exist when applying some other operators. Suppose the Join operator is applied before the Select operator. Since tree t2 can be applied by the Join operator, two operators trees t3 and t4 are created. They differ in

the way their inputs are in reversed order. Physical query plans of these trees are generated and their execution costs are estimated. They are 181.979 and 142.919 respectively in Figure 59.

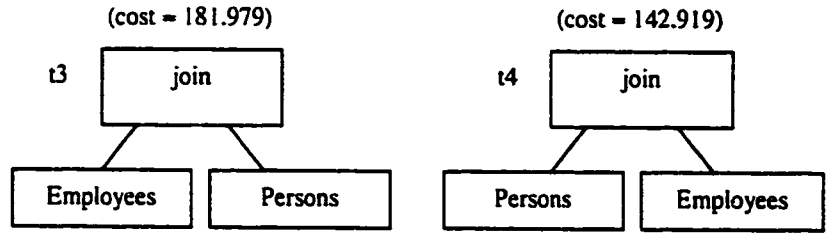

Figure 59: Resultant Trees After Expanding Tree t2

Since tree t3 and t4 represents the same part of the query, they are equivalent. But tree t3 is more expensive than tree t4. It is pruned out and deleted. Tree t4 is kept around for further expansion.

4. Expand Tree t4

   Only the Select operator can be applied to tree t4. The resultant operator tree is t5 (see Figure 60). Its physical query plan is generated, whose execution cost is estimated, that is 142.944.

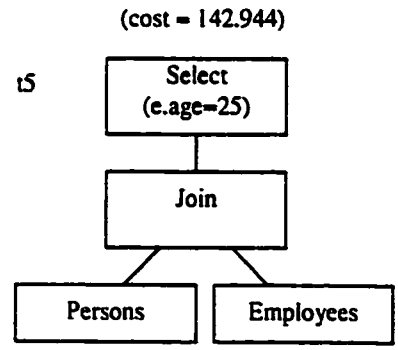

Figure 60: Resultant Trees After Expanding Tree t4

Since tree t5 represents the complete query, it is the initial operator tree that has been created in the first step of transformation optimization.

Transform Tree

For simplicity, we suppose that only rule Select Push Down is defined in the system.

Since the predicate (e.age=25) of the Select node in tree t5 only relates to the left input of the Join node, rule Select Push Down can be applied to push the Select node down the tree to the place as close as possible to its associated relation Employees.

1. Apply Select to both left and right children of Join

   The first step is to apply the Select operator to both inputs of the Join node.

   Since the predicate of the Select node does not relate to the relation Persons, the left input of the Join is kept unchanged. We use a new label t6 for it. Select operator can be applied to the right input of the Join node, we get a new right input t7. Physical query plan of t7 is generated, whose execution cost is estimated, that is 115.553. Both t6 and t7 are displayed in Figure 61.

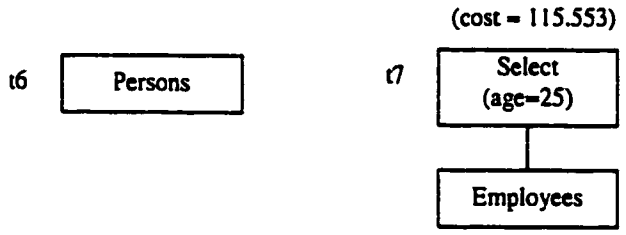

Figure 61: Resultant Trees After Applying Select

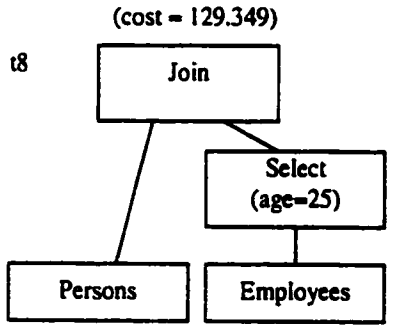

Figure 62: Create A New Join Tree

2. Apply SELECTPUSHDOWN to Trees t6 and t7

   Since SELECTPUSHDOWN can not be applied to both trees t6 and t7, they are kept unchanged and will serve as the left and right inputs in a new operator tree (rooted at Join operator) that will be built later.

3. Create a New Join Tree

   A new operator tree t8 (Figure 62) rooted at the Join operator is created. Tree

t6 and t7 serve as the left and right inputs of this new tree. The physical query plan of tree t8 is generated and its execution cost is estimated. It is 129.349.

Both trees t5 and t8 represent the complete query while t5 is more expensive than t8. Tree t5 is pruned out and deleted. Tree t8 is kept around for further optimization.

Since no rules can be applied to tree t8, it is the resultant tree of tree transformation. Its physical query plan is returned as the optimal plan with least cost.

Chapter 5

Examples of Customization

This chapter will illustrate how to customize the query optimization framework based on some examples, which are arranged from simple through to advanced. We will first illustrate how to customize each component of the query optimization framework. Then we show the customization of the system catalog. At the end we will illustrate how to build a new query optimization system using the framework and how to modify and extend an existing query optimization system built from the framework.

5.1 Customize the Algebra Component

The Algebra component incorporates the representation of the operations on the database and the various execution algorithms for these operations. It defines the logical and physical operators in the database system. The logical operator refers to an operation on the database, while the physical operator refers to an execution algorithm for a logical operator.

Since a physical operator is an execution algorithm for a logical operator, physical operators should be defined after their corresponding logical operators have been defined.

5.1.1 Define Logical Algebra

Figure 63 shows the structure of the logical algebra before customization. The root class is an abstract class named DBOperator. It has two immediate subclasses DBUnaryOperator and DBBinaryOperator for the unary logical operators

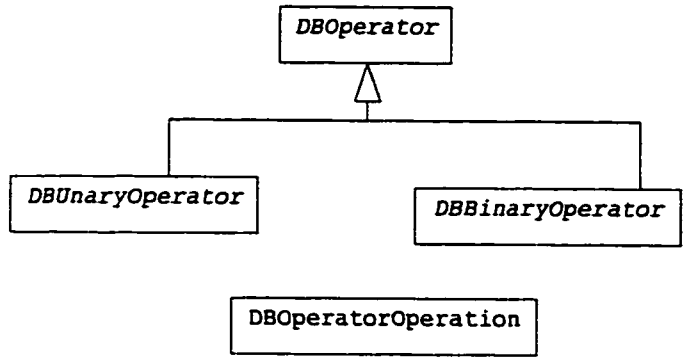

Figure 63: Class Diagram for Logical Algebra Before Customization

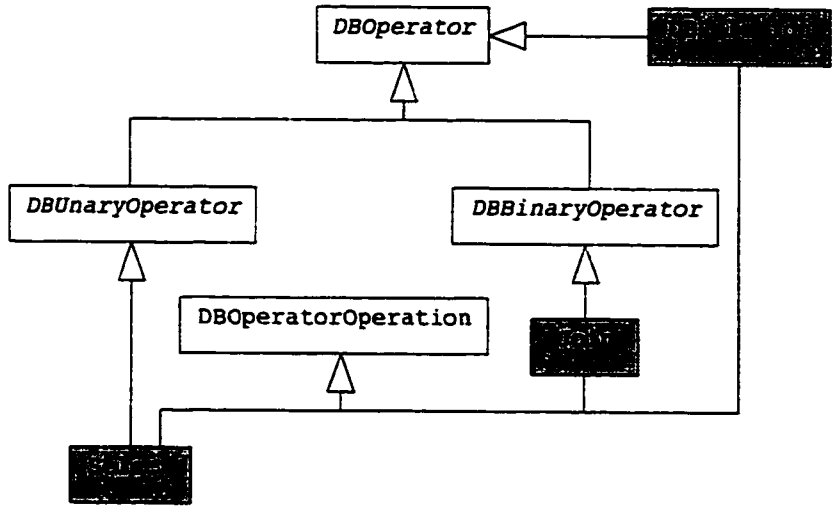

Figure 64: Example A: Customized Logical Algebra

and binary logical operators respectively. Both these subclasses are also abstract. A logical operator can be unary, binary, or none. There is also a class named DB-OPERATOROPERATION. It represents what operation is being applied by a logical operator.

Example A Suppose we want to optimize the basic SQL construct Select-From-Where. We will define the logical operators DBRelation, Select and Join. DBRelation represents an operator that has no inputs and serves as a leaf in the operator tree and refers to a relation stored in the database. The meanings of the other two operators are straightforward.

Figure 64 shows the class diagram after customization. The newly added classes are adorned in gray. For each logical operator, we define a new class in the logical

algebra class hierarchy. Operator DBRELATION has no inputs so it inherits from the root class DBOPERATOR. Operator SELECT is a unary operator (because it has only one input) so it inherits from the DBUNARYOPERATOR. Operator JOIN is a binary operator (because it has two inputs) so it inherits from the DBBINARYOPERATOR. All of these new logical operators must also inherit from the class DBOPERATOROPERATION. Any logical operator should have some associated operations that represent what operations are applied by this operator.

Since classes DBOPERATOR, DBUNARYOPERATOR, and DBBINARYOPERATOR in the framework are all abstract classes, the newly added subclasses must implement methods *Accept*, *Duplicate*, and *MakeLogProps*. Given an appropriate visitor object, the *Accept* method enables a potential operation to perform on a logical operator. *Duplicate* method asks the logical operator to copy itself. *MakeLogProps* sets up the logical properties for the operator tree rooted at this logical operator. A newly added logical operator may need to re-write the method *Clones*, which is a method representing different ways of applying this operator to its inputs, with different parameters.

Example B Suppose we want to add three complex logical operators to example A. The first one to add refers to reference-valued attributes that need to be de-referenced. We name it MATERIALIZATION. The second one refers to set-valued attributes that need unnesting. We name it UNNEST. The third one is the aggregation and grouping. Inputs can be grouped into partitions where any two records that have same values in the grouping attributes belong to the same partition. Operations are performed on these partitions. Operations MIN, MAX, AVG in the SQL query are examples of this kind of operations. We name this logical operator AGGREGATION.

Figure 65 shows the class diagram after customization. The newly added classes are adorned in gray. For each logical operator, we define a new class in the logical algebra class hierarchy. Operators MATERIALIZATION, UNNEST, and AGGREGATION are all unary operators (because each of them only has one input) so they all inherit from the DBUNARYOPERATOR. All of these new logical operators must also inherit from the class DBOPERATOROPERATION. Any logical operator should have some associated operations that represent what operations are applied by this operator.

Since classes DBOPERATOR, DBUNARYOPERATOR, and DBBINARYOPERATOR in the framework are all abstract classes, the newly added subclasses must implement

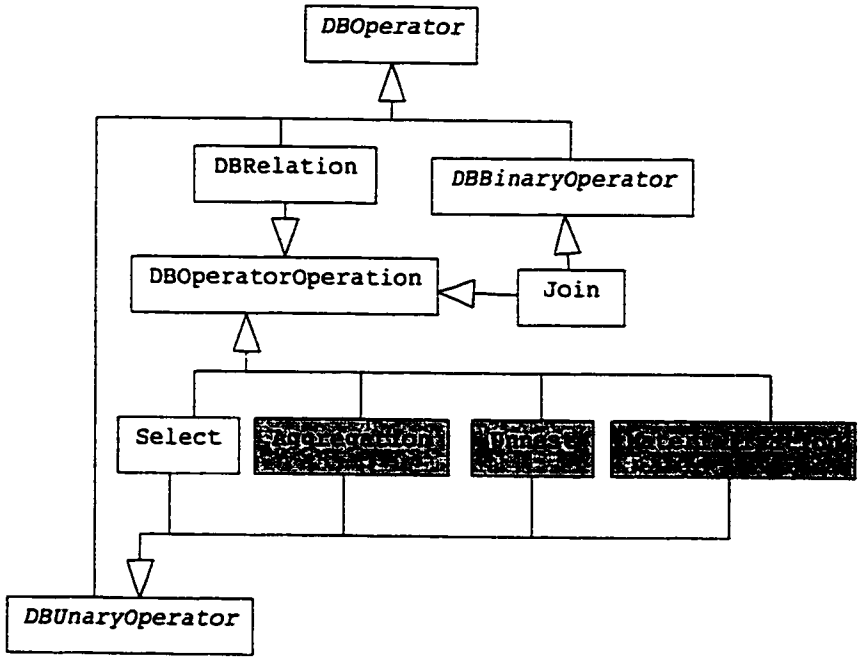

Figure 65: Example B: Customized Logical Algebra

methods *Accept*, *Duplicate*, and *MakeLogProps*. Given an appropriate visitor object, the *Accept* method enables a potential operation to perform on itself. *Duplicate* method asks the logical operator to copy itself. *MakeLogProps* sets up the logical properties for the operator tree rooted at this logical operator. A newly added logical operator may need to re-write the method *Clones*, which is a method representing different ways of applying this operator to its inputs, with different parameters.

5.1.2 Define Physical Algebra

Figure 66 shows the structure of the physical algebra before customization. The root class is an abstract class named DBALGORITHM. It has two immediate subclasses DBUNARYALGORITHM and DBBINARYALGORITHM for the unary physical operators and binary physical operators respectively. Both these two subclasses are also abstract. Any physical operator can be unary, binary, or none. Class ENFORCER is a subclass of the DBUNARYALGORITHM. It refers to any special execution algorithm that does not correspond to any operator in the logical algebra. The purpose of this kind of algorithm is not to perform any logical data manipulation but to enforce physical properties in the outputs that are required for subsequent query processing.

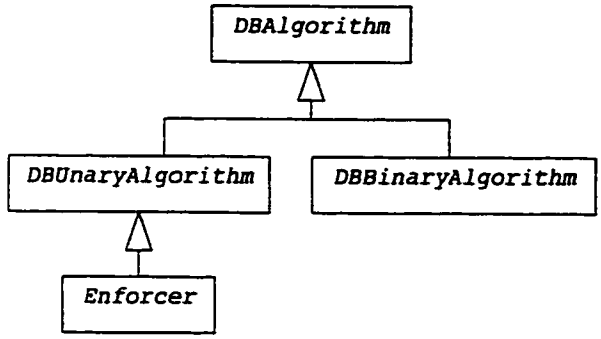

Figure 66: Class Diagram for Physical Algebra Before Customization

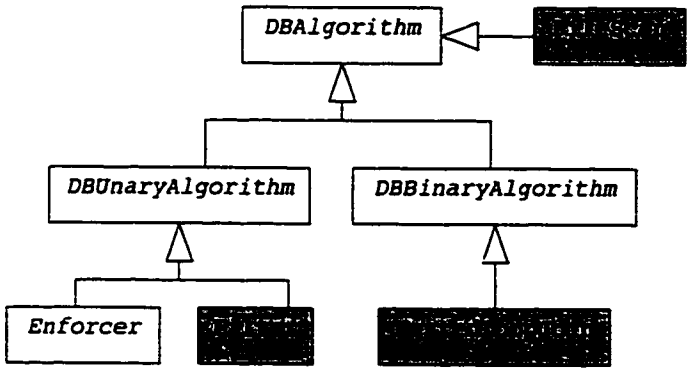

Figure 67: Example A: Customized Physical Algebra

For example, sort algorithm is an enforcer that can be used to ensure the inputs of the merge join are sorted on the join attributes.

Example A Suppose we want to define the execution algorithms for the logical operators in Example A of Section 5.1.1. For each logical operator, we define one corresponding execution algorithm. They are FileScan, Filter, and NestedLoopJoin corresponding to operators DBRelation, Select, and Join respectively. FileScan is an execution algorithm that is used to scan a relation sequentially. Filter is an execution algorithm that ensures the output to be part of the inputs that have met some specific conditions. NestedLoopJoin is a join execution algorithm where each tuple of its left input needs to find matches in all tuples of its right input.

Figure 67 shows the class diagram after customization. The newly added classes are adorned in gray. For each physical operator, we define a new class in the physical algebra class hierarchy. Operator FILESCAN has no inputs so it inherits from the root class DBALGORITHM. Operator FILTER is a unary operator (because it has

only one input) so it inherits from the DBUNARYALGORITHM. Operator NESTEDLOOPJOIN is a binary operator (because it has two inputs) so it inherits from the DBBINARYALGORITHM.

Since classes DBALGORITHM, DBUNARYALGORITHM, and DBBINARYALGORITHM in the framework are all abstract classes, the newly added subclasses must implement methods *Duplicate* and *MakePhyProps*. *Duplicate* method requires the physical operator to copy itself. *MakePhyProps* sets up the physical properties for the algorithm tree rooted at this physical operator. Class FILESCAN is a special case. It directly inherits from the root class DBALGORITHM, it has to implement two more methods: *Arity* and *MakePhyNodes*. *Arity* is a method that returns the number of inputs for this physical operator. *MakePhyNodes* is used to build the algorithm trees by delegating to the Search Space component. A newly added physical operator may need to re-write the method *Clones*, which implements different ways of applying this operator to its inputs, with different parameters.

Example B Suppose we want to define the execution algorithms for the logical operators in Example B of Section 5.1.1. For logical operator DBRELATION, we define two execution algorithms for it. They are FILESCAN and INDEXSCAN. FILESCAN is an execution algorithm that is used to scan a relation sequentially, while INDEXSCAN is an algorithm in which an index file is used to extract all the tuples that meet some conditions specified on the indexed attributes. For the logical operator Join, we define two execution algorithms for it. They are NESTEDLOOPJOIN and HASHJOIN. NESTEDLOOPJOIN is a join execution algorithm where each tuple of its left input needs to find matches in all tuples of its right input. HASHJOIN is a join execution algorithm in which tuples are matched by using hashing. MATERIALIZATIONALGORITHM, UNNESTALGORITHM, and AGGREGATIONALGORITHM are execution algorithms corresponding to logical operators MATERIALIZATION, UNNEST, and AGGREGATION defined in Example B of Section 5.1.1. MATERIALIZATIONALGORITHM is an algorithm that is used to de-reference the reference-valued attributes, i.e. a pointer. UNNESTALGORITHM is an algorithm that is used to unnest set-valued attributes. AGGREGATIONALGORITHM is an algorithm that is used for aggregation and grouping operations.

Figure 68 shows the class diagram after customization. The newly added classes are adorned in gray. For each physical operator, we define a new class in the physical

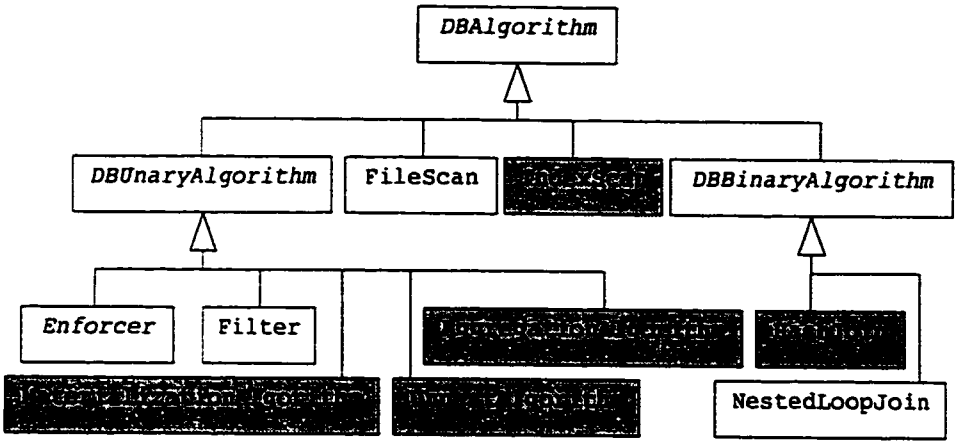

Figure 68: Example B: Customized Physical Algebra

algebra class hierarchy. Operators FILESCAN and INDEXSCAN have no inputs so they inherit from the root class DBALGORITHM. Operators FILTER, UNNESTALGORITHM, and AGGREGATIONALGORITHM are unary operators (because each of them only has one input) so they inherit from the DBUNARYALGORITHM. Operators NESTEDLOOPJOIN and HASHJOIN are binary operators (because each of them has two inputs) so they inherit from the DBBINARYALGORITHM.

Since classes DBALGORITHM, DBUNARYALGORITHM, and DBBINARYALGORITHM in the framework are all abstract classes, the newly added subclasses must implement methods *Duplicate* and *MakePhyProps*. *Duplicate* method requires the physical operator to copy itself. *MakePhyProps* sets up the physical properties for the algorithm tree rooted at this physical operator. Classes FILESCAN and INDEXSCAN directly inherit from the root class DBALGORITHM. They have to implement two more methods: *Arity* and *MakePhyNodes*. *Arity* is a method that returns the number of inputs for this physical operator. *MakePhyNodes* is used to build the algorithm trees by delegating to the Search Space component. A newly added physical operator may need to re-write the method *Clones*, which is a method represents different ways of applying this operator to its inputs, with different parameters.

5.1.3 Define OPERATORDEFINITION Class

Class OPERATORDEFINITION is defined for two purposes:

- Defines enumeration numbers for the logical operators and physical operators.

- Sets up relationships between the logical operators and their corresponding physical operators.

Example A Suppose we have defined the logical operators as that in Example A of Section 5.1.1 and physical operators as that in Example A of Section 5.1.2. Then in the header file where the OPERATORDEFINITION class declaration resides, we need to add the following C++ code:

```
enum DBOperatorNumber { DBRelation, Select, Join };

enum DBAlgorithmNumber {
    FileScan,
    Filter,
    NestedLoopJoin
};

class OperatorDefinition {
public:
  OperatorDefinition();
  ~ OperatorDefinition();

  // returns all execution algorithms associated with operator
  // DBRelation
  List<DBAlgorithm> GetDBRelationAlgorithms();
  // returns all execution algorithms associated with operator Select
  List<DBAlgorithm> GetSelectAlgorithms();
  // returns all execution algorithms associated with operator Join
  List<DBAlgorithm> GetJoinAlgorithms();

  // gets various logical operators defined in the database
  DBRelation* GetDBRelationOperator();
  Select* GetSelectOperator();
  Join* GetJoinOperator();
  List<DBOperator> GetAllLogicalOperators();
```

```
  // gets various physical operators defined in the database
  FileScan* GetFilescanAlgorithm();
  Filter* GetFilterAlgorithm();
  NestedLoopJoin* GetNestedLoopJoinAlgorithm();

private:
  // execution algorithms associated with the DBRelation operator
  List<DBAlgorithm> dbRelationAlgorithms;

  // execution algorithms associated with the Select operator
  List<DBAlgorithm> selectAlgorithms;
  // execution algorithms associated with the Join operator
  List<DBAlgorithm> joinAlgorithms;

  // logical operators
  DBRelation* dbrelation;
  Select* select;
  Join* join;
  // all logical operators
  List<DBOperator> allOperators;

  // physical operators;
  FileScan* filescan;
  Filter* filter;
  NestedLoopJoin* nestedloopjoin;
};
```

The constructor of this class will create objects for all logical operators and physical operators defined in the database, put all the logical operator objects into the attribute allOperators, and put each physical operator object into its corresponding list of algorithms associated with a logical operator. For example, object nestedloopjoin will be put into the list of joinAlgorithms, object filter will be put into the list of selectAlgorithms, etc. The *Get* methods defined in this class are used to return

corresponding attribute objects. We will not dive into details for these *Get* methods because their method names are explicit. The destructor of this class destroys all objects created in the constructor.

Example B Suppose we have defined the logical operators as that in Example B of Section 5.1.1 and physical operators as that in Example B of Section 5.1.2. Then in the header file where the OPERATORDEFINITION class declaration resides, we need to add the following C++ code:

```
enum DBOperatorNumber { DBRelation, Select, Join, Materialization,
Unnest, Aggregation };

enum DBAlgorithmNumber {
    FileScan,
    IndexScan,
    Filter,
    NestedLoopJoin,
    HashJoin,
    MaterializationAlgorithm,
    UnnestAlgorithm,
    AggregationAlgorithm
};

class OperatorDefinition {
public:
  OperatorDefinition();
  ~ OperatorDefinition();

  // returns all execution algorithms associated with operator
  // DBRelation
  List<DBAlgorithm> GetDBRelationAlgorithms();
  // returns all execution algorithms associated with operator Select
  List<DBAlgorithm> GetSelectAlgorithms();
  // returns all execution algorithms associated with operator Join
```

```
  List<DBAlgorithm> GetJoinAlgorithms();
  // returns all execution algorithms associated with Materialization
  List<DBAlgorithm> GetMaterializationAlgorithms();
  // returns all execution algorithms associated with operator Unnest
  List<DBAlgorithm> GetUnnestAlgorithms();
  // returns all execution algorithms associated with operator
  // Aggregation
  List<DBAlgorithm> GetAggregationAlgorithms();

  // gets various logical operators defined in the database
  DBRelation* GetDBRelationOperator();
  Select* GetSelectOperator();
  Join* GetJoinOperator();
  Materialization* GetMaterializationOperator();
  Unnest* GetUnnestOperator();
  Aggregation* GetAggregationOperator();

  // returns all the logical operators defined in the database
  List<DBOperator> GetAllLogicalOperators();

  // gets various physical operators defined in the database
  FileScan* GetFilescanAlgorithm();
  IndexScan* GetIndexScanAlgorithm();
  Filter* GetFilterAlgorithm();
  NestedLoopJoin* GetNestedLoopJoinAlgorithm();
  HashJoin* GetHashJoinAlgorithm();
  MaterializationAlgorithm* GetMaterializationAlgorithm();
  UnnestAlgorithm* GetUnnestAlgorithm();
  AggregationAlgorithm* GetAggregationAlgorithm();

private:
  // execution algorithms associated with the DBRelation operator
  List<DBAlgorithm> dbRelationAlgorithms;
```

```
// execution algorithms associated with the Select operator
List<DBAlgorithm> selectAlgorithms;

// execution algorithms associated with the Join operator
List<DBAlgorithm> joinAlgorithms;

// execution algorithms associated with the Materialization operator
List<DBAlgorithm> materializationAlgorithms;

// execution algorithms associated with the Unnest operator
List<DBAlgorithm> unnestAlgorithms;

// execution algorithms associated with the Aggregation operator
List<DBAlgorithm> aggregationAlgorithms;

// logical operators
DBRelation* dbrelation;
Select* select;
Join* join;
Materialization* materialization;
Unnest* unnest;
Aggregation* aggregation;

// all logical operators
List<DBOperator> allOperators;

// physical operators;
FileScan* filescan;
IndexScan* indexscan;
Filter* filter;
NestedLoopJoin* nestedloopjoin;
HashJoin* hashjoin;
```

```
  MaterializationAlgorithm* materializationAlgo;
  UnnestAlgorithm* unnestAlgo;
  AggregationAlgorithm* aggregationAlgo;
};
```

The constructor of this class will create objects for all logical operators and physical operators defined in the database, put all the logical operator objects into the attribute allOperators, and put each physical operator object into its corresponding list of algorithms associated with a logical operator. For example, object nestedloopjoin and hashjoin will be put into the list of joinAlgorithms, object materializationAlgo will be put into the list of materializationAlgorithms, etc. The *Get* methods defined in this class are used to return corresponding attribute objects. The destructor of this class destroys all objects created in the constructor.

5.2 Customize the Search Space Component

The Search Space component defines what the search space is. It decides how the logical operators and physical operators are put together to build the logical query plans and physical query plans. The abstraction of the kinds of manipulations on the database operators are encapsulated in this component.

Figure 40 shows the class diagram for the Search Space component before customization. There are two hierarchies in this component. One is the visitor hierarchy, the other is the generator hierarchy.

The visitor hierarchy performs operations on the logical algebra. The OPERATORTREEVISITOR is an abstract class and is used in places where polymorphism is needed. The EXPANDTREEVISITOR class is defined to expand an operator tree in a bottom-up fashion. The TRANSFORMTREEVISITOR class is designed to transform an operator tree to its equivalents. The TREETOPLANVISITOR class is designed to convert an operator tree to its corresponding algorithm trees. One operator tree may have more than one related algorithm trees if any operator in this tree has more than one execution algorithm. The visitor classes dispatch their responsibility to corresponding generator classes.

The generator hierarchy implements the responsibility of the Search Space component. There are three generator classes in the Search Space component, each is

created and used by a corresponding immediate OPERATORTREEVISITOR subclass. How the search space is shaped in query optimization is actually dependent on how these generator classes are implemented.

To customize the Search Space component, we need to customize both the visitor hierarchy and the generator hierarchy. The customization of the TRANSFORMTREEVISITOR and the TRANSFORMTREEGENERATOR is optional because they are only used in the Transformation optimization or any optimization such as Simulated Annealing that requires tree transformation.

5.2.1 A Simple Example

Suppose we want to optimize the basic SQL construct Select-From-Where. The logical operators that are defined are DBRelation, Select, and Join, as we have seen in Example A of Section 5.1.1. The physical operators are FileScan, Filter, and NestedLoopJoin corresponding to the above logical operators as that in Example A of Section 5.1.2.

There are several steps in customizing the Search Space component:

Step 1: Add the following methods to the class definition of OPERATORTREEVISITOR:

```
virtual void VisitDBRelation(DBRelation* op) = 0;
virtual voidVisitSelect(Select* op) = 0;
Virtual void VisitJoin(Join* op) =0;
```

Step 2: Add the following methods to all class definitions of EXPANDTREEVISITOR, TRANSFORMTREEVISITOR, and TREETOPLANVISITOR:

```
virtual void VisitDBRelation(DBRelation* op);
virtual void VisitSelect(Select* op);
virtual void VisitJoin(Join* op);
```

Step 3: Extend the EXPANDTREEGENERATOR class hierarchy.
For each logical operator defined in the Algebra component, there is a corresponding class defined for it in the EXPANDTREEGENERATOR hierarchy.

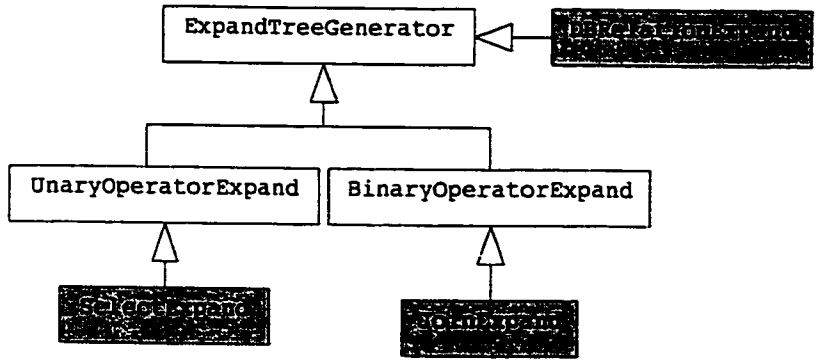

Figure 69: Simple Example: Customize the EXPANDTREEGENERATOR Hierarchy

Figure 69 shows the class diagram after customization. The newly added classes are adorned in gray. The EXPANDTREEGENERATOR class hierarchy has a similar structure as the DBOPERATOR hierarchy. Each class in the EXPANDTREEGENERATOR hierarchy has an associated logical operator in the DBOPERATOR hierarchy representing an operator tree is expanded by the class defined in the EXPANDTREEGENERATOR hierarchy through application of the corresponding logical operator defined in the DBOPERATOR hierarchy. The name convention for classes that are defined in the EXPANDTREEGENERATOR hierarchy and that correspond to some logical operators is : Each name is a concatenation of the name of the logical operator and the string "Expand".

Since DBRELATION operator has no input, its corresponding expand class DBRELATIONEXPAND inherits from the base class EXPANDTREEGENERATOR. The SELECT operator is a unary operator so its expand class SELECTEXPAND inherits from the UNARYOPERATOREXPAND. The JOIN operator is a binary operator and its expand class JOINEXPAND inherits from the BINARYOPERATOREXPAND. The newly added classes may need to re-write the method *Apply* in order to change the behavior when an operator tree is expanded by that expand class. Class JOINEXPAND may need to re-write the method *DfsNode* to change the way a depth first search is performed on the search tree for suitable nodes that can be paired with the given operator tree input.

Step 4: Extend the ALGORITHMTREEGENERATOR class hierarchy.
For each physical operator defined in the Algebra component, there is a corresponding class defined for it in the ALGORITHMTREEGENERATOR hierarchy.

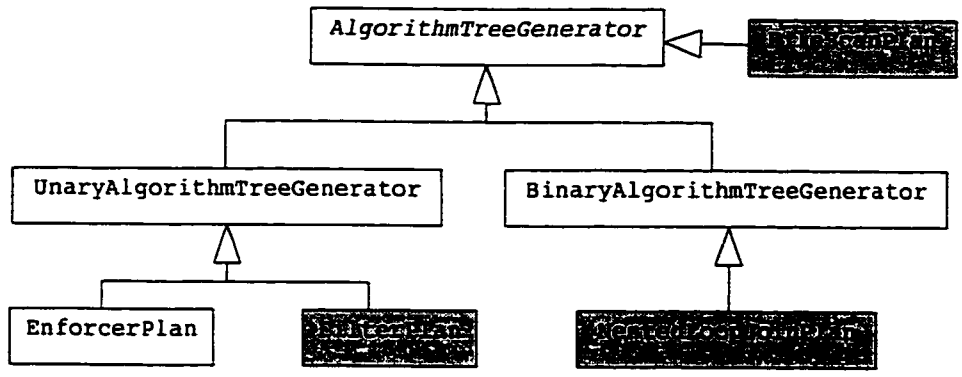

Figure 70: Simple Example: Customize the ALGORITHMTREEGENERATOR Hierarchy

Figure 70 shows the class diagram after customization. The newly added classes are adorned in gray. The ALGORITHMTREEGENERATOR class hierarchy has a similar structure as that of the DBALGORITHM hierarchy. Each class in the ALGORITHMTREEGENERATOR hierarchy has an associated physical operator in the DBALGORITHM hierarchy representing an algorithm tree is built by the class defined in the ALGORITHMTREEGENERATOR hierarchy through application of the corresponding physical operator defined in the DBALGORITHM hierarchy. The name convention for the classes defined in the ALGORITHMTREEGENERATOR hierarchy and that correspond to some physical operators is that each name is a concatenation of the name of the physical operator and the string "Plan".

Since the FILESCAN operator has no input, its corresponding plan class FileScanPlan inherits from the base class ALGORITHMTREEGENERATOR. The FILTER operator is a unary operator so its plan class FILTERPLAN inherits from the UnaryAlgorithmTreeGenerator. The NESTEDLOOPJOIN operator is a binary operator and its plan class NESTEDLOOPJOINPLAN inherits from the BINARYALGORITHMTREEGENERATOR. The newly added classes may need to re-write the method *CanBeApplied* to clarify the conditions it can be applied to build an algorithm tree. They may also re-write the method *MakePhyNodes* to change the behavior when an algorithm tree is built by that plan class.

Step 5: Define algebraic laws for operator tree transformation.
TRANSFORMTREEGENERATOR represents algebraic laws that will be used in tree transformation. There may not be one to one relationship between the logical operator

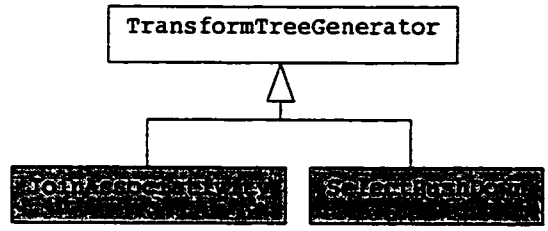

Figure 71: Simple Example: Customize the TRANSFORMTREEGENERATOR Hierarchy

and the algebraic law. But the algebraic laws should be defined according to all the logical operators defined in the database and try to cover all manipulations on these logical operators that can improve the quality of the operator trees. Insufficient algebraic laws may result in operator trees that are not good enough to be as close as possible to the best one in the search space.

Figure 71 shows the class diagram of the TRANSFORMTREEGENERATOR hierarchy after customization. The newly added classes are adorned in gray. For each algebraic law, there will be a concrete class defined in the TRANSFORMTREEGENERATOR hierarchy. In this example, there are two algebraic laws that are defined for the operator tree transformation. One is the SELECTPUSHDOWN. It is the most important algebraic law that is frequently used in the transformation optimization. It means putting the Select operator as close as possible to the leaves in an operator tree where its associated relations reside. The advantage of this rule is that it greatly reduces the cardinality of its output thus reducing the execution cost for further processing if its output will serve as some inputs in the later process. The other rule is JOINASSOCIATIVITY. It is the last step before producing the final logical query plan. JOINASSOCIATIVITY means grouping the Join operators in the operator tree together. Thinking of an operator such as Join as a multiway operator offers us opportunities to reorder the operands so that when the Join is executed as a sequence of binary Joins, they take less time than if we had executed the Joins in the order implied by the parse tree [12]. Since two argument relations play different roles in most Join execution algorithms, a potential advantage of JOINASSOCIATIVITY is that the smaller argument relation can be selected as the left argument and kept stored in the memory thus it reduces the total disk I/Os for the Join operation.

Step 6: Implement the newly added methods in classes EXPANDTREEVISITOR, TRANSFORMTREEVISITOR, and TREETOPLANVISITOR.

The implementation of the *Visit* methods in the class EXPANDTREEVISITOR is very simple. What they need to do is to pick up appropriate concrete generator classes and call the *Apply* method on them. For example, the *VisitJoin* method can be written as:

```
void ExpandTreeVisitor::VisitJoin ( Join* op ) {
    JoinExpand expand,
    expand.Apply(op, currentTree);
}
```

The object currentTree is the current operator tree to be expanded. It is stored in the EXPANDTREEVISITOR class.

The implementation of the *Visit* methods in the class TREETOPLANVISITOR is less simple, but it is not complicated either. What they need to do is to find the list of execution algorithms associated with that operator, iterate through these algorithms, and request them to build physical query plans by calling *MakePhyNodes* on them. Since the list of execution algorithms associated with one logical operator is encapsulated in that logical operator, all the *Visit* methods in the class TREETOPLANVISITOR can have the same implementation (but their function signatures are different). The following is an example for *VisitJoin* method:

```
void TreeToPlanVisitor ::VisitJoin ( Join* op ) {
    List<DBAlgorithm> list = op -> GetListOfAlgorithms();
    For_EACH_ELEMENT_OF_LIST (list) {
        DBAlgorithm* algorithm = list.Element();
        algorithm -> MakePhyNodes (currentTree);
    }
}
```

The object currentTree is the operator tree to be transformed into algorithm trees. It is stored in the class TREETOPLANVISITOR.

The implementation of the *Visit* methods in the class TRANSFORMTREEVISITOR is relatively complicated. First, the tree transformation is a recursive process. Second,

any tree transformation is based on the previous transformation resultant tree. Third, all rules are iterated and only some of them are used for tree transformation if they meet the application conditions. Fourth, the order of these rules are important. Two methods can be defined to facilitate the implementation of tree transformation:

- TransformOperatorTreeWithRules(OperatorTree* op)
  The argument op is the operator tree to be transformed. This method iterates through all algebraic laws defined in the TRANSFORMTREEGENERATOR hierarchy and asks each law (or rule) to apply. If a rule has been applied, then the tree to transform for the next rule is the resultant tree, not the original one.

- TransformOpeatorTreeWithRecursion (OperatorTree* op)
  The argument op is the operator tree to be transformed. This method transforms the root tree first by calling *TransformOperatorTreeWithRules* and providing the root tree as the argument tree. It then transforms the left and right operator trees recursively by calling itself but giving its left and right operator trees as arguments. The *Visit* methods in the TRANSFORMTREEVISITOR class only invoke the method *TransformOpeatorTreeWithRecursion* and pass the operator tree to transform as argument.

5.2.2 A Complex Example

We will show a more complex example in this section. We use the logical operators defined in the Example B of Section 5.1.1 and the physical operators defined in Example B of Section 5.1.2. In other words, we add logical operators MATERIALIZATION, UNNEST, and AGGREGATION and physical operators INDEXSCAN, HASHJOIN, MATERIALIZATIONALGORITHM, UNNESTALGORITHM, AGGREGATIONALGORITHM to the Simple Example in Section 5.2.1.

To customize this complex example, we use exactly the same number of steps as that in the simple example. We only show the additional methods or classes to add to the simple example for each step:

Step 1: Add the following additional methods to the class definition of OPERATORTREEVISITOR:

```
virtual void VisitMaterialization (Materialization* op) = 0;
```

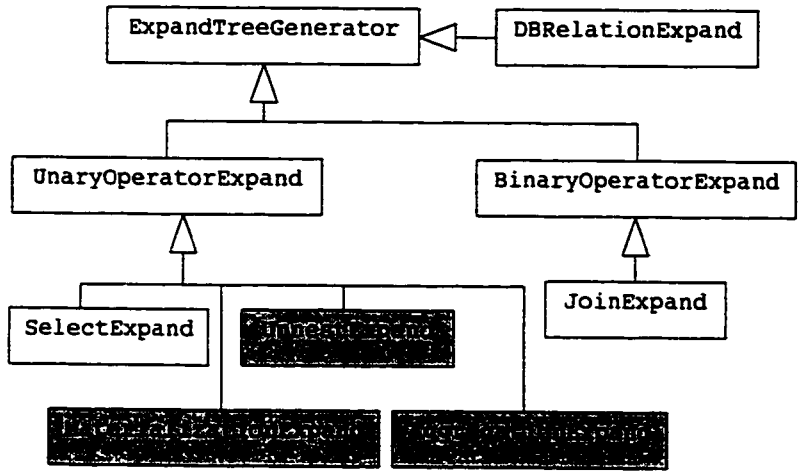

Figure 72: Complex Example: Customize the EXPANDTREEGENERATOR Hierarchy

```
virtual void VisitUnnest (Unnest* op) = 0;
virtual void Visit Aggregation (Aggregation* op) = 0;
```

Step 2: Add the following additional methods to each class definition of EXPANDTREEVISITOR, TRANSFORMTREEVISITOR, and TREETOPLANVISITOR:

```
virtual void VisitMaterialization (Materialization* op);
virtual void VisitUnnest (Unnest* op);
virtual void Visit Aggregation (Aggregation* op);
```

Step 3: Extend the EXPANDTREEGENERATOR class hierarchy.
For each logical operator defined in the Algebra component, there is a corresponding class defined for it in the EXPANDTREEGENERATOR hierarchy.

Figure 72 shows the class diagram after customization. The newly added classes are adorned in gray. Since logical operators MATERIALIZATION, UNNEST, AGGREGATION are all unary operators, their expand classes MATERIALIZATIONEXPAND, UNNESTEXPAND, and AGGREGATIONEXPAND all inherit from the UNARYOPERATOREXPAND. The newly added classes may need to re-write the method *Apply* to change the behavior when an operator tree is expanded.

Step 4: Extend the ALGORITHMTREEGENERATOR class hierarchy.
For each physical operator defined in the Algebra component, there is a corresponding class defined for it in the ALGORITHMTREEGENERATOR hierarchy.

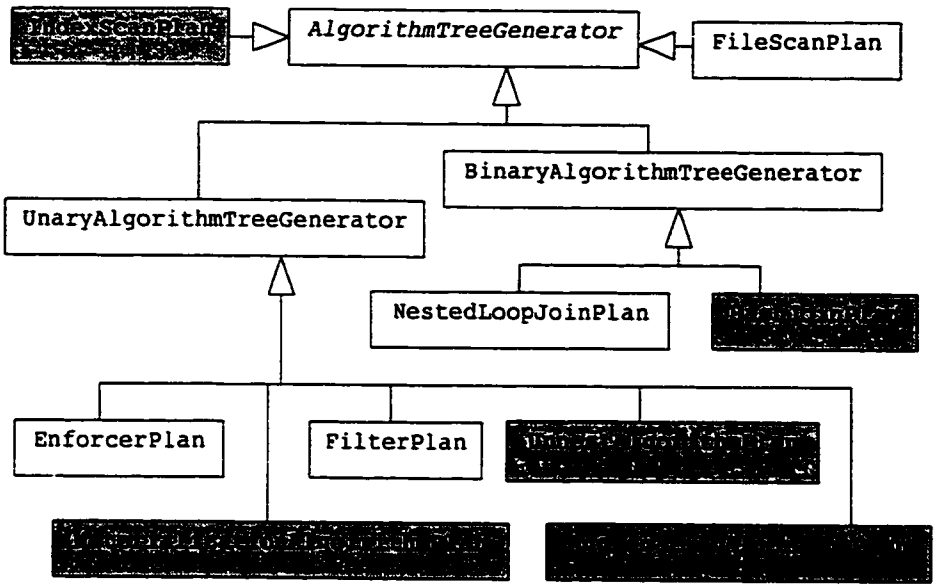

Figure 73: Complex Example: Customize the ALGORITHMTREEGENERATOR Hierarchy

Figure 73 shows the class diagram after customization. The newly added classes are adorned in gray. Since INDEXSCAN is a physical operator with no inputs, its plan class INDEXSCANPLAN directly inherits from the base class ALGORITHMTREEGENERATOR. MATERIALIZATIONALGORITHM, UNNESTALGORITHM, AGGREGATIONALGORITHM are all unary physical operators, their plan classes MATERIALIZATIONALGORITHMPLAN, UNNESTALGORITHMPLAN, AGGREGATIONALGORITHMPLAN all inherit from the class UNARYALGORITHMTREEGENERATOR. HASHJOIN is binary physical operator and its plan class HASHJOINPLAN inherits from class BINARYALGORITHMTREEGENERATOR. These newly added classes may need to re-write the method *CanBeApplied* to clarify the conditions it can be applied to build an algorithm tree. They may also re-write the method *MakePhyNodes* to change the behavior when an algorithm tree is built.

Step 5: Define algebraic laws for operator tree transformation.
Currently we are not sure what algebraic laws associated with operators MATERIALIZATION and UNNEST can be used to improve the logical query plan. But we do know there are transformation rules that are related to the operator AGGREGATION. Garcia-Molina et al. [12] propose two general algebraic laws related to the AGGREGATION operator:

- Operator Aggregation absorbs Operator Duplicate-Elimination. That means Duplicate-Elimination operation can be removed if its input is a resultant of the Aggregation operation.

- Useless attributes can be projected out prior to application of the Aggregation operation. That means a projection operation can be introduced to the input of the Aggregation operation to only keep those attributes that are mentioned in the aggregation and grouping expression.

We do not define a Duplicate-Elimination operator in our example. So the first law is ignored. But we can define an algebraic law (may be named PROJECTIONADD) as a subclass of the TRANSFORMTREEGENERATOR that is used to implement the second rule proposed by Garcia-Molina et al.

Please note that if any algebraic laws relating to the newly added operators that can be used to improve the quality of the operator tree, they should be added. Insufficient algebraic laws may result in operator trees that are not good enough to be as close as possible to the best one in the search space.

Step 6: Implement the newly added methods in classes EXPANDTREEVISITOR, TRANSFORMTREEVISITOR, and TREETOPLANVISITOR.
The implementation for methods related to the newly added operators is the same as that in the simple example.

5.2.3 Summary

The customization of the Algebra component will affect the customization of the Search Space component. Specifically, for each logical operator defined in the Algebra, we need to define one *Visit* method related to it across the visitor hierarchy. Moreover, we need to define a concrete class in the EXPANDTREEGENERATOR class hierarchy to implement the functionality of applying this logical operator to an operator tree. We may also need to define algebraic laws (tree transformation rules) for the related logical operators. Hopefully the application of these rules will improve the quality of the operator trees and lead to some physical query plans with less costs.

For each physical operator defined in the Algebra, we need to define a concrete

class in the ALGORITHMTREEGENERATOR class hierarchy to implement the functionality of applying this physical operator to build new physical query plans.

5.3 Customize the Search Strategy Component

5.3.1 Customize the Logical Properties

The logical properties for the logical query plans are stored in the class OPERATORTREEPROPERTY. Figure 33 shows its class structure before customization.

The logical properties of an operator tree are associated with the logical operator at the tree root. So for each logical operator in the database, a constructor is added to the class OPERATORTREEPROPERTY to construct the logical properties for the operator tree rooted at that logical operator.

Example A Suppose the logical operators defined in the database are the same as that in Example A of Section 5.1.1. The followings are constructors added to the class OPERATORTREEPROPERTY:

```
OperatorTreeProperty ( DBRelation*, OperatorTree* );
OperatorTreeProperty ( Select*, OperatorTree* );
OperatorTreeProperty ( Join*, OperatorTree* );
```

The second argument refers to the operator tree whose logical properties are to be made.

The optimizer-implementor may need to re-define the definitions for interesting logical properties and what equivalence of two operator trees means. S/he may also add some other logical properties to this class.

Example B Suppose we want to add more logical operators to Example A as what we have defined in the Example B of Section 5.1.2. That is, new logical operators MATERIALIZATION, UNNEST, and AGGREGATION are defined. The followings are additional constructors added to the class OPERATORTREEPROPERTY in addition to those in Example A:

```
OperatorTreeProperty (Materialization *, OperatorTree* );
OperatorTreeProperty (Unnest *, OperatorTree* );
```

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

The customization of this example is similar to that in Example A.

5.3.2 Customize the Physical Properties

The physical properties for the physical query plans are stored in the class ALGORITHMTREEPROPERTY. Figure 35 shows its class structure before customization.

The physical properties of an algorithm tree are associated with the physical operator at the tree root. So for each physical operator in the database, a constructor is added to the class ALGORITHMTREEPROPERTY to construct the physical properties for the algorithm tree rooted at that physical operator.

Example A Suppose the physical operators defined in the database are the same as that in Example A of Section 5.1.2. The followings are constructors added to the class ALGORITHMTREEPROPERTY:

```
AlgorithmTreeProperty (FileScan*, AlgorithmTree*);
AlgorithmTreeProperty (Filter*, AlgorithmTree*);
AlgorithmTreeProperty (NestedLoopJoin*, AlgorithmTree*);
```

The second argument refers to the algorithm tree whose physical properties are to be made.

The optimizer-implementor may need to re-define the definitions for interesting physical properties and what equivalence of two algorithm trees means. S/he may also add some other physical properties to this class.

Example B Suppose we want to add more physical operators to Example A as what we have defined in the Example B of Section 5.1.2. That is, new physical operators INDEXSCAN, HASHJOIN, MATERIALIZATIONALGORITHM, UNNESTALGORITHM, and AGGREGATIONALGORITHM are defined. The followings are additional constructors added to the class ALGORITHMTREEPROPERTY in addition to those in Example A:

```
AlgorithmTreeProperty (IndexScan*, AlgorithmTree*);
AlgorithmTreeProperty (HashJoin*, AlgorithmTree*);
```

```
AlgorithmTreeProperty (MaterializationAlgorithm *, AlgorithmTree*);
AlgorithmTreeProperty (UnnestAlgorithm *, AlgorithmTree*);
AlgorithmTreeProperty (AggregationAlgorithm *, AlgorithmTree*);
```

The customization of this example is similar to that in Example A.

5.3.3 Customize Cost Computation

The execution cost information is stored in the class COST. Figure 36 shows its class structure before customization.

For each execution algorithm, there should be a method defined in the COST class to compute the execution cost for it. The execution cost for an execution algorithm depends on the cost model used. Here is an example showing the cost value for the FileScan algorithm:

$$\text{Cost}_{FileScan} = \text{number of pages} \times \text{I/O cost per page} + \text{number of pages} \times \text{number of instructions per page} \times \text{execution cost per instruction}$$

The estimated execution cost for the FileScan algorithm is the sum of the disk I/O cost to bring the corresponding relation into the memory and the execution cost to scan all the tuples in that relation. Amongst these parameters, the number of pages is computed at run time, and the others are estimated values associated with the machine that is used for query optimization.

Example A Suppose the physical operators defined in the database are the same as that in Example A of Section 5.1.2. That is, they are FILESCAN, FILTER, and NESTEDLOOPJOIN. The followings are methods added to the class COST:

```
void Compute ( FileScan*, AlgorithmTree* );
void Compute ( Filter*, AlgorithmTree* );
void Compute ( NestedLoopJoin*, AlgorithmTree* );
```

The second argument refers to the algorithm tree whose execution cost is to compute.

Example B Suppose we define more physical operators than that in Example A as what we have defined in the Example B of Section 5.1.2. That is, new physical operators INDEXSCAN, HASHJOIN, MATERIALIZATIONALGORITHM, UNNESTALGORITHM, and AGGREGATIONALGORITHM are defined. The followings are additional methods to add to the class COST in addition to those in Example A:

```
void Compute (IndexScan*, AlgorithmTree*);
void Compute (HashJoin*, AlgorithmTree*);
void Compute (MaterializationAlgorithm*, AlgorithmTree*);
void Compute (UnnestAlgorithm*, AlgorithmTree*);
void Compute (AggregationAlgorithm*, AlgorithmTree*);
```

The second argument refers to the algorithm tree whose execution cost is to compute.

5.3.4 Selection of Input Format for Queries

In this query optimization framework, we provide two ways to use the query optimization system. One is a standalone query optimization system. This query optimization system incorporates a simple Query Parser that can parse user-input queries. It is used in cases where this system is repeatedly experimented with before delivery. The other is an embedded query optimization system. The queries are parsed outside this system and are turned into a file **.parsedquery** written with a protocol specified by the query optimization framework. This query optimization system will read this file in and turn it into internal representation for further optimization.

In a standalone query optimization system, the main control program will use the C++ code:

```
QueryOptimizerFacade* facade
                    = new QueryOptimizerFacadeWithParser;
```

Instead, in an embedded query optimization system, the main control program will use the C++ code:

```
QueryOptimizerFacade* facade
                    = new QueryOptimizerFacadeWithFormatedFile;
```

5.3.5 Selection of the Search Strategies

There are two search strategies equipped in this query optimization framework: the Bottom-Up Search Strategy and the Transformative Search Strategy. The optimizer-implementor can add some other search strategies to this framework or the query optimization system s/he builds.

Use Bottom-Up Search Strategy This search strategy is the default search method used in the system. No additional work needs to be done by the user. That is, after the user has chosen the input format for the queries and instantiated the corresponding facade object, a simple call as follows will automatically invoke the Bottom-Up Search Strategy.

```
facade -> Optimize();
```

Use Transformative Search Strategy The user needs to create a Transformative Search Strategy object and set it to be the one that will be used for search. That is, after the user has chosen the input format for the queries and instantiated the corresponding facade object, the following C++ code is written:

```
SearchStrategy* s = new TransformativeSearchStrategy;
facade -> SetSearchStrategy (s);
facade -> Optimize();
```

Then the Transformative Search Strategy will substitute the default search strategy and will be used for search.

Use Randomized Search Strategy No randomized search strategies are defined in this query optimization framework. But the user can add one without much effort. We will give an example of using the Simulated Annealing Search Strategy whose algorithm has been illustrated in Figure 12.

Since the Simulated Annealing Search Strategy uses the same search space as that explored by the Bottom-Up Search Strategy and the Transformative Search Strategy, no customization code in the Search Space component needs to be changed. Any code that relates to the new search strategy is added within the scope of the Search Strategy component.

First, we need to define the Simulated Annealing Search Strategy as an immediate subclass of the base class SEARCHSTRATEGY. Then a search tree is defined in the SEARCHTREE class hierarchy, we call it the SIMULATEDANNEALINGSEARCHTREE. There are two ways to define the SIMULATEDANNEALINGSEARCHTREE:

- It is defined as an immediate subclass of the class SEARCHTREE. The optimizer-implementor needs to write the method *MakeInitialTree* as that in the TRANSFORMEDSEARCHTREE in order to create a logical query plan representing the complete query. S/he also needs to override the method *DoSearch* to perform a random selection on options tree and plan. If the plan option is chosen, the current operator tree is transformed into algorithm trees and cost evaluation is performed. If the tree option is chosen, the current operator tree is transformed into an equivalent operator tree, the algorithm trees for the new tree are created, and the cost evaluation is performed.

- It is defined as a subclass of the TRANSFORMEDSEARCHTREE. The optimizer-implementor needs not define the method *MakeInitialTree*. S/he only re-writes the method *DoSearch* with the algorithm specified above.

The use of the Simulated Annealing Search Strategy is similar to that of the TRANSFORMATIVESEARCHSTRATEGY. That is,

```
SearchStrategy* s = new SimulatedAnnealingSearchStrategy;
facade->SetSearchStrategy (s);
facade -> Optimize();
```

Then the Simulated Annealing Search Strategy will substitute the default search strategy and will be used for search.

5.4 Customize the System Catalog

The system catalog information used in this query optimization framework is stored in the file .catalog in the bin subdirectory. This information can be collected from different layers of a DBMS.

This query optimization framework uses the same system catalog format as that in OPT++ [20]. The system catalog contains three parts of information: Type information, Set information, and Attributes/Methods information. We will illustrate them

22

Country	300	3	Y	Y	ext_Country
City	200	3	Y	N	ext_City
Capital	400	3	Y	Y	ext_Capital
Information	400	1	Y	Y	ext_Information
Destination	8	2	Y	N	—
Employee	250	4	Y	Y	ext_Employee
Plant	1000	3	Y	N	ext_Plant
Department	500	0	Y	Y	ext_Department
Job	400	0	Y	Y	ext_Job
Task	12	2	Y	Y	ext_Task
Hotel	100	0	Y	N	—
Person	100	1	Y	Y	ext_Person
CV	150	0	Y	N	—
Sale	16	2	Y	N	—
Continent	32	0	Y	Y	ext_Continent
Date	10	0	N	N	—
string	0	0	N	N	—
void	0	0	N	N	—
float	0	0	N	N	—
integer	4	0	N	N	—
boolean	1	0	N	N	—
Data_Type_Not_Set	1	0	N	N	—

Table 7: Example of System Catalog: Type Information

using the sample catalog file provided in the SampleOpt [19] of OPT++. SampleOpt is a complete query optimization system for object-relational query optimization. It is built from OPT++.

5.4.1 Type Information

The first line is the number of types in the system catalog. The following is the type information with columns from left to right representing the type name, object size for that type, number of references, is complex, has extent, and extent name.

Table 7 is the sample type information provided in SampleOpt. It contains 22 types, each has an object size, reference information, and extent information.

27

Departments	Department	100	Y	N	number	I1	—
Employees	Employee	50000	N	N	spouse.name	I2	—
Persons	Person	10000	Y	N	age	I3	—
Cities	City	10000	Y	N	mayor.name	I4	—
Continents	Continent	3	N	N	—	-	—
Countries	Country	100	Y	N	name	I5	—
Tasks	Task	10000	Y	N	time	I9	—
ext_Task	Task	10000	N	N	—	-	—
ext_Job	Job	5000	N	N	—	-	—
ext_Department	Department	1000	N	N	--	-	—
ext_Person	Person	100000	N	N	name	I6	—
ext_Employee	Employee	200000	Y	N	first_name	I10	—
ext_City	City	20000	N	N	—	-	—
ext_Plant	Plant	200	N	N	—	-	—
ext_Capital	Capital	160	N	N	—	-	—
ext_Continent	Continent	7	N	N	—	-	—
ext_Country	Country	160	Y	N	name	I8	—
ext_Information	Information	1000	N	N	—	-	—
I1	Data_Type_Not_Set	5	N	Y	—	Departments	p
I2	Data_Type_Not_Set	5000	N	Y	—	Employees	c
I3	Data_Type_Not_Set	200	N	Y	—	Persons	c
I4	Data_Type_Not_Set	5000	N	Y	—	Cities	sd
I5	Data_Type_Not_Set	100	N	Y	—	Countries	sd
I6	Data_Type_Not_Set	2000	N	Y	—	ext_Person	c
I8	Data_Type_Not_Set	160	N	Y	—	ext_Country	sd
I9	Data_Type_Not_Set	1000	N	Y	—	Tasks	c
I10	Data_Type_Not_Set	50	N	Y	—	ext_Employee	c

Table 8: Example of System Catalog: Set Information

5.4.2 Set Information

The first line is the number of sets in the system catalog. The following is the set information with columns from left to right representing the set name, the corresponding type name, cardinality, has index, is index, index path name, index or set name, and index type.

Table 8 shows the sample set information provided in SampleOpt. It contains 27 sets, each has a type name, cardinality, and index information.

75

name	Person	string	N 0	Attr Y N —	Data_Type_Not_Set
address	Person	string	N 0	Attr N N —	Data_Type_Not_Set
age	Person	integer	N 0	Attr N N —	Data_Type_Not_Set
spouse	Person	Person	N 0	Link N N —	Data_Type_Not_Set
print	Person	void	N 0	Attr N N —	Data_Type_Not_Set
self	Person	Person	N 0	Attr Y N —	Data_Type_Not_Set
time	Task	integer	N 0	Attr N N —	Data_Type_Not_Set
team_members	Task	Employee	Y 10	Link N N —	Data_Type_Not_Set
employee	Task	Employee	N 0	Link N N —	Data_Type_Not_Set
team_manager	Task	Employee	N 0	Link Y N —	Data_Type_Not_Set
name	Job	string	N 0	Attr Y N —	Data_Type_Not_Set
wage	Job	float	N 0	Attr N N —	Data_Type_Not_Set
self	Job	Job	N 0	Attr Y N —	Data_Type_Not_Set
location	Plant	string	N 0	Attr Y N —	Data_Type_Not_Set
self	Plant	Plant	N 0	Attr Y N —	Data_Type_Not_Set
department	Employee	Department	N 0	Link N N —	Data_Type_Not_Set
cv	Employee	CV	N 0	Link Y N —	Data_Type_Not_Set
sales	Employee	Sale	Y 25	Link N N —	Data_Type_Not_Set
number	Employee	integer	N 0	Attr Y N —	Data_Type_Not_Set
age	Employee	integer	N 0	Attr N N —	Data_Type_Not_Set
job	Employee	Job	N 0	Link N N —	Data_Type_Not_Set
job_description	Employee	string	N 0	Attr N N —	Data_Type_Not_Set
last_raise	Employee	Date	N 0	Attr N N —	Data_Type_Not_Set
print	Employee	void	N 0	Attr N N —	Data_Type_Not_Set
self	Employee	Employee	N 0	Attr Y N —	Data_Type_Not_Set
name	Employee	string	N 0	Attr Y N —	Data_Type_Not_Set
first_name	Employee	string	N 0	Attr N N —	Data_Type_Not_Set
address	Employee	string	N 0	Attr N N —	Data_Type_Not_Set
age	Employee	integer	N 0	Attr N N —	Data_Type_Not_Set
spouse	Employee	Person	N 0	Link N N —	Data_Type_Not_Set

Table 9: Example of System Catalog: Attributes and Methods (1/3)

5.4.3 Attributes and Methods

The first line is the number of attributes and methods in the system catalog. The following is the detail information with columns from left to right representing the attribute/method name, the owner name, the type, is set, set size, reference string(attribute or link), is key, has inverse relationship, name of inverse relationship, type of the inverse relationship.

Table 9, Table 10, and Table 11 show the sample attributes and methods information provided in SampleOpt. It contains 75 attributes and methods, each has an owner name (type name), a type name (string, float, etc), and relationship information.

print	Employee	void	N	0	Attr	N	N — Data_Type_Not_Set
name	Department	string	N	0	Attr	Y	N — Data_Type_Not_Set
floor	Department	integer	N	0	Attr	N	N — Data_Type_Not_Set
number	Department	integer	N	0	Attr	Y	N — Data_Type_Not_Set
plant	Department	Plant	N	0	Link	N	N — Data_Type_Not_Set
print	Department	void	N	0	Attr	N	N — Data_Type_Not_Set
self	Department	Department	N	0	Attr	Y	N — Data_Type_Not_Set
president	Country	Person	N	0	Link	Y	N — Data_Type_Not_Set
age	Country	integer	N	0	Attr	N	N — Data_Type_Not_Set
continent	Country	string	N	0	Attr	N	N — Data_Type_Not_Set
name	Country	string	N	0	Attr	Y	N — Data_Type_Not_Set
capital	Country	Capital	N	0	Link	Y	Y country Capital
self	Country	Country	N	0	Attr	Y	N — Data_Type_Not_Set
country	City	Country	N	0	Link	N	N — Data_Type_Not_Set
name	City	string	N	0	Attr	Y	N — Data_Type_Not_Set
age	City	integer	N	0	Attr	N	N — Data_Type_Not_Set
mayor	City	Person	N	0	Link	Y	N — Data_Type_Not_Set
self	City	City	N	0	Attr	Y	N — Data_Type_Not_Set
info	City	Information	N	0	Link	Y	N — Data_Type_Not_Set
country	Capital	Country	N	0	Link	Y	Y capital Country
name	Capital	string	N	0	Attr	Y	N — Data_Type_Not_Set
age	Capital	integer	N	0	Attr	N	N — Data_Type_Not_Set
mayor	Capital	Person	N	0	Link	Y	N — Data_Type_Not_Set
self	Capital	Capital	N	0	Attr	Y	N — Data_Type_Not_Set
info	Capital	Information	N	0	Link	Y	N — Data_Type_Not_Set
name	Continent	string	N	0	Attr	Y	N — Data_Type_Not_Set
self	Continent	Continent	N	0	Attr	Y	N — Data_Type_Not_Set
hotels	Information	Hotel	Y	75	Link	N	N — Data_Type_Not_Set
day	Information	string	Y	35	Attr	N	N — Data_Type_Not_Set
night	Information	string	Y	25	Attr	N	N — Data_Type_Not_Set

Table 10: Example of System Catalog: Attributes and Methods (2/3)

5.5 Putting It All Together — Build A New Query Optimization System

This section will illustrate how to build a query optimization system using this query optimization framework. Two examples will be given. All illustrations in this section are based on the previous examples in customizing individual components. We will not dive into details except when necessary.

self	Information	Information	N 0	Attr Y N —		Data_Type_Not_Set
self	Destination	Destination	N 0	Attr Y N —		Data_Type_Not_Set
city	Destination	City	N 0	Link N N —		Data_Type_Not_Set
hotel	Destination	Hotel	N 0	Link N N —		Data_Type_Not_Set
self	CV	CV	N 0	Attr N N —		Data_Type_Not_Set
name	CV	string	N 0	Attr N N —		Data_Type_Not_Set
birth_city	CV	string	N 0	Attr N N —		Data_Type_Not_Set
birth_country	CV	string	N 0	Attr N N —		Data_Type_Not_Set
self	Sale	Sale	N 0	Attr Y N —		Data_Type_Not_Set
dest	Sale	Destination	N 0	Link N N —		Data_Type_Not_Set
employee	Sale	Employee	N 0	Link N N —		Data_Type_Not_Set
amount	Sale	float	N 0	Attr N N —		Data_Type_Not_Set
name	Hotel	string	N 0	Attr N N —		Data_Type_Not_Set
self	Hotel	Hotel	N 0	Attr N N —		Data_Type_Not_Set
location	Hotel	string	N 0	Attr N N —		Data_Type_Not_Set

Table 11: Example of System Catalog: Attributes and Methods (3/3)

5.5.1 A Simple Example

Suppose we want to develop a transformation query optimization system that can accept basic SQL construct Select-From-Where. This system will be embedded in a DBMS system. Also suppose the database uses the same system catalog as the SampleOpt customization of OPT++ (see Section 5.4).

Several steps are sketched out as follows:

Step 1: Customize the Algebra Component

1. Define Logical Algebra

   Three logical operators DBRELATION, SELECT, and JOIN are defined. Please refer to Example A of Section 5.1.1 for detailed customization information.

2. Define Physical Algebra

   Three physical operators FILESCAN, FILTER, and NESTEDLOOPJOIN are defined according to the logical operators. Please refer to Example A of Section 5.1.2 for detailed customization information.

3. Define OPERATORDEFINITION Class

   Enumeration numbers are assigned to the logical and physical operators defined

above. Relationships between DBRELATION and FILESCAN, SELECT and FILTER, JOIN and NESTEDLOOPJOIN are defined.

Please refer to Example A of Section 5.1.3 for detailed customization information.

Figure 74 shows the overall class diagram of the Algebra component after customization.

Step 2: Customize the Search Space Component

The customization of the Search Space component requires both the visitor and generator hierarchies to be customized. Please refer to Section 5.2 for detailed customization information.

Figure 75 shows the overall class diagram of the Search Space component after customization.

Step 3: Customize the Search Strategy Component

1. Customize the Logical Properties

   Three constructors are added to the class OPERATORTREEPROPERTY for logical operators DBRELATION, SELECT, and JOIN.

   Please refer to Example A in Section 5.3.1 for detailed customization information.

2. Customize the Physical Properties

   Three constructors are added to the class ALGORITHMTREEPROPERTY for physical operators FILESCAN, FILTER, and NESTEDLOOPJOIN.

   Please refer to Example A in Section 5.3.2 for detailed customization information.

3. Customize Cost Computation

   Three *Compute* methods are added to the class COST for physical operators FILESCAN, FILTER, and NESTEDLOOPJOIN. Please refer to Example A in Section 5.3.3 for detailed customization information.

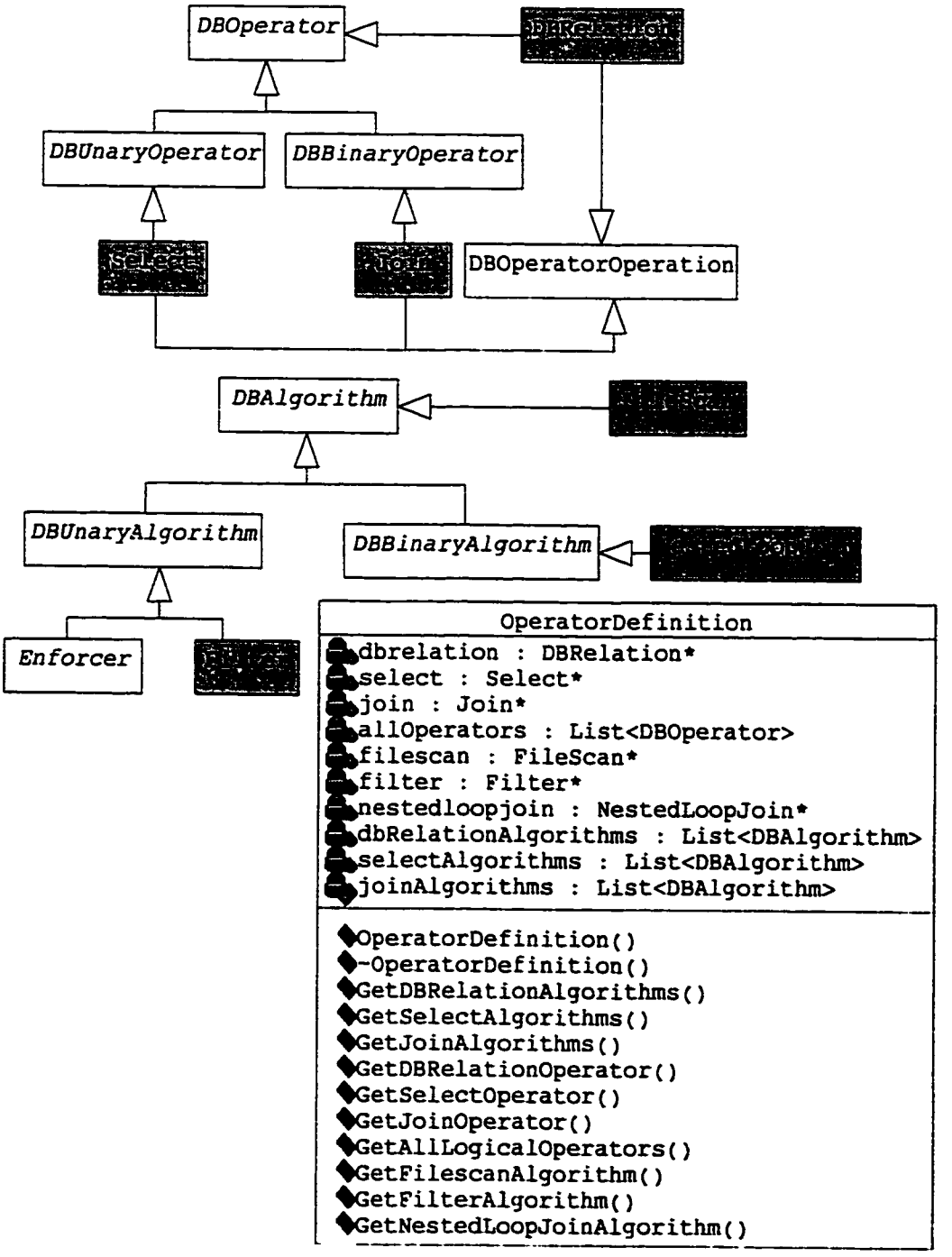

Figure 74: Simple Example: Customize the Algebra Component

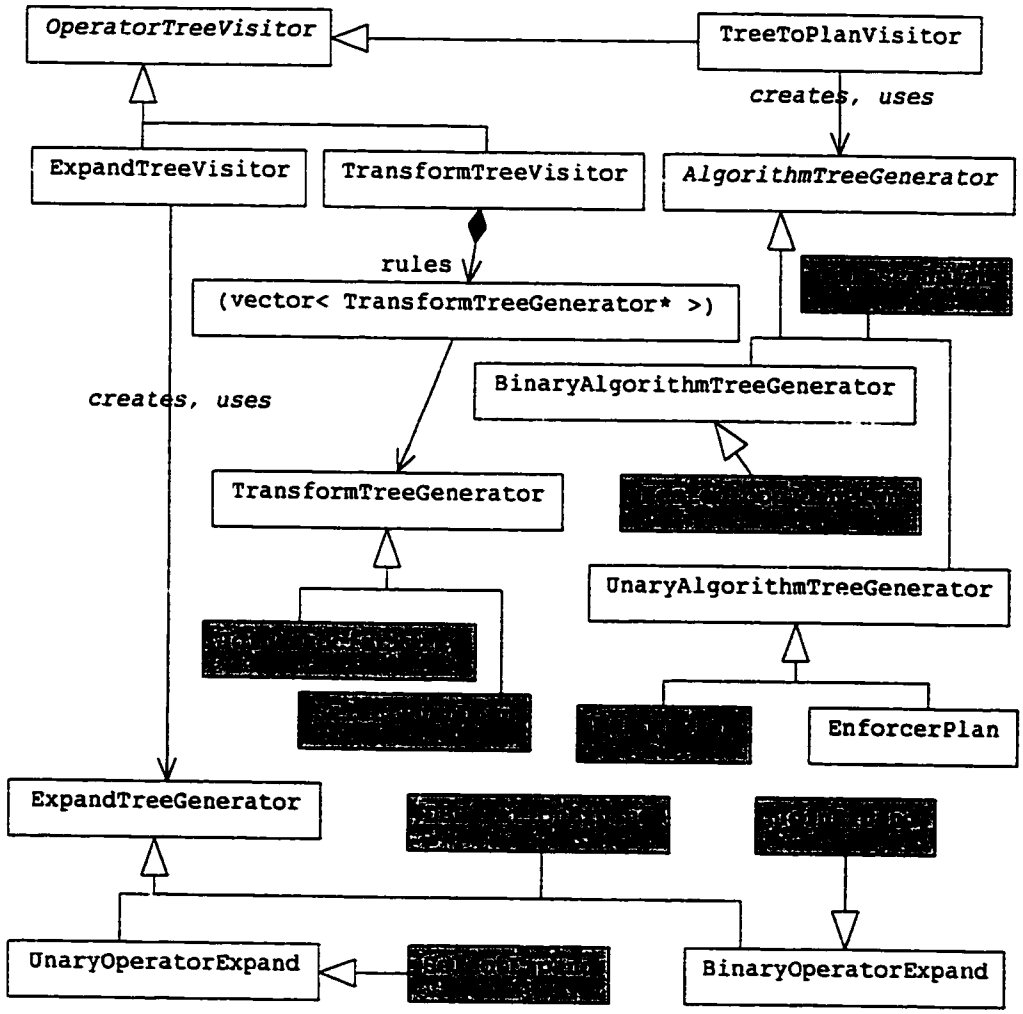

Figure 75: Simple Example: Customize the Search Space Component

4. Selection of Input Format for Queries

   In the main control program, use the C++ code:

```
QueryOptimizerFacade* facade
                    = new QueryOptimizerFacadeWithFormatedFile;
```

5. Selection of the Search Strategies

   The user needs to create a Transformative Search Strategy object and sets it to the one that will be used for search. That is, after the user has chosen the input format for the queries and instantiated the corresponding facade object, the following C++ code is written:

```
SearchStrategy* s = new TransformativeSearchStrategy;
facade -> SetSearchStrategy (s);
facade -> Optimize();
```

   Then the Transformative Search Strategy will substitute the default search strategy and will be used for search.

Step 4: Create file .parsedquery

File **.parsedquery** is a file that contains the information for a query being parsed. It is read by the query optimization system and is transformed into an internal representation for further optimization.

A **.parsedquery** file contains seven items of information:

Relation Information This information includes the number of relations involved in the query followed by a list of paired values of relation names and the relation variable names.

Attribute Information This information includes the number of attributes involved in the query followed by a list of paired values of relation names the attributes belong to and the names of the attributes.

Predicate Information This information includes the number of predicates involved in the query followed by a list of information of each predicate. Information for a predicate includes:

- Operation and number of operands
  For instance, in the predicate "Cities.age=10", operation is "=" and number of operands is 2.

- Attribute information or literal information
  Attribute information is identified by the keyword "attref" followed by the attribute name as well as its input information including: the number of input, input type, relation number, relation name, and relation variable name. For instance, in the predicate "c.age=10", suppose c is the relation variable name for the relation Cities which is the first relation stored in the system catalog, then we can represent the attribute age as follows:

  ```
  attref age 1   tuplevar, 1, Cities, c
  ```

  The literal information is identified by the keyword "lit". It also includes the type of the literal such as integer, string, boolean, etc (defined in the system catalog) and its value (in a double quote). For instance, we can represent the second operand in the predicate "c.age=10" as follows:

  ```
  lit integer "10"
  ```

- Target Attribute Information
  This information includes a list of Attribute information written in the same format as the attributes in the predicates.

- Group-by Information
  This information includes a boolean value (0 or 1). If it is 0, there is no aggregation operation in the query. Otherwise, a list of paired values of relation names and attribute names should be included, representing the grouping attributes on whose values the data is partitioned.

- Order-by Information

  This information includes a boolean value (0 or 1). If it is 0, there is no ordering operation in the query. Otherwise, a list of paired values of relation names and attribute names should be included, representing the attributes on whose values the data is sorted. Moreover, there is another boolean value representing these values are ordered in ascending order or not.

Example Suppose the query to be optimized is:

select name **from** Cities **where** age = 10;

Then its corresponding file **.parsedquery** should be:

```
1 Cities Cities

2
Cities name
Cities age

1
=
  2
  attref age
    1
    tuplevar 0 Cities Cities
       0
  lit integer  "10"

1
attref name
  1
  tuplevar    0     Cities    Cities
     0

0 0 0
```

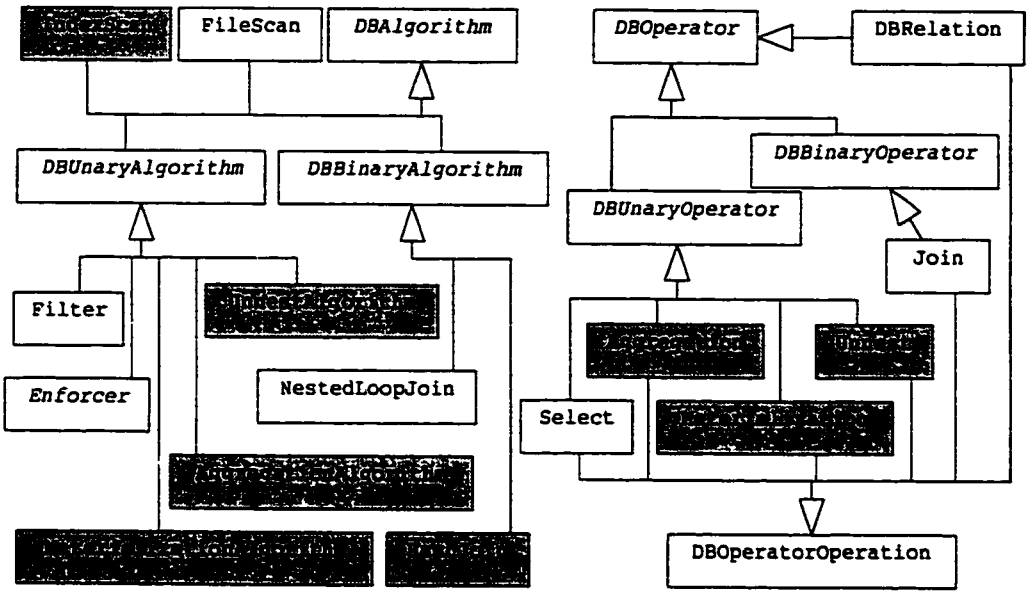

Figure 76: Complex Example: Customize the Algebra Component (1/2)

There is only one relation involved in this query. It is Cities. Two of its attributes name and age are referenced. There is one predicate in this query that has two operands: attribute age and literal 10. There is only one target attribute (attribute to be projected), that is name. No aggregation and order-by information involved in this query.

5.5.2 A More Complex Example

Suppose we want to develop a Bottomup query optimization system that can optimize the basic SQL construct Select-From-Where and some more complex operations on reference-valued attributes, set-valued attributes, and grouping and aggregation. This system can be standalone. Also suppose the database uses the same system catalog as the SampleOpt customization of OPT++ (see Section 5.4).

Several steps are sketched out as follows:

Step 1: Customize the Algebra Component

1. Define Logical Algebra. Six logical operators DBRELATION, SELECT, JOIN, MATERIALIZATION, UNNEST, and AGGREGATION are defined.

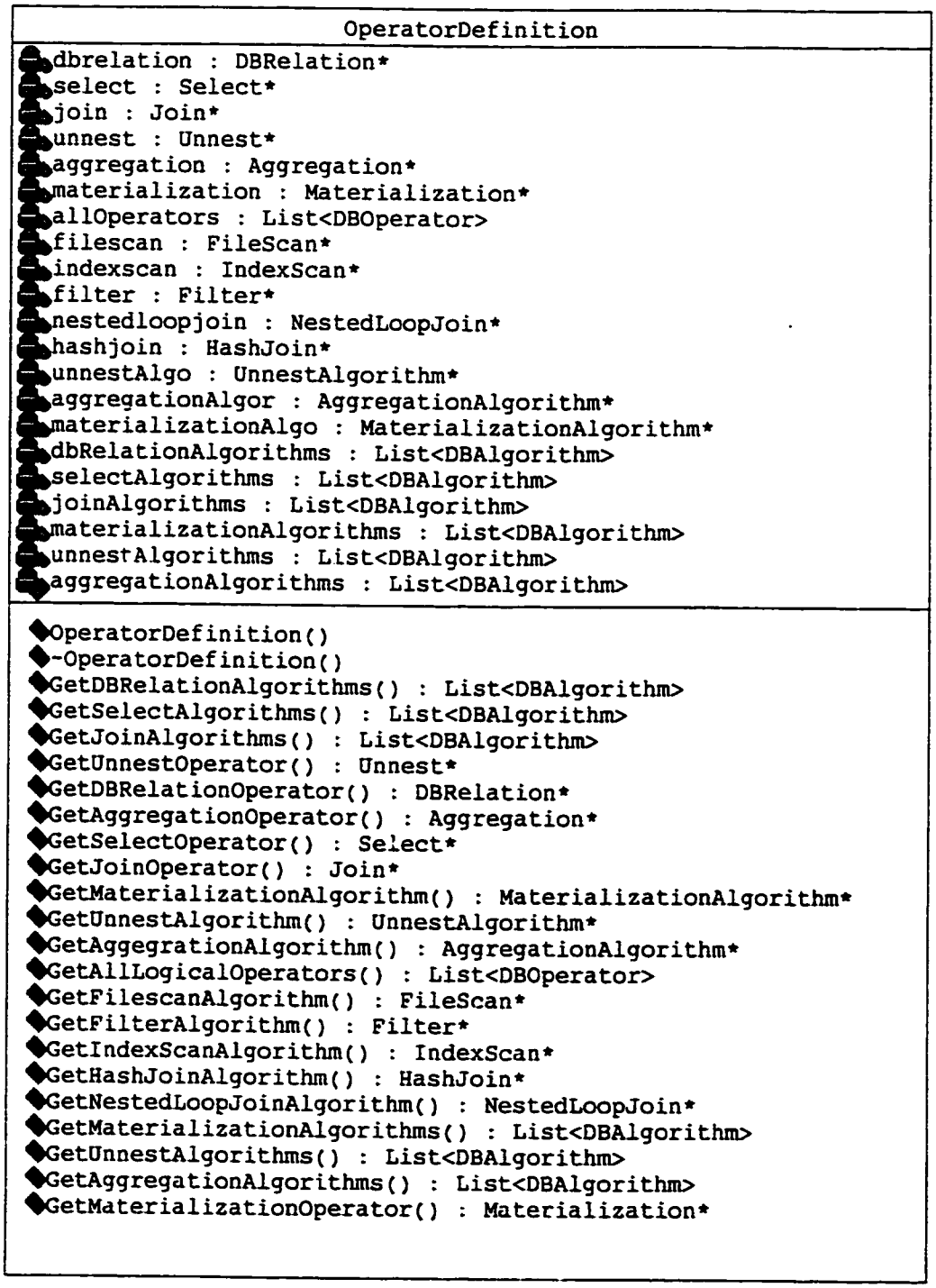

Figure 77: Complex Example: Customize the Algebra Component (2/2)

Please refer to Example B of Section 5.1.1 for detailed customization information.

2. Define Physical Algebra. Eight physical operators FILESCAN, INDEXSCAN, FILTER, NESTEDLOOPJOIN, HASHJOIN, MATERIALIZATIONALGORITHM, UNNESTALGORITHM, and AGGREGATIONALGORITHM are defined according to the logical operators defined above.

   Please refer to Example B of Section 5.1.2 for detailed customization information.

3. Define OPERATORDEFINITION Class. Enumeration numbers are assigned to the logical and physical operators defined above. Relationships are set up amongst them:

   - Logical operator DBRELATION and physical operators FILESCAN and INDEXSCAN.
   - Logical operator SELECT and physical operator FILTER.
   - Logical operator JOIN and physical operators HASHJOIN and NESTEDLOOPJOIN.
   - Logical operator MATERIALIZATION and physical operator MATERIALIZATIONALGORITHM.
   - Logical operator UNNEST and physical operator UNNESTALGORITHM.
   - Logical operator AGGREGATION and physical operator AGGREGATIONALGORITHM.

Please refer to Example B of Section 5.1.3 for detailed customization information.

Figure 76 and Figure 77 show the class diagrams in the Algebra component after customization.

Step 2: Customize the Search Space Component The customization of the Search Space component requires both the visitor and generator hierarchies to be customized. Please refer to Section 5.2 for detailed customization information.

Figure 78 shows the overall class diagram of the Search Space component after customization.

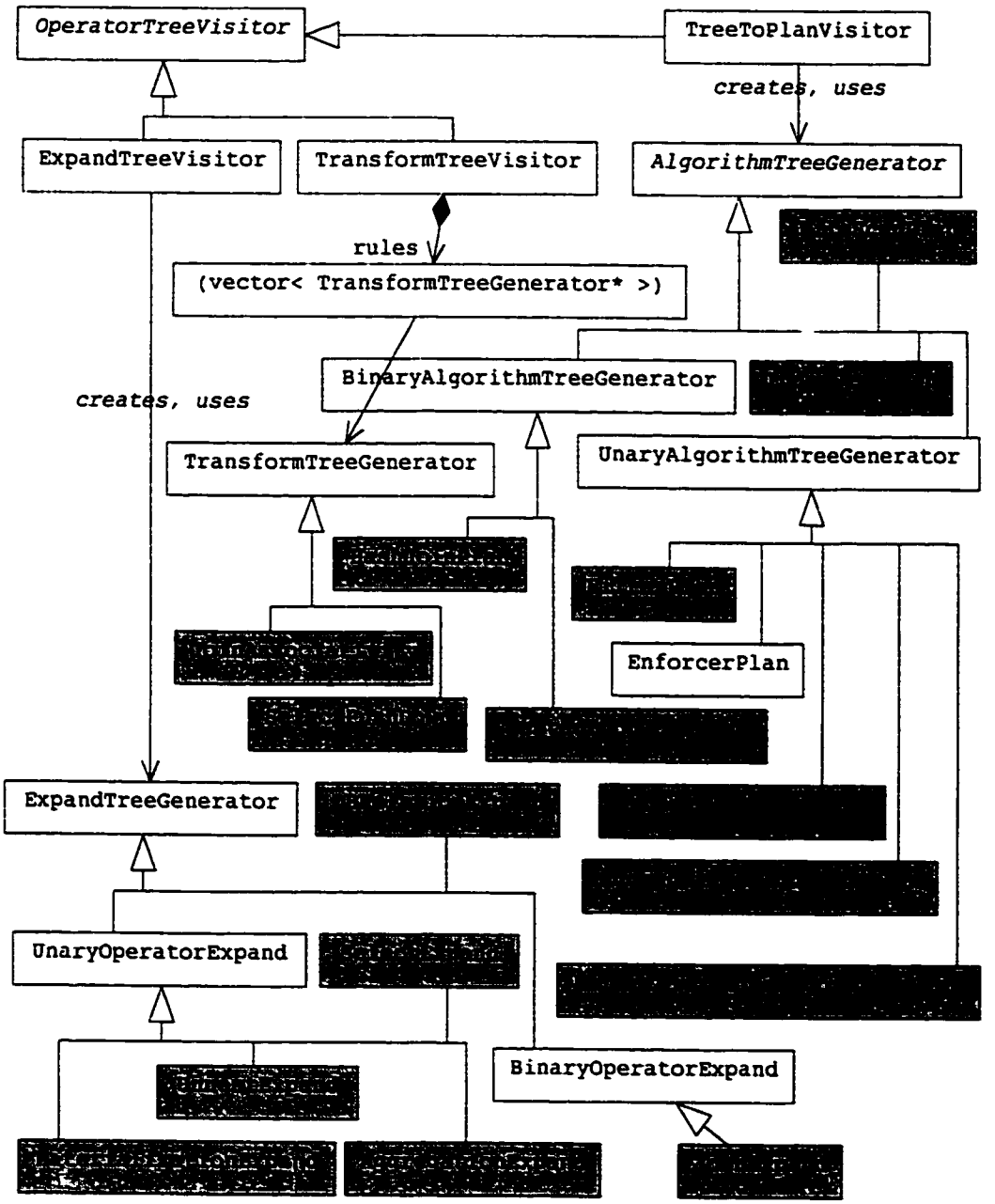

Figure 78: Complex Example: Customize the Search Space Component

Step 3: Customize the Search Strategy Component

1. Customize the Logical Properties. Six constructors are added to the class OPERATORTREEPROPERTY for logical operators DBRELATION, SELECT, JOIN, MATERIALIZATION, UNNEST, and AGGREGATION.

   Please refer to Example B in Section 5.3.1 for detailed customization information.

2. Customize the Physical Properties. Eight constructors are added to the class ALGORITHMTREEPROPERTY for physical operators FILESCAN, INDEXSCAN, FILTER, NESTEDLOOPJOIN, HASHJOIN, MATERIALIZATIONAGLORITHM, AGGREGATIONALGORITHM, and UNNESTALGORITHM.

   Please refer to Example B in Section 5.3.2 for detailed customization information.

3. Customize Cost Computation. Eight *Compute* methods are added to the class COST for physical operators FILESCAN, INDEXSCAN, FILTER, HASHJOIN, NESTEDLOOPJOIN, MATERIALIZATIONAGLORITHM, UNNESTALGORITHM, and AGGREGATIONALGORITHM.

   Please refer to Example B in Section 5.3.3 for detailed customization information.

4. Selection of Input Format for Queries. In the main control program, use the C++ code:

```
QueryOptimizerFacade* facade
                    = new QueryOptimizerFacadeWithParser;
```

5. Selection of the Search Strategies. Since this example uses the default search strategy for search. No additional code needs to write. That is, a simple line of C++ code is enough:

```
facade -> Optimize();
```

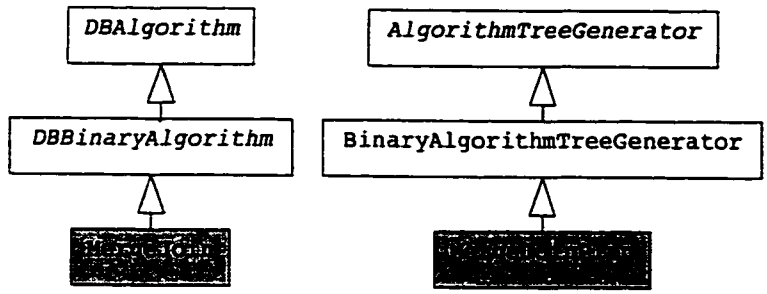

Figure 79: Add A Merge Join Algorithm

5.6 Putting It All Together — Modify A Query Optimization System

In this section, we will illustrate how to modify a query optimization system built from this query optimization framework. We will try to enumerate some potential changes that may be made to the query optimization system.

5.6.1 Add A New Execution Algorithm

Adding a new execution algorithm for an existing logical operator needs to extend both the DBALGORITHM hierarchy and the ALGORITHMTREEGENERATOR hierarchy. For example, we want to add a MERGEJOIN execution algorithm for the JOIN operator. Figure 79 shows the changes of these two hierarchies.

The newly added classes are adorned in gray. Since the JOIN operator is a binary operator, class MERGEJOIN inherits from the class DBBINARYALGORITHM and its expand class MERGEJOINPLAN inherits from the class BINARYALGOIRTHMGENERATOR. Class MERGEJOIN should implement methods *Duplicate* and *MakePhyProps*. It may also need to re-write the method *Clones* to apply this operator to its inputs with different ways. Class MERGEJOINPLAN may need to re-write the method *CanBeApplied* to clarify the conditions it can be applied to build an algorithm tree. It may also re-write the method *MakePhyNodes* to change the behavior when an algorithm tree is built by that plan class.

The OPERATORDEFINITION class should be modified. An enumeration number for this algorithm should be given and this physical operator should be added to the list of algorithms associated with its related logical operator, JOIN. The class ALGOIRTHMTREEPROPERTY should also be changed to add a new constructor that

is used to set up the physical properties for this physical operator. Moreover, the *Compute* method regarding to the execution cost of this algorithm should be added to the COST class.

5.6.2 Add A New Logical Operator

Adding a new logical operator (for example, MyOperator) involves:

- Add a class MYOPERATOR representing the new logical operator to the DB-OPERATOR hierarchy.
- Add a constructor to the class OPERATORTREEPROPERTY to compute the logical properties associated with this operator.
- Add its corresponding execution algorithms as that described in Section 5.6.1.
- Add a method

  ```
  virtual void VisitMyOperator ( MyOperator* op ) = 0 ;
  ```

  to the OPERATORTREEVISITOR class definition.
- Add methods

  ```
  virtual void VisitMyOperator ( MyOperator* op );
  ```

  to classes EXPANDTREEVISITOR, TRANSFORMTREEVISITOR, and TREETO-PLANVISITOR.
- Add a class MYOPERATOREXPAND to the EXPANDTREEGENERATOR class hierarchy.
- May need to add transformation rules related to MYOPERATOR to the TRANS-FORMTREEGENERATOR class hierarchy.

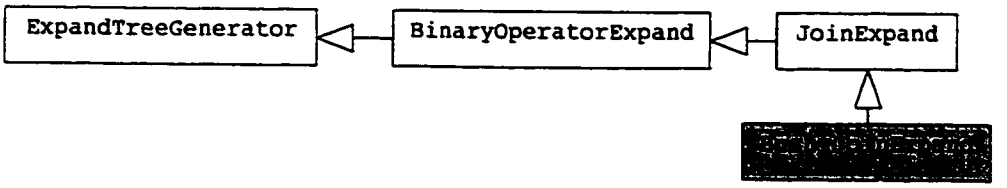

Figure 80: Limit Search Space to Bushy Join Trees

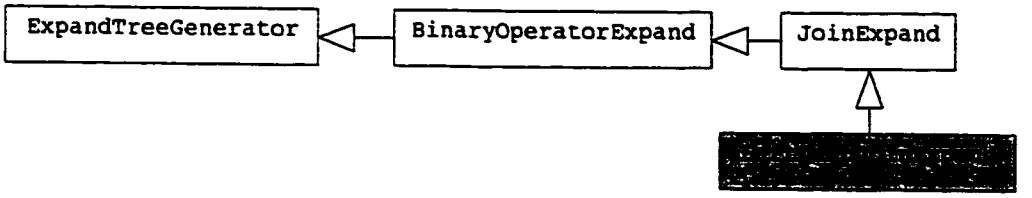

Figure 81: Limit Search Space to Left Deep Join Trees

5.6.3 Limit or Extend the Search Space

The Search Space component is in charge of creating any logical query plans and physical query plans. How the search space is shaped will affect the quality of the final physical query plan. The query optimization system can be experimented with changes to the search space by limiting it or extending it. No matter what the changes will be, the modification only limits to the scope of the Search Space component.

Example A Suppose we want to limit the search space to only contain the bushy join trees. A new class BUSHYJOINEXPAND is defined to subclass the class JOINEXPAND. BUSHYJOINEXPAND overrides the *Apply* method to eliminate any creation of left-deep-join trees and right-deep-join trees. The new class will be used in places where the JOINEXPAND is used. Figure 80 shows the newly added class in its class hierarchy.

Example B Suppose we want to limit the search space to only contain the left-deep-join trees. A new class LEFTDEEPJOINEXPAND is defined to subclass the class JOINEXPAND. LEFTDEEPJOINEXPAND overrides the *Apply* method to eliminate any creation of bushy-join trees and right-deep-join trees. The new class will be used in places where the JOINEXPAND is used. Figure 81 shows the newly added class in its class hierarchy.

In general, a binary tree is left-deep if all its right children are leaves. But in fact, these "leaves" can be interior nodes with operators other than JOIN. If we

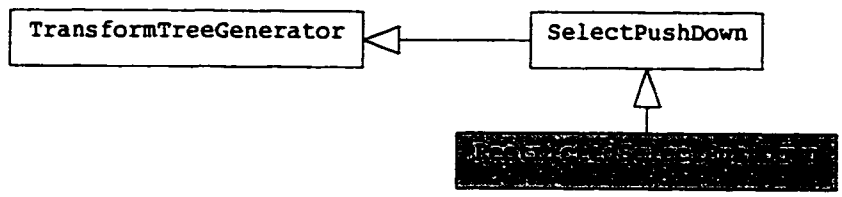

Figure 82: Use Restricted Select Push Down Rule

want to avoid all right children from interior nodes, we can define a new algebra law named RESTRICTEDSELECTPUSHDOWN as a subclass of the SELECTPUSHDOWN (see Figure 82). This new class is used to substitute the original SELECTPUSHDOWN to enforce the SELECT operator can not push down to the right inputs.

Example C Suppose we want to extend the search space that is used to explore the best plan.

The search space can be extended by introducing any new transformation rules. For example, we can define a new rule that introduces a projection anywhere in the operator tree, as long as it eliminates useless attributes that will not be used any more by the subsequent processing and that are not in the resultant attributes. It is obvious that use of this rule will result in several more equivalent operator trees for the original one that will lead to generation of more physical query plans that can be evaluated and compared. Chances of locating the physical query plans with less costs thus increase.

Any modification to the Search Space component due an addition of a physical operator and/or logical operator can also extend the search space explored.

5.6.4 Change Search Strategies

Changes of the search strategies would not affect the logical and physical operators defined in the database algebra and the search space that is used to explore the optimal plan. (We assume the search space designed in this query optimization framework is enough for any potential search strategy to use.) So modification is limited in the Search Strategy component.

This query opimization framework has equipped with two search strategies. One is the Bottom-Up Search Strategy, the other is the Transformative Search Strategy.

Example A Suppose we want to use the Transformative Search Strategy for search. Also suppose this system can accept user-input queries from keyboard. Then in the main control program, the following C++ code is added:

```
QueryOptimizerFacade* facade = new QueryOptimizerFacadeWithParser;
SearchStrategy* s = new TransformativeSearchStrategy;
facade -> SetSearchStrategy (s);
facade -> Optimize();
```

Example B Suppose we want to restore the Bottom-Up Search Strategy for search. Then remove the middle two lines from Example A, that is, the C++ code is changed to:

```
QueryOptimizerFacade* facade = new QueryOptimizerFacadeWithParser;
facade -> Optimize();
```

Example C Suppose we want to add a randomized search strategy for search, e.g. the Simulated Annealing Search Strategy. Since the Simulated Annealing Search Strategy uses both tree transformation and conversion from the logical query plans to physical query plans, we do not need to change anything in the Search Space component.

First, we need to define the SIMULATEDANNEALINGSEARCHSTRATEGY as an immediate subclass of the base class SEARCHSTRATEGY. Then a search tree is defined in the SEARCHTREE class hierarchy, we call it the SIMULATEDANNEALINGSEARCHTREE. There are two ways to define the class SIMULATEDANNEALINGSEARCHTREE:

- It is defined as an immediate subclass of the class SEARCHTREE. The optimizer-implementor needs to write the method *MakeInitialTree* as that in the TRANSFORMEDSEARCHTREE to create a logical query plan representing the complete query. S/he also needs to override the method *DoSearch* to perform a random selection on options tree and plan. If the plan option is chosen, the current operator tree is transformed into algorithm trees and cost evaluation is performed. If the tree option is chosen, the current operator tree is transformed into an equivalent operator tree, the algorithm trees for the new tree are created, and the cost evaluation is performed.

- It is defined as a subclass of the TRANSFORMEDSEARCHTREE. The optimizer-implementor need not define the method *MakeInitialTree*. S/he only re-writes the method *DoSearch* with algorithm specified above.

The use of the Simulated Annealing Search Strategy is similar to that of the TRANSFORMATIVESEARCHSTRATEGY. That is,

```
QueryOptimizerFacade* facade = new QueryOptimizerFacadeWithParser;
SearchStrategy* s = new SimulatedAnnealingSearchStrategy;
facade -> SetSearchStrategy (s);
facade -> Optimize();
```

Chapter 6

Framework Cookbook

This chapter provides a general guideline on how to customize this query optimization framework. It is organized into a set of recipes, each is a description of a typical customization on one aspect.

6.1 Recipe 1: Overview of the Cookbook

This first recipe provides an overview of this cookbook. Recipes 2 to 4 are descriptions of customization of individual components of this framework. Recipes 5 and 6 are the descriptions of the customization of the interfaces of the system built from this framework. Each recipe follows a structure including its purpose, steps involved in customization, and cross-reference to other recipes.

6.2 Recipe 2: Customize the Algebra Component

Purpose Define the logical operators and physical operators in the database.

Steps

1. Define logical operators

   - Define a concrete subclass in the DBOPERATOR hierarchy for each logical operator.

   - Implement the following methods:

Accept — Accepts potential operations (represented by a kind of OPERATORTREE-VISITOR) on this operator.
Duplicate — Copies itself.
MakeLogProps — Creates logical properties for the operator tree rooted at this operator.

- The following methods may be overridden:
  Clones — Provides different ways to apply this operator to its inputs, with different parameters.

2. Define physical operators

   - Define a concrete subclass in the DBALGORITHM hierarchy for each physical operator.

   - Implement the following methods:
     Duplicate — Copies itself.
     MakePhyProps — Creates physical properties for the algorithm tree rooted at this operator.

   - May implement the following methods:
     MakePhyNodes — Builds the algorithm trees for an operator tree by delegating to the Search Space component.

   - The following methods may be overridden:
     Clones — Provides different ways to apply this operator to its inputs, with different parameters.

3. Define class OPERATORDEFINITION

   - Assign an enumeration number to each logical operator.

   - Assign an enumeration number to each physical operator.

   - Set up the relationships between the logical operators and the physical operators.

Cross-Reference None

6.3 Recipe 3: Customize the Search Space Component

Purpose Define the shape of the search space that is used to explore the optimal plan.

Steps

1. Add abstract methods to the class definition of OPERATORTREEVISITOR for each logical operator XX with the following format:

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

2. Add a method to the class definitions of EXPANDTREEVISITOR, TRANSFORMTREEVISITOR, and TREETOPLANVISITOR for each logical operator XX with the following format:

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

3. Define a concrete subclass in the EXPANDTREEGENERATOR hierarchy for each logical operator.

- Each new class name is the concatenation of the name of the corresponding logical operator and the string "Expand".
- The following methods may be overridden:
  Apply — Determines how an operator tree is built by applying this logical operator.
  DfsNode — Performs a depth first search on the search tree to locate suitable nodes that can be paired with the given operator tree parameter.

4. Define a concrete subclass in the ALGORITHMTREEGENERATOR hierarchy for each physical operator.

- Each new class name is the concatenation of the name of the corresponding physical operator and the string "Plan".
- The following methods may be overridden:
  CanBeApplied -- Returns true if this physical operator can apply to the given

algorithm tree inputs.

MakePhyNodes — Creates the corresponding algorithm trees for a given operator tree.

5. Define concrete subclasses that represent tree transformation rules in the TRANSFORMTREEGENERATOR hierarchy for related logical operators.
6. Implement the methods defined in step 2 by delegating to appropriate generator concrete classes.

Cross-Reference Recipe 2

6.4 Recipe 4: Customize the Search Strategy Component

Purpose Define search tree properties, cost information, and strategies for search.

Steps

1. Define logical properties

   - Add a constructor to the class OPERATORTREEPROPERTY for each logical operator XX with format:

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

     The second parameter refers to the operator tree whose logical properties are to be created.

   - May re-define the definitions for interesting logical properties and what equivalence of two operator trees means.

   - May add some other logical properties to this class definition.

2. Define physical properties

   - Add a constructor to the class ALGORITHMTREEPROPERTY for each physical operator XX with format:

```
AlgorithmTreeProperty ( XX*, AlgorithmTree* );
```

The second parameter refers to the algorithm tree whose physical properties are to be created.

- May re-define the definitions for interesting physical properties and what equivalence of two algorithm trees means.
- May add some other physical properties to this class definition.

3. Define execution cost value.

- Add a *Compute* method to the class COST for each physical operator XX with format:

```
Void Compute ( XX*, AlgorithmTree* );
```

The second parameter refers to the algorithm tree whose execution cost is to compute.

4. Selection of search strategies. Suppose facade is an instantiated object of a concrete subclass of the QUERYOPTIMIZERFACADE.

- Use of the Bottomup Search Strategy with a direct call:

```
facade -> Optimize();
```

- Use of the Transformative Search Strategy with C++ code:

```
SearchStrategy* s = new TransformativeSearchStrategy;
facade -> SetSearchStrategy (s);
facade -> Optimize();
```

- Use of a randomized search strategy includes four steps:
  1. Define a subclass XX for this new search strategy in the SEARCHSTRATEGY hierarchy.
  2. Define a subclass in the SEARCHTREE hierarchy and use it for search by the newly defined search strategy XX.

3. Implement the following methods in the class defined in step 2:
MakeInitialTree — Creates an operator tree that represents the complete query.
DoSearch — Builds the search tree based on the initial operator tree using the randomized approach.
4. Use it with C++ code:

```
SearchStrategy* s = new XX;
facade -> SetSearchStrategy (s);
facade -> Optimize();
```

Cross-Reference Recipe 2, Recipe 3

6.5 Recipe 5: Customize Inputs to the Query Optimization System

Purpose Define interface to the client (optimizer-implementor, DBA, or DBMS) for input.

Steps

1. Define query input format. This framework defines the following two query input formats:

- Query written in a query language. In this case, use the following C++ code to instantiate a facade object:

```
QueryOptimzerFacade* facade = new QueryOptimzerFacadeWithParser;
```

- Query that is parsed outside this system and is turned into a file **.parsedquery** (see Section 5.5.1). In this case, use the following C++ code to instantiate a facade object:

```
QueryOptimzerFacade* facade
                     = new QueryOptimzerFacadeWithFormatedFile;
```

2. Define the system catalog. The system catalog includes information for relations, attributes, methods, indexes, and cost (e.g. object size). Please refer to Section 5.4 for more details.

Cross-Reference Recipe 4

6.6 Recipe 6: Customize Output of the Query Optimization System

Purpose Define interface to the DBMS for output.

Steps

1. Define a converter class with functionality of recursively converting each node in the optimal plan to a corresponding node in the executable plan tree.
2. Implement all methods in the class defined in step 1.

Cross-Reference None

Chapter 7

Conclusion

Although query optimization has been studied for decades, developing, modifying, and extending a query optimization system is still difficult. The problem with existing query optimization frameworks is that either the addition of new query operators/algorithms is fixed or the search strategy that is used to explore the optimal plan is fixed with respect to the query algebra.

Kabra [20] develops a query optimization framework named OPT++ that uses object-oriented design to simplify the task of developing, extending, and modifying a query optimization system. First, it allows new operators/algorithms to be easily added. Second, it offers a choice of different search strategies so that various heuristics can be used to limit or extend the search space explored. Third, the flexibility in both the query algebra and the search strategy can be achieved without compromising the efficiency of the system. The biggest advantage of a query optimization system built from OPT++ is that it accommodates most existing extensible query optimization systems without sacrificing the performance while still maintaining a more flexible structure. Although OPT++ is very robust, it has a few drawbacks such as partial implementation of the design; poor partition of the framework; non-reusable system design; switching from one search strategy to another may be hard; framework documentation is insufficient, etc.

This thesis proposes a reusable architecture for extensible query optimization. A query optimization system is physically divided into three major components. The framework is designed to span across these components, where each contributes to a single purpose in the system. Design patterns and object-oriented techniques are used

to de-couple these components and improve the flexibility within each component. This thesis also makes a clear separation of the search strategy and the search tree. This separation conforms to a design convention, that is, separate interface from implementation. We believe this separation promotes the reusability of the system and is good for clarity. Also, we define a search strategy interface that allows different search strategies to be easily installed and be used interchangeably. Switching from one search strategy to another only requires modification of two lines of code within the same component. The design is further verified in light of implementation in C++. Moreover, we believe the documentation of a framework is as important as its design. We attempt to provide a series of framework documents to assist the application developer to better reuse it. These documents include the framework overview, the framework design, examples of customization, and the framework cookbook.

We understand the importance of efficiency in query optimization. As we have mentioned before, query optimization directly affects the performance that the user sees. OPT++ has done performance comparison on different search strategies (Bottomup, Transformation, and Randomized) and comparison on the transformation query optimization system built from OPT++ and that built from the Volcano Optimizer Generator. It concludes that:

- For smaller queries, the exhaustive algorithms (Bottomup and Transformation) consume much less time for optimization than the randomized algorithms and yet produce equivalent or better plans. For larger queries, the randomized algorithms take much less time to find plans that are almost as good as those found by the exhaustive algorithms.

- The randomized strategies require a negligible amount of memory irrespective of the size of the input query, while the exhaustive strategies require exponentially increasing amounts of memory. Hence, for queries that are large enough, the randomized strategies will continue to give reasonable performance while the exhaustive strategies will fail due to lack of enough memory.

- Although the Bottomup and Transformation search strategies have comparable performance in terms of optimization time and quality of produced plans (because both are exhaustive strategies and explore the same search space), the Bottomup strategy has a significant advantage in space consumption as it can

perform more aggressive pruning of operator trees.

- The Transformation Search Strategy of OPT++ is almost as good as that of the Volcano search engine, with a degradation of only about 5% in the optimization time, while space utilization is roughly equivalent.

We reuse the core code in OPT++. We believe this part of code is the major contribution to the performance of a query optimization system. Although we do not do the performance measurement on the query optimization system built from this framework, we can simply conclude that the performance of the system built from this framework is roughly as good as that built from OPT++, with a negligible degree of degradation. But this framework provides a more general and reusable structure, and at the same time, allows the major part code that contributes to efficiency to be fine tuned with minimum impact on the rest of the system.

Reusable design results from evolution and iteration. The future work is to develop a series of graded applications using this framework. In particular, the following fields should be considered:

- Object-oriented database system.
- Use of Multidimensional index.
- Complex algebraic laws.

At this time, we are not clear how general this framework is to be easily reused in developing applications in the above fields.

We may need to refine the framework to incorporate a memory mechanism so that the same query will not be optimized twice. In other words, if the history shows that a processing query has been optimized before, its optimal plan is directly picked as the result of optimization without actually performing the optimization. Furthermore, the file representations of a parsed query and the system catalog should be improved as class interfaces. They also need to be generalized to accommodate various complex query language syntax.

Difficulties encountered in this thesis work lie in the understanding of the problem domain and the reuse of existing query optimization frameworks. They helped me understand the importance to clarify some important concepts in query optimization

domain and to view the query optimization process from an object-oriented perspective. They further convince me to develop a reusable object-oriented framework for the extensible query optimization domain, to seek means to promote its reusability, and to maximize the benefits of reusing it.

This research work is a good trial towards a reusable object-oriented framework for extensible query optimization.

Bibliography

[1] Alfons Kemper, Guido Moerkotte, and Klaus Peithner. A Blackboard Architecture for Query Optimization in Object Bases. *Proceedings of the 19th VLDB Conference*, 1993.

[2] B. Stroustrup. *The C++ Programming Language, third edition.* Addison-Wesley, 1997.

[3] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[4] Edward Sciore and John Scig Jr. A Modular query Optimizer Generator. *Proceedings of IEEE Conference on Data Engineering, Los Angeles, California*, February 1990.

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

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons. Inc., 1996.

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

[8] G. Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *Proceedings of IEEE Conference on Data Engineering, Vienna, Austria*, 1993.

[9] Gail Mitchell, Umeshwar Dayal, and Stanley B. Zdonik. Control of an Extensible Query Optimzier: A Planning Based Approach. *Proceedings of the 19th VLDB Conference, Dublin, Ireland*, 1993.

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

[11] Gregory Butler and Pierre Dénommée. *Documenting Frameworks to Assist Application Developers, Chapter of Building Application Frameworks: Object-Oriented Foundations of Framework Design, edited by Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson.* John Wiley & Sons. Inc., 1999.

[12] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation.* Prentice Hall, 2000.

[13] I. Jacobson, M. Christorson, P. Johnsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, Mass., 1992.

[14] Ian Sommerville. *Software Engineering, fifth edition.* Addison-Wesley, 1997.

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

[16] 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*, pages 55–78, 1988.

[17] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley & Sons. Inc., 1999.

[18] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Implementing Application Frameworks: Object-Oriented Framework at Work.* John Wiley & Sons. Inc., 1999.

[19] Navin Kabra. OPT++ User Manual. *Technical Report, University of Wisconsin*, 1995.

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

[21] Raghu Ramakrishnan. *Database Management Systems.* McGraw-Hill, 1997.

[22] R.E. Johnson. Documenting Frameworks using Patterns. *Proceedings of OOPSLA'92, ACM/SIGPLAN*, pages 63–76, 1992.

[23] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-OrientedProgramming, 1*, pages 22–35, 1988.

[24] R.J.A. Buhr and R.S. Casselman. Architectures with Pictures. *Proceedings of OOPSLA'92. ACM/SiGPLAN, New York*, pages 466–483, 1992.

[25] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the Search Strategy in a Query Optimzier. *Proceedings of the 17th VLDB Conference, Barcelona*, September 1991.

[26] Steven P. Reiss. *A Practical Introduction To Software Design With C++.* John Wiley & Sons. Inc., 1998.

[27] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Seattle*, pages 34–43, June 1998.

[28] Taligent Inc. Building Object-Oriented Frameworks. *A Taligent White Paper*, 1994.

[29] Timothy Budd. *An Introduction to Object-Oriented Programming, second edition.* Addison-Wesley, 1997.

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

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