REENGINEERING AN OBJECT-ORIENTED FRAMEWORK FOR EXTENSIBLE QUERY OPTIMIZATION

QIU WEN 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

JULY 2004

Your file Votre référence
ISBN: 0-612-94746-7
Our file Notre référence
ISBN: 0-612-94746-7

Abstract

Reengineering an object-oriented framework for extensible query optimization

Qiu Wen Li

In this thesis, we describe a third-generation extensible query optimization framework that has evolved from the OPT++ framework of Navin Kabra for relational databases. Our framework does not change the infrastructure of the OPT++ architecture, which consists of three components: a Search Strategy component, a Search Space component and an Algebra component. However, we address the problems encountered while building two query optimizers in the framework: a simple bottom-up optimizer and an instance of the PostgreSQL query optimizer and enhance the modularity and the collaboration of the three components of the framework at detailed level, which in turn leads to a more flexible, easier to extend and cleaner implementation. The framework has been validated by implementing the two query optimizers. While both these cases were for the relational data model, we believe the design does cover the optimization process for other data models.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Dr. Gregory Butler. His encouragement and invaluable guidance make my thesis work a pleasant and extremely educational experience.

I would like to thank my colleague Ju Wang for his valuable suggestions and the helpful discussions we had. He also provided the performance test result.

I would also like to thank Dr. Peter Grogono and Dr. Volker Haarslev for reading and correcting my thesis.

Finally, I would like to thank my husband, Qiang Tang, my mother, Ming Xiao Deng, and my father, Xian Biao Li. There is no doubt that without their unconditional love and support, I would not have gotten through my study at Concordia University.

Table of Contents

List of Figures

List of Table

Chapter 1

Introduction

Query optimization, which is one of the critical processes in a database management system (DBMS), has been studied for over twenty years, but building a query optimizer to handle it in a DBMS is still a difficult and expensive task. Furthermore, new query algebra and search techniques are continuously introduced. Different DBMS might not support exactly the same query algebra, and also there is no one query optimization technique that is best for all queries, *i.e.* different search strategies might generate different "good" execution plans for a certain query. As a result, there is a need to build an extensible query optimizer, which allows easily adding and modifying query algebra and meanwhile can easily switch among different search strategies. A lot of effort was put in attempting to make such changes easier while building a query optimizer. Representative examples include the Starburst query optimizer [18] [21], the Volcano optimizer generator [16], the EXODUS optimizer generator [17] and the OPT++ framework [20].

Frameworks are an object-oriented reuse technology and have been applied to the query optimization domain to meet the reuse demand. In this thesis, we present the

design and implementation of a third-generation query optimization framework evolved from the OPT++ framework, which satisfies both extensibility requirements presented above, along with the improved performance of an instance of the PostgreSQL query optimizer [29] customized from the framework.

1.1 Related Work and the Problem

OPT++, which is proposed by Navin Kabra for his PHD thesis at the University of Wisconsin, is an extensible query optimization framework that allows the extensibilities of both adding new algebraic operators/algorithms and changing search strategies. OPT++ is written in C++ and exploits the object-oriented features, such as the reusability and dynamic binding of C++. It is mainly divided into three components [20]: the Search Strategy component, the Search Space component and the Algebra component. The Search Strategy component determines what strategy is used to explore the search space (*e.g.*, dynamic-programming, randomized, *etc.*); the Search Space component determines what that search space is (*e.g.*, space of left-deep join trees, space of bushy join trees, *etc.*); and the Algebra component determines the actual logical and physical algebra for which the optimizer is written. The modularity of the three-component makes it possible to modify one of the three components while having minimum impact on the other two. For example, different search strategies can be applied on the same query algebra, and expanding the query algebra will not affect the implementation of the search logic.

In spite of the big improvement made by OPT++ in designing an extensible

query optimization framework, OPT++ has some limitations, such as the framework is not completely implemented, components are strongly coupled by implementation details, and so on, as stated in Jinmiao Li's thesis [22]. Jinmiao Li refined OPT++, which forms the second-generation framework, and customized it to a simple bottom-up query optimizer.

A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [19]. The best way to tell if a framework is reusable for a domain is to customize it. Plus, OPT++ has not been fully used besides its author and the simple bottom-up query optimizer implemented by Jinmiao Li. Therefore, further studies on OPT++ are necessary. Ju Wang implemented an instance of the PostgreSQL query optimizer on top of the second-generation framework [36]. He also extended the framework to allow sub-queries and explicit joins.

During the course of building the two applications *(i.e.* the simple bottom-up query optimizer and the PostgreSQL-like query optimizer), we found substantial need to improve the design at the detailed level though the main abstractions of OPT++ did not change. Moreover, the second application raised many issues with the performance of OPT++. In this thesis, we further study the query optimization framework. Our work enhances the detailed design and implementation of the three-component architecture and dramatically improves the performance of the PostgreSQL-like query optimizer.

1.2 Objective and Scope of the Thesis

The objective of the thesis is to reengineer the detailed design and implementation of the second-generation query optimization framework that originally derived from OPT++ and to further evaluate OPT++ as an extensible query optimization framework within a database management system. The reengineering aims to improve the extensibility, flexibility and understandability of the framework and experience framework-base system development. It addresses the problems encountered while building the two optimizer applications: the simple bottom-up query optimizer and the PostgreSQL-like query optimizer, which include the performance issues raised by the second application.

The scope of the thesis basically includes: understanding the framework; analyzing the problems of the framework and solving the problems; integrating the PostgreSQL-like optimizer to the new framework and improving its performance.

- Understanding the framework

  The first step of working with a framework is to understand it. And because OPT++ is a white-box framework, adequate knowledge on the internal implementation of the framework is a must. Moreover, query optimization is a complex process. Understanding the query optimization framework is not a minor work, but takes a considerable portion of time of the whole project.

- Analyzing the problems

  After getting sufficient knowledge on the framework, we move to the next step – identifying the problems, which involves the following steps:

1. Review the previous framework.
2. Analyze the two framework applications: the simple bottom-up query optimizer and the PostgreSQL-like query optimizer.
3. Investigate the problems that cause the difficulties in the implementation of the two applications.

- Reengineering the framework to solve the encountered problems and improve the flexibility and extensibility of the framework.
  1. Abstract a query optimizer object and localize the global variables that limit the flexibility of the framework to the query optimizer object.
  2. Decouple the three components and reassign the functionalities of each component.
  3. Abstract new objects (*e.g.* interface to the Search Space component) to provide clearer interfaces for the components to interact with each other.
- Integrating the PostgreSQL-like optimizer to the new framework. Profiling the PostgreSQL-like optimizer to identify and fix the performance problems. As mentioned previously, the PostgreSQL-like optimizer has raised many performance issues. The test results presented in Ju Wang's thesis [36] shows a big gap of performance in terms of the time needed to optimize queries between the PostgreSQL-like optimizer and the native PostgreSQL optimizer. The application takes 8-10 times the time used by the native PostgreSQL optimizer when the number of joins in a query exceeds nine.

1.3 Contribution of the Thesis

This thesis proposes a third-generation extensible object-oriented framework for database query optimization that originally derived from OPT++. The fitness of OPT++ framework as a query optimization sub-framework within a database management system framework, *e.g.* the Know-It-All framework [3], is further studied.

The new generation improves the reusability and extensibility of the framework and provides cleaner and more understandable program codes. An object that represents the query optimizer is constructed and template technology is applied. The query optimizer object provides a simple manner to switch among different search strategies and different sets of query algebra. The interfaces to the three components are abstracted or refined, and the functionalities among the components are reassigned, which clearly decouple the three components and produce more extensibility.

The difficulties and problems encountered while implementing the simple bottom-up optimizer and the PostgreSQL-like optimizer in the framework are addressed in this thesis. For example, heavy coupling between the Search Strategy component and the Algebra component, incomplete implementation of Search Space component, limitation caused by the global variables, and so on.

The PostgreSQL-like optimizer has been further studied and is integrated to the new framework. The performance of the optimizer has been obviously improved as shown in Table 7. For example, to optimize a query with nine joins, the time used drops from 10 times slower than the native PostgreSQL optimizer to around 4 times.

And the time drops from 6.6 times to 1.7 times when the number of joins reaches 20, where the genetic algorithm is used and a fixed size (1024 by default) pool of join plans are needed to initialized.

1.4 Layout of the Thesis

The organization of the thesis is as follows. *Chapter 2* gives the basic background knowledge on frameworks, database query optimization, OPT++, and the PostgreSQL query optimizer. *Chapter 3* describes the reengineering of the framework. In this chapter, we demonstrate the architecture and abstraction of the framework, and then review the second-generation framework. Finally, we present the reengineering in the design phase and give a small discussion on the solutions. In *Chapter 4*, we demonstrate the implementation of the reengineering decisions presented in *chapter 3* in each component and the performance testing result of the PostgreSQL-like optimizer. *Chapter 5* finally concludes the thesis.

Chapter 2

Background

Our work is to reengineer an extensible query optimization framework, and the knowledge involved is mainly in two domains - frameworks and database query optimization. In this chapter, we provide corresponding background knowledge on both domains to help readers understand our work. OPT++, an extensible query optimization framework, on which our work is based, is introduced. And a mature optimizer, the PostgreSQL query optimizer, which was used as a case study of our framework, is also demonstrated.

2.1 Framework and Design Pattern

Frameworks are an object-oriented reuse technology and have become very popular for software development in both industry and academic. Many framework examples have been successfully used in software development, *e.g.* MFC and DCOM of Microsoft, *etc.* Then, what is a framework?

2.1.1 What is a framework

A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [19]. Object-oriented approach has been adopted to contribute to the reusability of frameworks. Hereafter, we mean "object-oriented framework" when we use just "framework".

A framework is not an application, while it is an application generator for a particular domain. "Hot spot" is the term used to represent variable aspects of a framework. Hot spots are places where specific requirements can be implemented. A hotspot is embedded in a component or class of the framework using a template method and hook methods. A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior. Within the template method there are calls to other operations: some calls are there to provide variable operation, these are the hook methods; while some calls are there to provide good decomposition of a complex algorithm into simpler steps. A hook method provides default behavior that subclasses can extend if necessary. Hook methods are often protected methods of the base abstract class. The public method is the template method, which is called by client classes, and the variability is provided by subclasses that override the hook methods but not the template method [35].

In contrast to hot spots, "frozen spots" represent the stable points of a framework, which are not supposed to be changed and actually it is very difficult to modify them in a framework. They define the skeleton of a framework and most of the time control the whole flow of an application built in the framework. The frozen spots of a

framework call the specific implementation of the hot spots of the framework provided by the framework users when a particular application customized from the framework is executed. That is why people use the famous Hollywood proverb – "Don't call us, we'll call you." – to describe frameworks.

Frameworks can be classified into white-box framework or black-box framework according to the ways in which it is extended. To work with a white-box framework, the user needs to have adequate knowledge on the internal architecture of the framework. Normally, the hot spots of a white-box framework are implemented by sub-classing the abstract classes and providing concrete implementation of behaviors of these classes. Unlike a white-box framework, the black-box framework does not require framework users to learn the detailed internal implementation of the framework. It is reused by composition, instead of inheritance.

OPT++, the extensible query optimization framework, on which our work is based, is a white-box framework. Therefore, to get a thorough understanding of OPT++ is the first step of the thesis.

2.1.2 Developing a framework

The primary benefits [9] of frameworks stem from the modularity, reusability, extensibility, *and* inversion of control they provide to developers, as described below:

- "**Modularity** -- Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and

implementation changes. This localization reduces the effort required to understand and maintain existing software."

- "**Reusability** -- The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability and interoperability of software."
- "**Extensibility** -- A framework enhances extensibility by providing explicit hook methods that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features."
- "**Inversion of control** -- The run-time architecture of a framework is characterized by an "inversion of control". This architecture enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing

on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end-users or packets arriving on communication ports)."

Even though a framework can provide so many advantages, it is not easy to develop a framework. Since a framework covers applications in a certain domain and has to consider all relevant concepts of the domain, developing a framework for a specific domain requires expert knowledge of that domain. The framework developer should have a strong ability to abstract generic concepts from a domain and the ability to predict different specific requirements that might occur within a domain in the future. Furthermore, developing a framework is not a one-time task and has to evolve repeatedly. Because a framework itself is not a complete application, its reusability, extensibility and correctness can be tested only after the framework is instantiated. It has become common that at least three applications needed to be customized before we can determine that a framework is reusable, extensible and efficient for developing applications in a certain domain.

Framework development can be summarized into three major stages: domain analysis, framework design, and framework instantiation [24]. Domain analysis attempts to discover the domain's requirements and possible future requirements. Some of the hot spots and frozen spots of the framework can be determined during this stage. The framework design phase defines the framework's abstractions. Hot spots and frozen spots are fully modeled, and the extensibility and flexibility proposed

in the domain analysis is outlined. The last stage is to validate the framework by customizing applications from the framework. If the extensibility and flexibility outlined in the second stage can not be satisfied, we need to go back to the first stage and redesign the framework.

2.1.3 Design Pattern

Patterns are the software-engineering concept most often coupled with frameworks and are discovered in most of well-designed software. In this respect, frameworks are not exceptions: design patterns are usually applied in building the architecture of a framework and in shaping the adaptation mechanisms in the hot spots [15]. Design patterns are helpful in achieving flexibility in framework design. And they are also very useful in framework documentation.

While both design pattern and frameworks abstract key aspects of a problem in a context and contribute to improving the reusability of software development, they are different in the following ways:

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

Two design patterns are employed in this thesis: the Facade Design Pattern and the Visitor Design Pattern.

2.2 Database Query Optimization

Query optimization is a very important procedure in a DBMS. It helps find an efficient execution plan of a user-input query on a database. The optimizer is the component that is responsible for carrying out the procedure.

2.2.1 Query Algebra

A query is written in a declarative form that specifies the results that are required, but the query does not present a procedure or algorithm for finding those results in the database. Structured Query Language (SQL) is a standard programming language that is used to represent a user query for getting information from or updating a database. A query in SQL is parsed and translated into the underlying relational data model. A data model is represented by an algebra of operators, such as *select, join, project, etc* that comprise the relational algebra. These are also called logical operators. A logical operator together with its inputs forms a tree, called an operator tree, which represents the particular operations on data and the order of the operations. Each logical operation may be implemented by several algorithms, *e.g. join* can be implemented by *hash-join* or *merge-join*, which are also called physical operators. Some algorithms have prerequisites to create necessary conditions. For example, a merge join requires the inputs to be sorted, so there is a sort physical operator; temporary tables may be created and indexed in order to facilitate a physical operator. Analogously, a physical operator together with it inputs forms a tree, called an algorithm tree, which represents the execution plan of how to compute the results of the query.

2.2.2 Search Strategy

Query optimization requires a method of selecting an optimal, or a good, physical query plan from the space of all physical query plans that are equivalent to the query. Given a query, the query optimizer can use different approaches to find the optimal execution plan. Optimization searching can basically be classified into two catalogs: exhaustive search and non-exhaustive search. Exhaustive search considers all the possibilities in the optimization and guarantees that a best execution plan will be found. However, exhaustive search is only feasible when the space is small. In many cases, the space is too large to exhaustively search, and heuristics must be applied to prune the search space. Such a heuristic is called a search strategy [35].

- Bottom-up search strategy – the tree is built from the leaves to the root. The initial trees all contain only one leaf, which represents a database entity within the query. To generate bigger trees, one or two (*e.g.* binary operator *join* in relational database) existing operator tree(s) is/are picked and expanded by adding an operation of the query as the root of the tree or the two trees. Before expanding the tree, the applicability of the operation on the tree has to be checked, such as, is the operation included in the tree already, and so on. For each new operator tree, all the corresponding execution plans are generated. Cost-based pruning of these execution plans is done to narrow the search space. Optimization is complete when none of the operator trees can be further expanded. The answer of the optimization (the optimal execution plan) is the cheapest execution plan that represents the complete input query.

- Transformative search strategy – a complete tree is built using other search strategies, for example, bottom-up search strategy. Then a transformation rule, *e.g.* Select-push-down, which is the most common transformation rule, is applied on the tree and produces an alternative, algebraically equivalent tree. Before applying the transforming rule, the applicability of the rule on the tree has to be checked. The optimization completes when none of the existing operator trees can be further transformed, and the cheapest execution plan that represents the complete query is returned.

2.2.3 Query Optimization

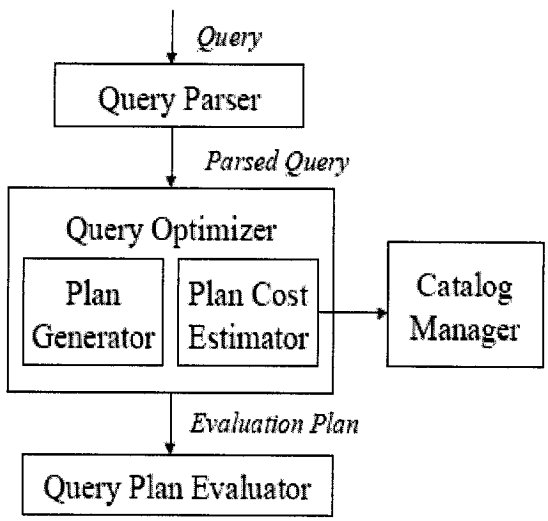

Figure 1: Query Parsing, Optimization, and Execution [31]

Figure 1 illustrates how query optimization is processed in a database management system. The optimizer takes the algebraic expression of a user-input query as the input. Based on the input query, it generates alternative plans for executing the query and

estimates the costs of the plans. The optimization stops when no more alternative plans can be generated and returns the best one, the plan with the lowest cost.

Obviously, the query algebra the optimizer is working on and the search strategy the optimizer will use to generate plans are two variable points in a query optimization process. Therefore, a query optimization framework which generates optimizers should provide flexibility and extensibility for these two points. OPT++, introduced in the next section, is an extensible query optimization framework that allows changes in both the query algebra and the search strategy.

2.3 OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization

OPT++ [20] is an extensible query optimization framework that aimed to allow changes to the query algebra and the search strategy without sacrificing efficiency.

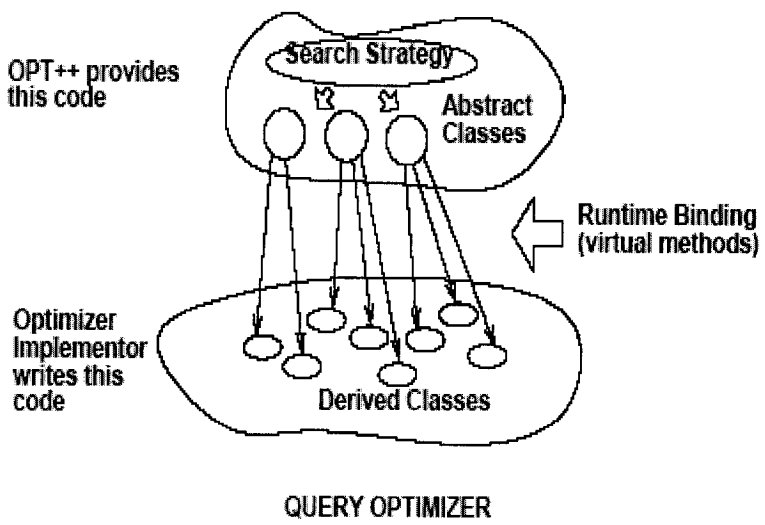

Figure 2: OPT++ Basic System Design [20]

OPT++ is a white-box framework and written in C++. It benefits from the object-oriented features of C++, such as, the inheritance and dynamic binding. Figure 2 shows the basic system design of OPT++. The search strategy is written in abstract classes. The optimizer implementer (OI) has to subclass these abstract classes for specific requirements. With the dynamic binding mechanism of C++, the search strategy will call the subclasses provided by the OI during the run time.

A comparison of OPT++ with Volcano [16], a research system, is made by Kabra. Both systems implement the same transformative optimizer, and OPT++ performs within 5% of Volcano in terms of the time to optimize a query using random queries with zero to twelve joins. No performance study of an OPT++ optimizer against an optimizer in production use is made.

2.3.1 The Three-Component Architecture

Figure 3 is an abstract overview of the OPT++ architecture, which is composed of three components: a Search Strategy component, a Search Space component and an Algebra component. The Search Strategy component determines what strategy is used to explore the search space (*e.g.*, dynamic-programming, randomized, *etc.*), the Search Space component determines what that search space is (*e.g.*, space of left-deep join trees, space of bushy join trees, *etc.*), and the Algebra component determines the actual logical and physical algebra for which the optimizer is written [20].

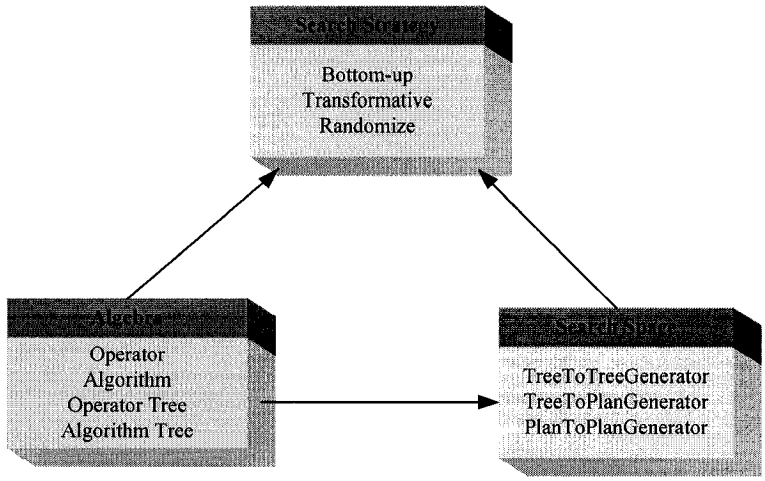

Figure 3: An Abstract Overview of the Three Components

The three-component decomposition conforms to the nature of query optimization. It provides a valuable guidance for building an extensible query optimization framework. The separation of Search Strategy component and Algebra component gains the flexibility of different search strategies can be applied on the same query algebra and expanding/modifying the query algebra will not affect the search logic. The clear separation between them becomes possible by introducing the Search Space component, which performs operations on the Algebra under the guide of the Search Strategy. One the whole, the well-separation of the three components makes it possible to change each of these components while having minimum impact on the other two.

2.3.2 Data Model

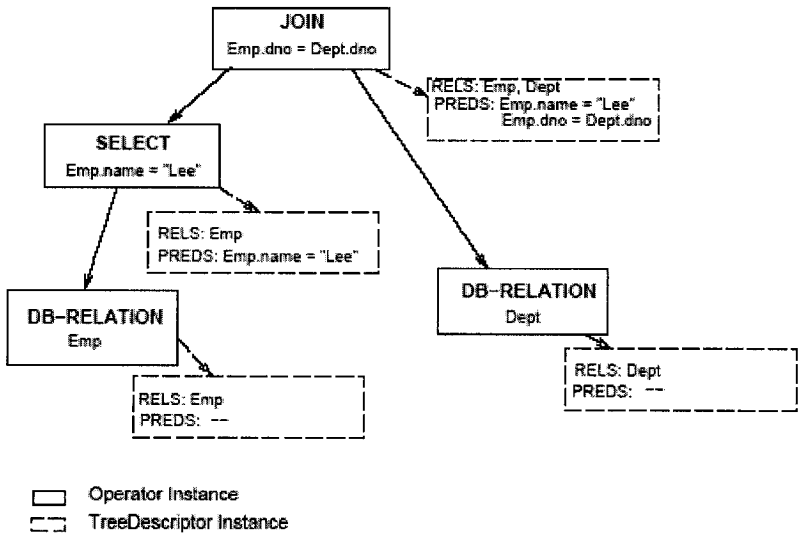

Figure 4: An Example Operator Tree with Its Properties [20]

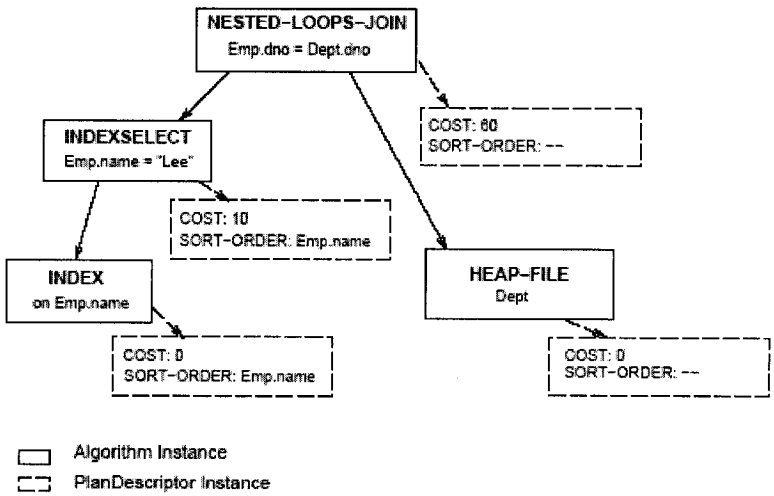

Figure 5: An Example Algorithm Tree with Its Properties [20]

The Algebra component implements the data model of the query optimization. Let us firstly look at some basic concepts in this component:

operator – represents a logical algebraic operator, *e.g. select, join, etc.*. An

operator has a list of *algorithms*, which represent its physical execution algorithms in a DBMS. For example, *join* can be implemented as *merge-join* or *hash-join*. An *operator* can take no input (*e.g. DB relation*), one input (called unary operator, *e.g. select*) and two inputs (called binary operator, *e.g. join*).

algorithm – represents an algebraic algorithm which is one of the actual implementations of an operator. Each *algorithm* associates with a method to calculate the ***cost*** in terms of execution time of the algorithm in a DBMS. The cost is the main criterion to judge an optimal tree. An algorithm can have an ***enforcer***, which represents the prerequisites to execute the *algorithm*. For example, *sort* has to be done in order to execute *merge-join*. Similar to an *operator*, an *algorithm* can take no input, one input (unary algorithm) and two inputs (binary algorithm).

operator tree – a tree structure in which each node is an *operator* being applied to its inputs.

algorithm tree – the implementations of each algebraic operator of an *operator tree* also found a tree called *algorithm tree*, in which each node analogously represents an *algorithm* being applied to its inputs.

operator tree descriptor – a descriptor of an *operator tree* that stores information about the tree, *e.g.* set of relations already joined in, the index path, and any other interesting information that can

distinguish the operator tree. The *operator tree descriptor* also helps to decide if two operator trees are algebraically equivalent.

algorithm tree descriptor – a descriptor of an *algorithm tree* that stores information about the tree, *e.g.* the sort-order of the result, the cost of the algorithm tree and any other interesting information that can distinguish the algorithm tree. Similar to the operator tree, the *algorithm tree descriptor* can be used to check if two *algorithm tree*s are algebraically equivalent.

Figure 4 and Figure 5 give examples of an operator tree (with its descriptor) and an algorithm tree (with its descriptor). In the framework, it is assumed that a query can be logically represented as an operator tree. And an operator tree has more than one algorithm tree because an algebraic operator can be implemented using more than one algebraic algorithm.

2.3.3 Cost Model and Pruning Mechanism

During the course of optimizing a query, the optimizer must generate various operator trees that represent the query (or parts of it) and their corresponding algorithm trees. As a result, the search space will becomes bigger and bigger. Cost model defines a criterion for search space pruning.

Each algorithm associates with a method to calculate the cost. Normally, the cost of an execution plan is the time the database uses to execute it. The optimal plan is the plan with the less cost among the set of algebraically equivalent execution plans.

Cost-based pruning [20] of access plans in OPT++ is done in a manner similar to the techniques used by the System-R optimizer [12], as illustrated in Figure 6. Whenever a new access plan is created, the virtual methods of the `Algorithm` class are used to determine the cost of that access plan, to determine whether it has any interesting physical properties, and to local all other access plans that are equivalent to it. From this set of equivalent access plans, only the cheapest plan and those plans that have interesting physical properties are retained. All others are deleted. An operator tree is deleted if all its algorithm trees are pruned.

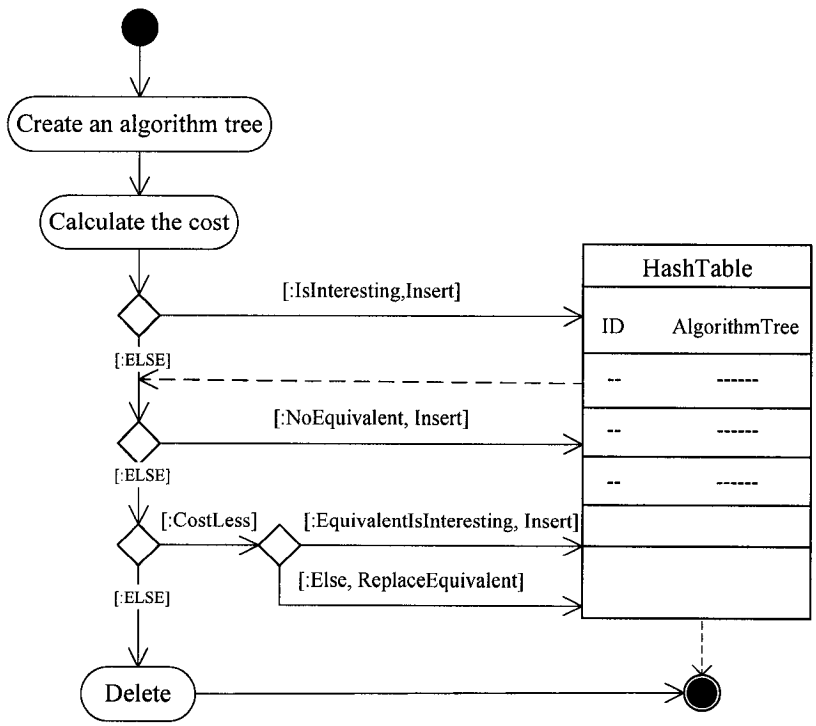

Figure 6: OPT++ Cost-based Pruning Mechanism

2.4 PostgreSQL Query Optimization

PostgreSQL [29] is an open source object-relational database management system. The search strategies implemented in PostgreSQL optimizer includes transformative

rules, constrained dynamic programming and the genetic algorithm. The latter two are implemented in the PostgreSQL-like optimizer built in our framework. The framework design is also extended to allow sub-queries and explicit joins which are essential features of PostgreSQL optimizer.

2.4.1 Data Model

The PostgreSQL optimizer was written in C, thus the data structures were implemented in C Structs. The data structure that represents a query is called Struct `Query`. There is no data structure defined for an operator tree in PostgreSQL because the PostgreSQL optimizer produces algorithm trees directly instead of generating operator trees and then converting them into algorithm trees as done in OPT++. The algorithm tree is represented by Struct `Plan`.

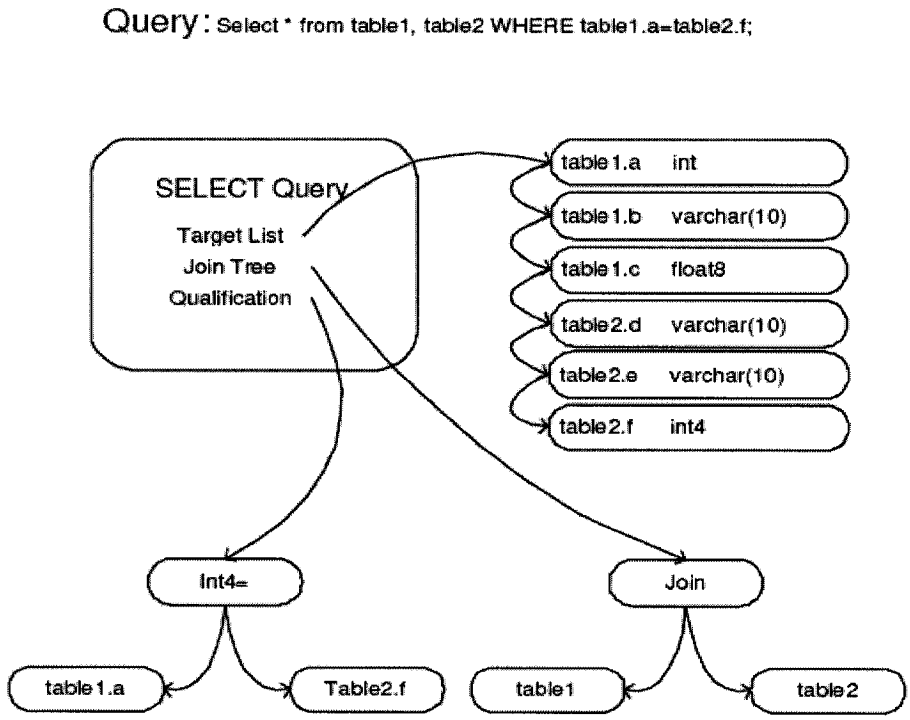

Figure 7: A PostgreSQL Query Tree [29]

Struct `Query` The information of a user input query is parsed and saved in Struct `Query`. Figure 7 illustrates the Struct `Query`, which is composed of the following parts [8] [35]:

- **The command type** that takes its value from an enum type consisting of SELECT, INSERT, UPDATE or DELETE.
- **The range table** which is a list of relations that are used in the query.
- **The result relation** which is an index into **the range table** that identifies the relation where the results of the query go.
- **The target list** is a list of expressions that define the result of the query. It is expanded into a list of attribute names.
- **The qualification** is a Boolean expression that tells whether the command for the result row should be executed or not.
- **The join tree** which shows the structure of the FROM clause. Nodes in the tree explicitly indicate how the relations will be joined using the join directive keywords: INNER/OUTER, NATURAL, LEFT/RIGHT/FULL.
- The other parts of the query tree representing the group clause, having qualification, DISTINCT clause, SORT clause and set operations. There are also some lists for the subsequent use of optimization.

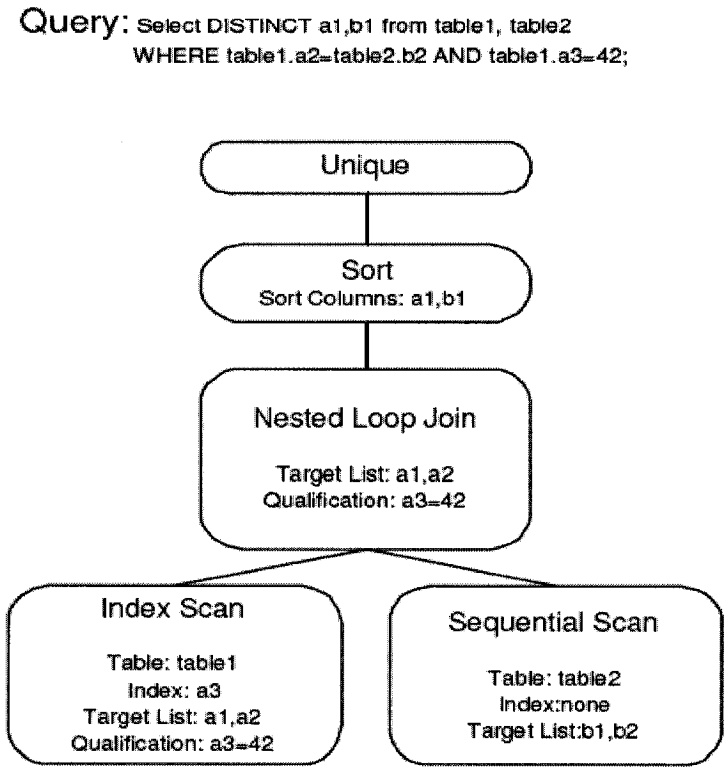

Figure 8: A PostgreSQL Physical Plan [29]

Struct `Plan` Struct `Plan` represents an algorithm tree, which is a binary tree where each of the nodes represents an algorithm. Figure 8 illustrates the Struct `Plan`, which is composed of the following parts [8] [35]:

- **type** – the algorithm type that takes its value from an enum type that includes all the available algorithms, *e.g. HashJoin, MergeJoin, etc.*
- **cost** – the cost is an estimate cost up to and including the execution this node's algorithm.
- **plan_rows** – the number of rows the plan is expected to emit.
- **plan_width** – the average row width in bytes.
- **tartet_list** – the target list, as for a query.

- **qualications** – the qualification, as in a query, but represented as a list of Boolean conditions. The overall expression is the AND of the conditions in the list.
- **left_tree** – the pointer to the left input which is also a plan tree.
- **right_tree** – the pointer to the right input which is also a plan tree.
- Some other entries for run time information and to specify other plans which must be executed before the current node.

2.4.2 Constrained Dynamic Programming

The constrained dynamic programming (hereafter, use dynamic programming in short) search strategy is a kind of exhaustive search strategy that explores all the possibilities in the join space. The PostgreSQL optimizer uses dynamic programming when the number of joins in a query does not exceed a certain threshold (10, normally).

The items in the From-clause of a query are base relations, joins among which will be considered by the dynamic programming strategy. Within these base relations, an explicit join indicating a join directive or a sub-query is treated as a whole (a base relation), of which the join order is fixed and only physical method and inner/outer position are determined by calculation. And, the left-sided and the right-sided trees, which is believed be able to produce the best plan, are considered first and then a subset of bushy trees is taken into account. While joining two base relations, if there is no join specified in the WHERE-clause between the two relations, a Cartesian product is produced. If two trees are algebraically-equivalent, meaning both two trees

contain the same set of relations, the one with the higher cost will be removed. The concept **level** in dynamic programming indicates the number of base relations involved in the join. We here use an example [36] to demonstrate how the dynamic programming strategy works.

Note: In the example below, subscript (b) means bushy tree, and striking through a tree means the tree is removed because it is algebraically-equivalent to some tree but costs more than that tree.

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

FROM-clause: {t1}, {t2}, {t3}, {t4}, {t5}

WHERE-clause: {t1,t2}, {t3,t4} and {t2,t3}

Level 1: {t1}, {t2}, {t3}, {t4}, {t5}

Level 2: {t1,t2}, {t3,t4}, {t2,t3}, {t5,t1}, {t5,t2}, {t5,t3}, {t5,t4}

Level 3: {{t1,t2},t3}, {{t3,t4},t2}, {{t5,t1},t2}, ~~{{t5,t2},t1}~~, {{t5,t2},t3}, {{t5,t3},t4}, ~~{{t5,t3},t2}~~, ~~{{t5,t4},t3}~~

Level 4: {{{t1,t2},t3},t4}, {{{t5,t1},t2},t3}, {{{t1,t2},t5},t4}, {{{t5,t3},t4},t2}, ~~{{t1,t2},{t3, t4}}~~$_{(b)}$

Level5: {{{{t1,t2},t3},t4},t5}, ~~{{{{t5,t1},t2},t3},t4}~~, ~~{{{{t1,t2},t5},t4},t3}~~, ~~{{{{t5,t3},t4},t2},t1}~~, ~~{{{t1,t2},t3},{t5,t4}}~~$_{(b)}$, ~~{{{t3,t4},t2},{t5,t1}}~~$_{(b)}$, ~~{{{t5,t1},t2},{t3,t4}}~~$_{(b)}$, ~~{{{t5,t3},t4}, {t1,t2}}~~$_{(b)}$

Answer: {{{{t1,t2},t3},t4},t5}

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

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

2.4.3 Genetic Algorithm

The Genetic Algorithm (GA) search strategy is adopted for queries with a large number joins. It is a non-exhaustive search method and does not guarantee that the optimal plan will be found. However, it reaches a fairly good solution in a fixed time.

The GA [29] is a heuristic optimization method which operates through determined, randomized search. The set of possible solutions for the optimization problem is considered as a *population* of *individuals*. The degree of adaption of an individual to its environment is specified by its *fitness*. The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*. Through simulation of the evolutionary operations *recombination*, *mutation*,

and *selection* new generations of search points are found that show a higher average fitness than their ancestors.

In the PostgreSQL optimizer, each relation is associated with an integer. A query plan is encoded as an integer string in which the integers represent the relations to join and the order of the integers represent the join order. For example, a query plan of integer string `1-4-3-2' means first join relations `1' and `4', then join relation `3', and finally join relation `2'. To reduce search time, only left-sided trees are considered, so a sole join tree can be constructed from a chromosome [29].

The population of individuals is put into a pool, which is actually an array of chromosomes. At the beginning, the pool size and number of the needed generations are calculated based on the number of relations involved in a query, and then the chromosomes in the pool are initialized randomly. In each repetition, parents are selected from the pool based on a given linear bias. A child is generated by a crossover of its parents. In the crossover, the heuristic of solving the Traveling Salesman Problem (TSP) [26] is adopted. Priority is given to the "shared" edges which refer to the edges between two cities shared by more than one route in the TSP problem. There are many kinds of ways to do crossover. Five variants, edge recombination crossover, partially matched crossover, cycle crossover, position crossover, and order crossover, are supplied in the PostgreSQL optimizer.

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

2.4.4 The Optimization Procedure

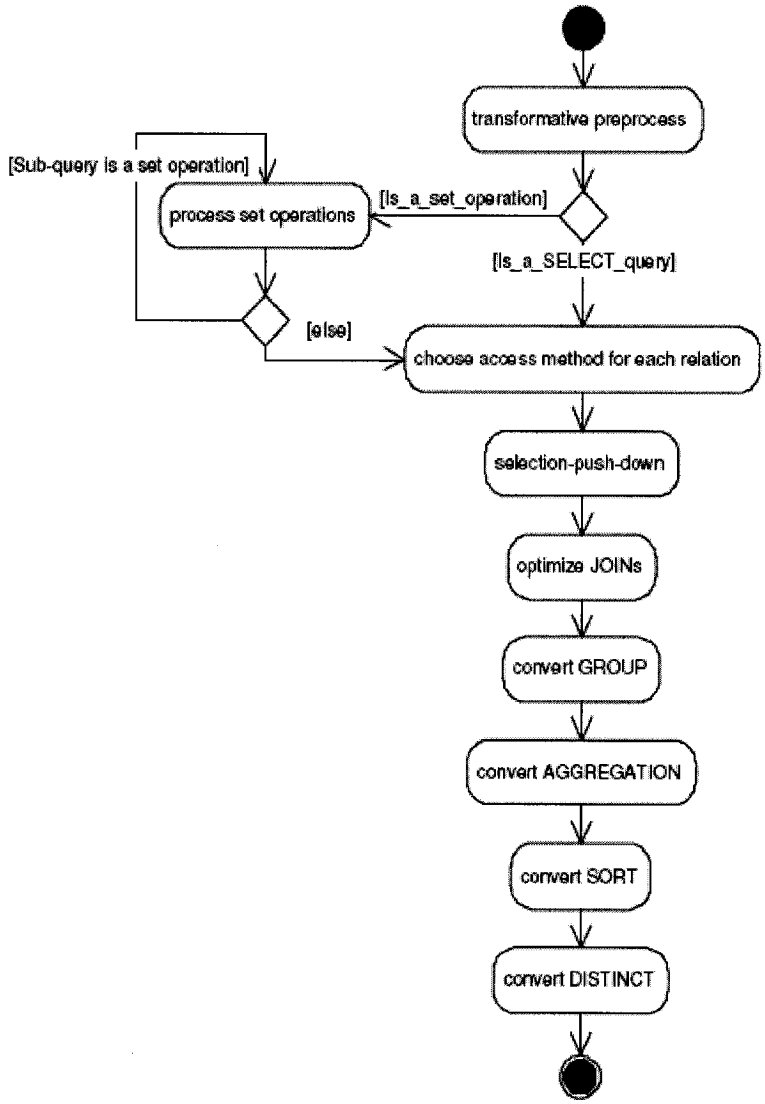

Figure 9: Activity Diagram for PostgreSQL Optimization [29]

Figure 9 demonstrates the overall optimization process of PostgreSQL. A `Query` Struct is populated after parsing a user input query. The `Query` Struct is preprocessed using the following rules before the actual optimization is carried out [29].

- **or-to-union**: A qualified query with or-operator in its where-clause is

converted to union of two queries.

- **constant-expression-simplification**: Reduce any recognizably constant sub-expressions of the given expression tree. Simplify Boolean expressions containing constant sub-expressions if possible.
- **expressions-normalization**: Convert a qualification to the most useful normalized form, either CNF (AND-of-ORs) or DNF (OR-of-ANDs). Push down NOTs.
- **select-push-down**: Push selections down to their corresponding relations as low as possible.
- **sub-query-pull-up**: When a query contains a simple sub-query in its FROM-clause, the sub-query will be pulled up and merged into the upper query. Sub-query-pull-up must be done recursively.

After the rewriting preprocess, the optimizer starts optimizing join operator, which is the most difficult one to optimize among all the relational operators. If the number of relations in the range table of the Query Struct does not exceed a certain threshold, constrained dynamic programming is used to do an exhaustive search in the join space. Otherwise, a genetic algorithm query optimization is adopted [30], because the cost of an exhaustive search will increase exponentially with the number of joins. Finally, the operators GROUP, AGGREGATION, ORDER, and DISTINCT are converted into the corresponding physical nodes in a fixed way.

Chapter 3

Reengineering the Framework

Our work is a third-generation query optimization framework that originally derived from OPT++ proposed by Navin Kabra [20]. Jinmiao Li redesigned Kabra's version, which formed the second-generation query optimization framework, and she customized the framework to a simple bottom-up optimizer [22]. Ju Wang furthered the study on the framework by building an instance of the PostgreSQL optimizer in the second-generation framework [36].

The overall aim of studies with OPT++ sought to determine its fitness as a query optimization sub-framework within the database management system framework. While the two optimizer applications have shown that the major abstractions of OPT++ are well-conceived with a good separation between responsibilities of the three major components and that OPT++ is easy to extend and apply, we found substantial need to improve the design and implementation of the framework at the detailed level.

The purpose of the reengineering is to solve the problems encountered in the implementations of the simple bottom-up optimizer and the PostgreSQL-like

optimizer and to enhance the reusability, extensibility and understandability of the framework. It addresses the performance problem raised in the PostgreSQL-like optimizer. In total, it aims to make the framework really achieve the following flexibilities claimed by Kabra [20]. "First, it should be easy to add new operators as well as new execution algorithms for existing operators. Second, the framework should allow the Optimizer-Implementer (OI) to evaluate various heuristics that can limit the search space explored by the optimizer. The OI should also be able to explore different search strategies, and, if necessary, to mix multiple strategies in a single optimizer. Finally, this flexibility should be achieved without sacrificing performance – *i.e.*, an optimizer built in this extensible framework should not be much worse in its space or time requirements than an equivalent "custom-made" optimizer."

This chapter presents the reengineering work done in the design phase. The implementation details will be found in *Chapter 4*. The initial step of the reengineering activity is to understand the infrastructure of the framework – the three-component architecture, which has been proven to be flexible and extensible for building query optimizers by the previous studies with OPT++ and is not changed in the third-generation version. We then analyze Jinmiao's implementation and list the existing problems. The valuable refinement of OPT++ done in Jinmiao's implementation is presented in her thesis [22] and will not be repeated in this thesis. We focus on the reengineering made on top of Jinmiao's version. Finally, we present our solutions – the actual reengineering – and a small discussion on our solutions.

3.1 System Overview

Before we can go to the actual reengineering done at the detailed level, we need to understand the core of the system - the system architecture and the abstract of the components of the framework.

3.1.1 The Architecture

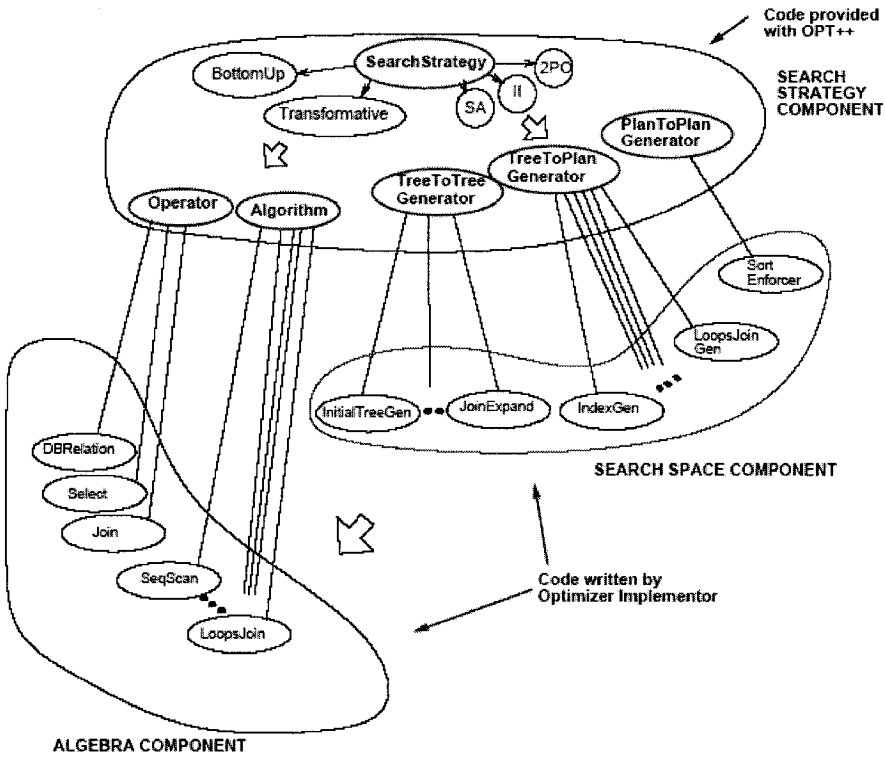

Figure 10: OPT++ Three-Component Architecture [20]

Figure 10 shows an abstract overview of the OPT++ architecture. A query optimizer built in OPT++ framework will consist of three components – a Search Strategy component, a Search Space component and an Algebra component. The framework

itself consists in the Search Strategy component together with abstract classes, namely `SearchStrategy` to interface the Search Strategy component, `Operator` and `Algorithm` to interface to the Algebra component, and `TreeToTreeGenerator`, `TreeToPlanGenerator`, and `PlanToPlanGenerator` to interface to the Search Spaces component. The optimizer developer is meant to write the concrete classes that implement the actual search strategy, e.g. Bottom-up, Transformative, etc., the actual operators & algorithms and the tree & plan generators. The comparatively stable interfaces to the three components contribute to remarkable increase the flexibilities of the framework.

3.1.2 Abstractions of the Three Components

OPT++ is written in C++ and takes the advantage of the object-oriented features, *e.g.* reusability, inheritance, dynamic bounding, *etc.* The three components are conceptualized into some key abstract classes which contain abstract and virtual methods. The abstract methods define the interfaces of some "must-have" behaviors of a class and the implementation of those methods is provided by its subclasses, while the virtual methods implement the general behaviors of a class and do not involve any specific detailed knowledge of the actual local query optimization. An optimizer for a specific database management system can be written by deriving new classes from these abstract classes. Information about the specific query algebra and execution engine for which the optimizer is built, and the search space of execution plans to be explored, are encoded in the virtual methods of these derived classes. The

OI needs to implement the abstract method and override the virtual method when creating a subclass from those abstract super classes for specific requirements [20].

Algebra Component This component maintains the fundamental data structures of the system. It includes the data structures that represent the logical operators and the corresponding physical execution algorithms in the query algebra. It also maintains the tree structures formed by the operators and the associated algorithms. The operator tree represents the whole or part of the query and the algorithms tree represents the execution plan of the whole or part of the query.

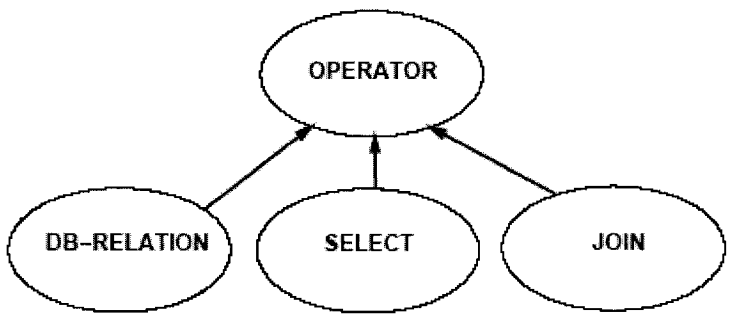

Figure 11: OPT++ Operator Class Hierarchy for a Relational Optimizer [20]

The abstract `Operator` class represents the concept of the operators in the query algebra. Classes to implement specific concrete operators in the actual query algebra, *e.g. Select, Join* in relational database, *etc.*, have to inherit from this class. The inputs of an operator can be database entities (for example, relations for a relational database) that already exist in the database, or they can be the result of the application of other operators [20]. An operator together with its input forms an operator tree, in which the leaf nodes of the tree are database entities and the internal nodes of the tree are other operators being applied to it inputs, which actually are

operator trees too as demonstrated in Figure 4. Basically, a database entity is also being treated as an operator tree of a dummy operator, called `DBRelation`, taking no input. In total, an operator can be applied to one operator tree (unary operator, *e.g. select*), two operator trees (binary operator, *e.g. join*) or no input (dummy operator, *e.g. DB entity*) in the framework, and the result forms another operator tree.

The operator tree is described by class `OperatorTree`, while the detailed information about the tree, such as, set of relations already joined in, predicated applied, does the tree contain interesting information according to certain criteria, and so on, is stored in class `OperatorTreeProperty`. Each node of an operator tree, which actually is a sub-tree that includes the operator and the nodes rooted under it, contains a pointer to an instance of the `OperatorTreeProperty` class. The `OperatorTreeProperty` class includes an `IsEquivalent(OperatorTreeProperty*)` method that determines whether two `OperatorTreeProperty` instances are equal or not. Two `OperatorTreeProperty` instances should be equivalent if the corresponding two operator trees are algebraically equivalent [20].

The abstract `Algorithm` class represents the concept of algorithm that is an access plan of an operator in the query algebra. The concrete algorithms have to be implemented as a sub-class of the abstract `Algorithm` class. Similar to the operator, the algorithm applied to its inputs also forms a tree – an algorithm tree. Correspondingly, an algorithm can be applied to one algorithm tree (unary algorithm, *e.g. filter*), two algorithm trees (binary algorithm, *e.g. HashJoin*) or no input (dummy

algorithm, *e.g. DB entity*) in the framework, and the result forms another algorithm tree. Besides, an `Algorithm` instance associated with an object of the `Cost` class, which defines the methods to calculate the cost of executing the algorithm with the specified inputs. The cost is the main criterion in judging an optimal tree. An algorithm can have an `Enforcer` object, which represents the prerequisites to execute the algorithm. For example, *sort* has to be done in order to execute *merge-join.* Figure 5 demonstrates an algorithm tree.

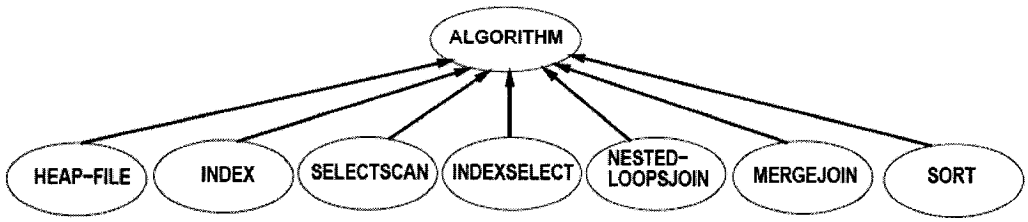

Figure 12: OPT++ Algorithm Class Hierarchy for a Relational Optimizer [20]

Analogous to the operator tree, class `AlgorithmTree` represents an algorithm tree, and class `AlgorithmTreeProperty` stores the detail information about the algorithm tree, which can determine whether two `AlgorithmTreeProperty` instances are equal or not through method `IsEquivalent(AlgorithmTreeProperty*)`. Again, if two algorithm trees are algebraically equivalent, the corresponding two `OperatorTreeProperty` instances should be equivalent.

An operator can be implemented by more than one algorithm, for example, *Join* can be implemented as *HashJoin*, *MergeJoin* or *NestedLoopJoin*. Thus, the `Operator` class also maintains a pointer to a list of `Algorithm` objects. And correspondingly, the `OperatorTree` also includes a pointer to a list of

AlgorithmTree objects.

Search Space Component This component manages the whole search space the optimization is carried out. During the course of query optimization, a query optimizer must generate various operator trees that represent the input query (or parts of it), generate various access plans corresponding to each operator tree and compute/estimate various properties of the operator trees and access plans (for example, cardinality of the output relation, estimated execution cost, etc.) [20]. The best plan will be picked at the end of the optimization. In the framework, those tree generation actions are carried out in the Search Space component. The TreeGenerator abstract class in the Search Space component defines abstract methods that work as interfaces to tree generation actions. The concrete class that implements specific tree generation actions, such as, class JoinTreeExpand that generate an operator tree by joining two operator trees, must be a sub-class of the TreeGenerator abstract class and must implement those abstract methods.

There are three types of tree generators in the framework:

- TreeToTreeGenerator – that includes ExpandTreeGenerator and TransformTreeGenerator. ExpandTreeGenerator generates a new operator tree by applying the specified operator on an operator tree if the operator is a unary operator (*e.g.*, apply operator *Select* on an operator tree) or two operator trees if the operator is a binary operator (*e.g.*, join two trees). TransformTreeGenerator generates an alternative operator tree by applying transforming rules on the operator tree.

- `TreeToPlanGenerator` – that converts an operator tree into a list of algorithm trees correspondingly.
- `PlanToPlanGenerator` – that produce an alternative algorithm tree of the input algorithm tree.

Search Strategy Component The component implements the search heuristic that determines and controls how to carry on the optimization in query-algebra-independent manner [20], *i.e.* the actual operators, algorithms and generators in the system can be modified without modifying the search strategy code. Basically, it explores the search space to generate alternative execution plans for the input query. It stops when no new plans can be generated or are needed to generate and returns the best optimal plan to the optimizer. The Search Strategy contains an abstract class `SearchStrategy` that defines a query-algebra-independent interface to the concept of search strategy. Classes that implement concrete search strategies, such as, bottom-up strategy, dynamic strategy and so on, have to derive from the abstract `SearchStrategy` class, and override the virtual methods or implement the abstract methods to meet specific requirement.

3.2 Review the Previous Optimizer Framework

In this section, we review the previous (second-generation) query optimization framework. We present the overall structure of the framework and analyze the components and their collaborations in the framework. And finally we discuss the problems and limitations of the framework.

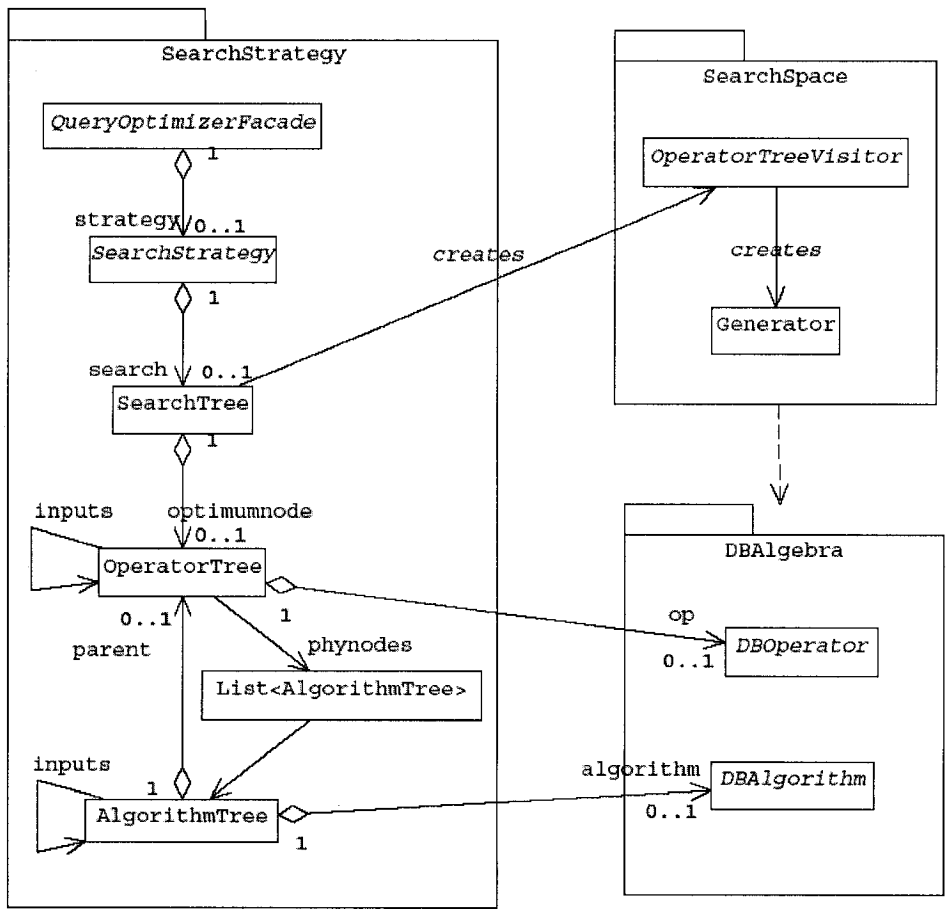

Figure 13: System Overall Diagram of the Second-Generation Framework [22]

The three components and their main objects are shown in Figure 13. The Search Strategy component encapsulates the `QueryOptimizationFacade` class which provides the system entry and controls the optimization flow. This component also includes the `OperatorTree` class and the `AlgorithmTree` class, which are aggregated of class `Operator` or `Algorithm` respectively. The `SearchTree` class is the core class of the framework. It is a container that maintains the operator trees generated during the optimization and is also in charge of implementing the search logic delegated from the `SearchStrategy` class. New operator trees are generated by creating instances of the `OperatorTreeVisitor` class in the Search

Space component. The `OperatorTreeVisitor` instance in turn creates `Generator` instance to generate new `OperatorTree` or `AlgorithmTree` objects defined in the Search Strategy component.

3.2.1 The Strategy Components

The major classes in the Search Strategy component include: the `QueryOptimizerFacade` class, the `SearchStrategy` class, the `SearchTree` class, the `OperatorTree` class, the `OperatorTreeProperty` class, the `AlgorithmTree` class and the `AlgorithmTreeProperty` class. It also encapsulates the `Cost` class that implements the cost estimation of the algorithms in the query algebra.

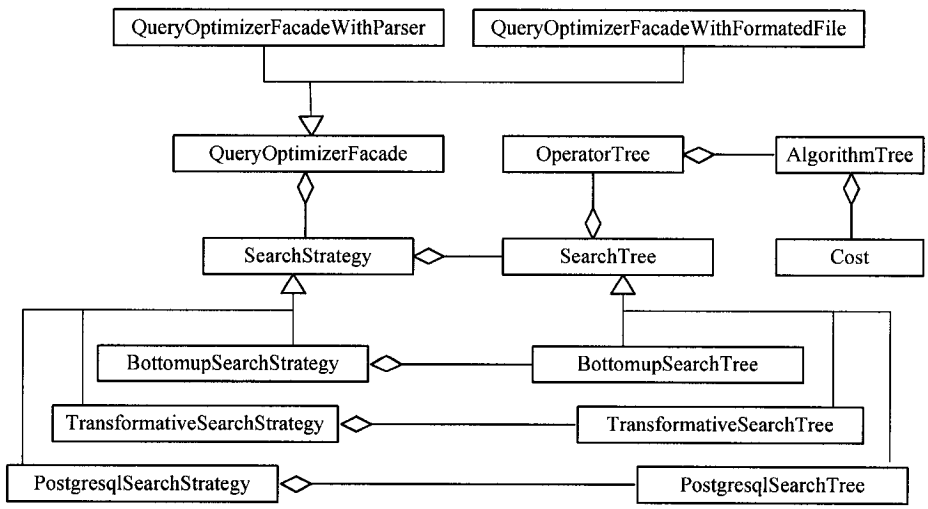

Figure 14: Search Strategy Component of the Second-Generation Framework

`QueryOptimizerFacade` Class that employs the Façade Design Pattern, which provides the entry of the system. It initializes the global variables defined in class `Aglob_vars_t`, *i.e.* `query` – the query to be optimized, `cat` – the catalog of

the system, `oopt` – the optimization option, `hashtable` – the hash table used for suboptimal pruning, `strategy` – the instance of `SearchStrategy` subclass that will be used to do the optimization. It then calls `strategy->DoOptimize()` method to complete the whole job.

`SearchStrategy` Abstract class that defines interfaces for all search strategy approaches that are used in query optimization. The class consists of a `SearchTree` object that is used to perform the actual optimal plan searching. Three concrete `SearchStrategy` classes are implemented: the `BottomupSearchStrategy` class, the `TransformativeSearchStrategy` class and the `PostgreSQLSearchStrategy` class. They differ in using different type of `SearchTree` objects to perform the search process. The `SearchTree` object initialized by the `SearchStrategy` class is also put in the global variables.

`SearchTree` Abstract class maintains the `OperatorTree` objects generated during the course of the optimization. Corresponding to the `SearchStrategy` hierarchy, there are three concrete `SearchTree` classes: the `BottomupSearchTree` class, the `TransformativeSearchTree` class and the `PostgreSQLSearchTree` class. Those concrete classes implement the `DoSearch()` method, which implements the search algorithm of the corresponding concrete `SearchStrategy` class. The `Prune()` method in the `SearchTree` class implements the suboptimal pruning mechanism. In sum, the `SearchTree` class contains both the optimization data and the optimization strategy implementation.

`OperatorTree` Data structure that represents an operator tree. It contains a

pointer pointing to an `Operator` object that roots the tree, a pointer to an `OperatorTreeProperty` object that stores the detailed information of the tree and a list of `AlgorithmTree` objects that represent the physical execution plans of the tree. These objects are initialized when the operator tree is constructed.

`AlgorithmTree` Data structure that represents an algorithm tree. It contains a pointer pointing to an `Algorithm` object that roots the tree, a pointer to an `AlgorithmTreeProperty` object that stores the detailed information of the tree, a pointer to the operator tree with which it is associated and a `Cost` object.

3.2.2 The Search Space Components

The Visitor Design Pattern is applied in the framework to associate the data structure defined in the Algebra component and the operations on the data structure (tree generations) implemented by the Visitors in this component, which increases the extensibility of the framework and provides a very organized separation of program codes. The Visitors are implemented by a set of `OperatorTreeVisitor` classes.

The search space is organized by two sets of classes: the `OperatorTreeVisitor` classes and the `Generator` classes. In order to get a clearer layout, the visitors do not contain the actual implementation of the tree generations, while they call a set of generators to actually implement the operations respectively. In other words, the `OperatorTreeVisitor` classes define the interfaces to the tree generations, and the implementations of these interfaces are hided in the `Generator` classes. One method in the visitor class defines one type of

tree generation, *e.g.* `VisitSelect()` generates a new tree by applying the *Select* operator on an operator tree. Correspondingly, one generator class implements one type of tree generation, *e.g.* `ExpandSelect` class implements the tree generation defined by method `VisitSelect()`. The association between the two sets of classes is: a method defined in an `OperatorTreeVisitor` class creates an instance of the `Generator` class that implements the method and uses the `Generator` instance to complete the tree generation.

The reason why the interface is defined as a method in an `OperatorTreeVisitor` class while the implementation is defined as `Generator` class is aiming for a clear code separation since the definition of an interface only takes several lines of codes while the implementation of the interface might need hundreds of lines of codes.

Figure 15 shows the two hierarchies – the `OperatorTreeVisitor` class hierarchy and the `Generator` class hierarchy. The root of the visitor hierarchy is the abstract class – `OperatorTreeVisitor`, which has three types of subclass: the `ExpandTreeVisitor`, the `TransformTreeVisitor`, and the `TreeToPlanVisitor`. There is no abstract root class for the generator hierarchy, while the generators also fall in three types corresponding to the visitors: the `ExpandTreeGenerator`, the `TransformTreeGenerator`, and the `AlgorithmTreeGenerator`.

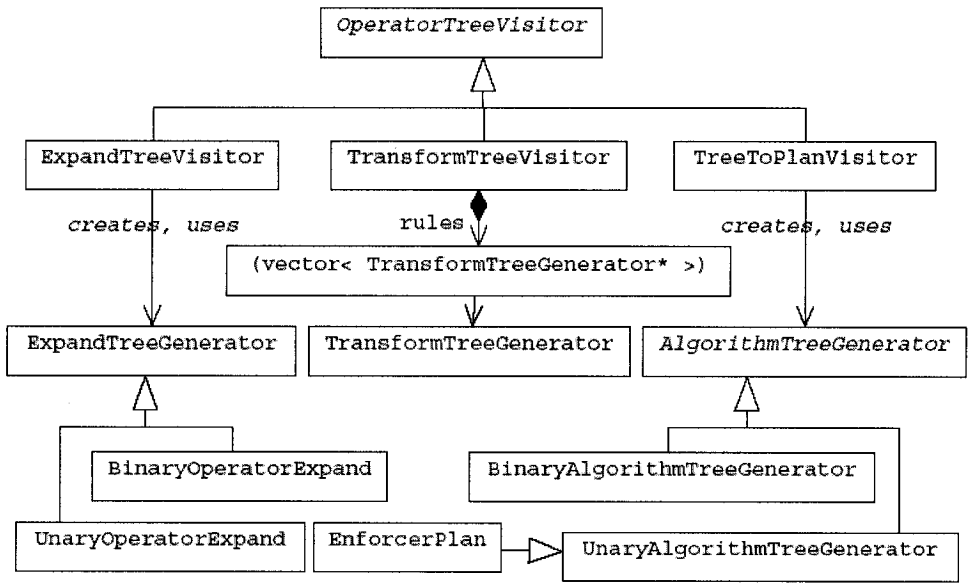

Figure 15: The Visitor Hierarchies in the Second-Generation Framework [22]

To create a tree generator to generate an operator tree or an algorithm tree, the user needs to initialize a tree visitor object and calls the `accept()` method of the `Operator` object which would become the root of the operator tree for tree expanding or which is already the root operator of the operator tree for tree transforming or tree-to-plan converting. In the `ExpandTreeGenerator` class, the visit-methods (*e.g.* `VisitDBRelation`, `VisitSelect`, *etc.*) can distinguish which `ExpandTreeGenerator` subclass they should create and directly create it according to the input `Operator` object, while they do not know which `TransformTreeGenerator` subclass or which `TreeToPlanGenerator` subclass they should create with the input `Operator` object. Method `VisitDBOperator` is used to redirect these visit-methods to create a corresponding `TransformTreeGenerator` subclass or `TreeToPlanGenerator` subclass. More details on the associations between the

visitor methods and the generator classes are given in *3.2.4*.

3.2.3 The Algebra Components

The Algebra component is relatively simple. It is composed of the `Operator` abstract class, the `Algorithm` abstract class and their subclasses that implement the concrete logical operators and their associated physical execution algorithms respectively in the query algebra.

Class `OperatorAndAlgorithm` provides an interface to the Algebra component. It defines objects of the `Operator` subclasses and the `Algorithm` subclasses that implement the concrete operators and algorithms.

The Visitor Design Pattern employed between the Algebra component and the Search Space component physically separates the algebra data structure and the algorithm implementation and allows the Search Space to be experimented with different implementations without affecting the Algebra component.

3.2.4 Problems Encountered

The previous query optimization framework implementation raises the following issues:

Issue 1: The PostgreSQL-like optimizer customized from the framework has raised many performance issues. The testing figure presented in Ju Wang's thesis [36] shows a big gap of performance in terms of the time needed to optimize queries between the PostgreSQL-like optimizer and the native PostgreSQL optimizer. For

constrained dynamic programming search strategy, the application takes 9.4 times the time used by the native PostgreSQL optimizer when the number of joins in a query reaches nine. And for genetic algorithm search strategy, the application is 6.6 times slower than the native to optimize a query with number of joins of 20.

Issue 2: Even though the goal of the framework is to build optimizers that organize the three components to process the query optimization, no concept of optimizer object is defined. Instead, the three components are bound together via saving the optimization context as global variables as shown in Figure 16. In the program codes, the global variables are stored in a class called `Aglob_vars_t`, and they are accessed everywhere in the whole program by hard-coding an inline function that returns a pointer to the `Aglob_vars_t` class, for example, `GlobalVariable()->query` refers to the query object being optimized. These global variables include:

A `Query` object – represents the current query to be optimized.

An `OptimizationOption` object – contains the optimization options, *e.g.* left_deep_ join, do_merge_join, *etc.*

A `Catalog` object – contains the catalog information of the system.

A `Hashtable` object – maintains hash codes of the operator trees generated for suboptimimal pruning.

A `SearchTree` object – contains the operator trees generated during the optimization.

A `SearchStrategy` object – represents the search strategy currently used.

An `OperatorAndAlgorithm` object – defines a list of operators and algorithms available in the query algebra.

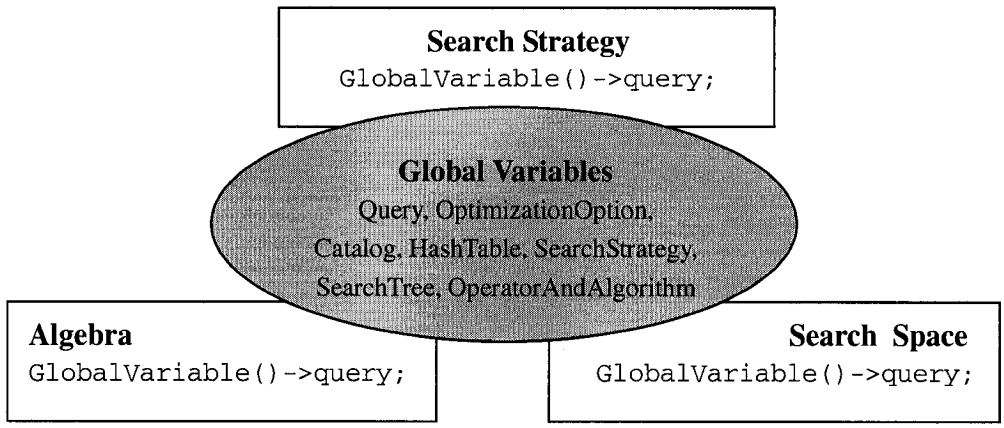

Figure 16: Global Variables Tightly Bind the Three Components

The use of hard-coding accesses to the global variables limits reusability and extensibility of the framework. First of all, the information hiding and encapsulation are violated. Secondly, the three components are tightly bound together. Thirdly, no more than one optimization process can exist at the same time, which makes it very difficult to deal with situations where the whole optimization process is needed to be divided into several sub-optimization processes with different contexts, *e.g.* different search strategies, different optimization options, *etc.* For example, it is hard to extend the framework to deal with queries containing sub-queries using recursive way as done in the native PostgreSQL optimizer.

Moreover, the template `Set` class that implements a bitmap tightly fixed with the `GlobalVariable()->query` object, which determines the size of the bitmap. For example, the constructor of the Set class is defined as:

```
Aset_t(void):Abitmap_t (SetElementType::TotalNumber()){}
```

There are three types of set element defined in the framework: the Aptree_t class, the Arel_t class and the Aattr_t class, which mainly comprise the Query data structure. The Aptree_t class represents an operation in a query, the Arel_t class represents a relation in a query and the Aattr_t class represents an attribute of a relation in a query. The TotalNumber() method implemented in the Aptree_t class, for instance, is as follows:

```
return GlobalVariable()->query->numoperations();
```

The same thing applies for retrieving an element in the Set, which needs to call the SetElementType::NthNumber(int N). And the NthNumber(int N) method in the Aptree_t class, for instance, is implemented as follows:

```
return GlobalVariable()->query->operation (N);
```

With this mechanism, no more than one Set that represent data in different queries can exist at the same time since the GlobalVariable()->query can only point to one Query object.

Issue 3: The SearchStrategy class does not implement the search heuristics but delegates it to the SearchTree class, which also performs as a container holding the OperatorTree instances generated during the optimization. Thus, different search strategies have to implement different search trees even though they can use the same container structure. That way produces duplicate codes for generic data structure. Furthermore, because the OperatorTree instances contained in the SearchTree class is accessed very frequently by the tree generators defined in the Search Space component, mixing the search logic implementation and the data

structure basically increase the coupling between the Search Strategy component and the Search Space component in the framework.

Issue 4: The Search Strategy component heavily couples with the Algebra component. The operator tree composed of operators is naturally adhering with the operators. In other words, the `OperatorTree` class and the `OperatorTreeProperty` defined in the Search Strategy component unavoidably cohere with the `Operator` class defined in the Algebra component. The same thing applies to the relation between the `Algorithm` class in the Algebra component and the `AlgorithmTree` & the `AlgorithmTreeProperty` classes in the Search Strategy component. Moreover, some operator tree-specific behaviors, such as retrieving the best algorithm tree from the list of algorithm trees of an operator tree and deleting the list of algorithm trees, are implemented in the `SearchTree` class in which the implementation of the search logic is in the previous framework.

Issue 5: Basically, the concept of the Search Space component is not completely put into effect in the implementation. No clear interfaces are provided for accessing the search space. And some tree generations, *e.g.* joining two trees, are not implemented by the `Generator` subclasses in the Search Space component. But they are implemented in the Search Strategy component, which causes the classes defined in the Search Strategy component directly access detailed information defined in the Algebra component. That way thwarts achieving one of the aims of the framework – modifying the query algebra definitions without affecting the search strategy implementation.

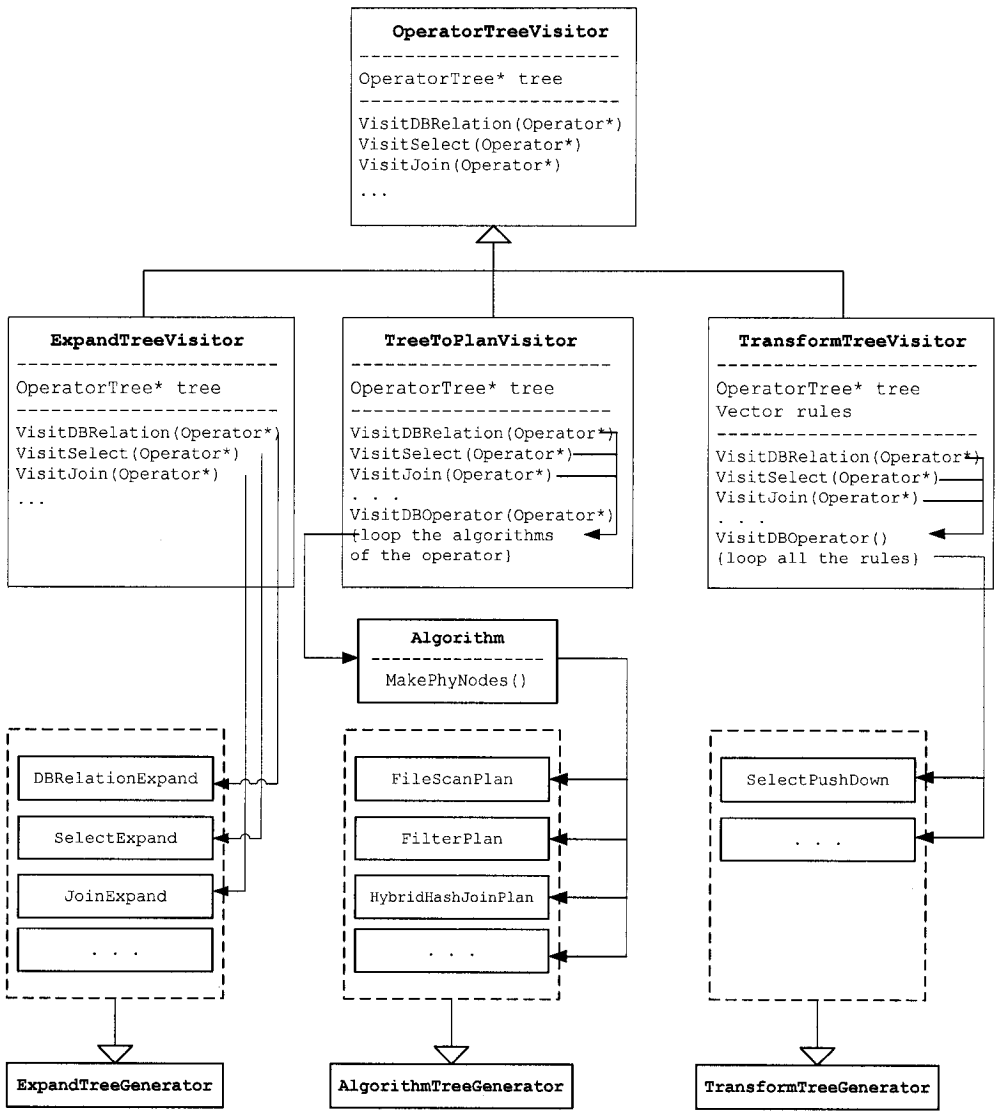

Figure 17: The Original Visitor & Generator Structure

Issue 6: The tree generations are implemented by two sets of classes, the tree visitor classes and the tree generator classes, in the Search Space component. The tree visitor classes provide interfaces to the tree generations while the detailed implementation of the tree generations are hidden in the tree generator classes. Figure 17 is a layout of the two sets of classes. There are two problems: the implementation of the visitor class hierarchy is unclear and the associations between the visitor methods and the generator classes are inconsistent.

There are three types of tree visitor classes defined in the framework: the `ExpandTreeVisitor` class defines interfaces to expanding an operator tree; the `TreeToPlanVisitor` class defines interfaces to converting an operator tree to a list of algorithm trees; the `TransformTreeVisitor` class defines interfaces to transforming an operator tree to an alternative. As shown in Figure 17, the same methods defined in the `ExpandTreeVisitor` class are also defined in the `TransformTreeVisitor` class and the `TreeToPlanVisitor` class, which yet have different behaviors from the `ExpandTreeVisitor` class. For example, method `VisitSelect(Select* op)` defines the behavior of applying the `Select` operator on an `OperatorTree` object to produce a new `OperatorTree` object in the `ExpandTreeVisitor` class, while this method means nothing in the `TransformTreeVisitor` class or the `TreeToPlanVisitor` class.

The unclearness of the definitions of the tree visitor classes causes the associations between the tree visitors and the tree generators inconsistent. In the `ExpandTreeVisitor` class, each VisitXXX(Operator* op) method directly associates with a tree generator class. However, in the `TreeToPlanVisitor` class, these VisitXXX(Operator* op) methods, *e.g.* `VisitSelect(Select* op)`, `VisitIndexCollapse(IndexCollapse* op)`, do nothing but just forward the input operator to another method, marked as *`VisitDBOperator(Operator*op)`*. That method then loops over all the associated `Algorithm` objects of the given `Operator` object, and for each of them, calls the `Algorithm` object to create a corresponding `Generator` object to

generate an algorithm tree, instead of creating the `Generator` object via the VisitXXX(Operator* op) methods as done in the `ExpandTreeVisitor` class. The situation becomes even stranger for transforming a tree. Again, all the `VisitXXX(Operator*op)` methods, *e.g.* `VisitSelect(Select* op)`, `VisitIndexCollapse(IndexCollapse* op)`, do nothing but forward to another method, called `VisitDBOperator()`, and that method does not have the `Operator` object passed as a parameter since it does not need that `Operator` object. That method iterates all the defined transforming rules, and applies each of them on the operator tree.

3.3 The Reengineering

This section illustrates the reengineering done on the previous framework. We first summarize the points how we improved the framework. We then detail the improvements for each component.

3.3.1 Reengineering Summary

The reengineering sticks to the abstractions of the three components as discussed in the first section of this chapter. Corresponding to the issues we analyzed in the previous section, we made the following reengineering:

1. Profile the PostgreSQL-like optimizer and improve the performance of the framework. We analyze the PostgreSQL-like optimizer application and try a lot of valuable rules that suggested by Scott Meyers [25]. Among all the attempts we

tried, three of them contribute the most to the performance improvement. Performance testing result of the new PostgreSQL-like optimizer compared to the original version and the native version is reported in Table 7.

- Change the attribute **`_operation`** that represents the algebraic expressions of a user-input query from a linked-list of `Aptree_t` objects to array of `Aptree_t` pointers in the `Query` class. According the profiling report generated by the g++ compiler, we found the attribute **`_operation`** of the `Query` class is the most accessed container. We got a big improvement of the performance (approximately 85%) after changing this attribute to an array of `Aptree_t` pointers.
- Eliminate the number of `AlgorithmTree` objects generated while converting an `OperatorTree` object to a list of `AlgorithmTree` objects by calculating the cost of an algorithm tree before really creating a new algorithm tree object. Do not create the algorithm tree if it is not interesting and cost more than other algorithm trees in the list of `AlgorithmTree` objects of the `OperatorTree` object.
- Avoid reinitializing `Set` (it actually is a `bitmap`) objects because the initialization of a `Set` object is expensive and it causes a loop over all the elements in the set. Change the attributes of `Set` objects into `Set` pointers where the size of the `Set` can not be determined in the constructor of the class and the attributes of `Set` objects are needed to be reinitialized afterwards.

2. Abstract a class template called `QueryOptimizer` to coordinate the three components to complete the query optimization process. And localize the global variables that were referred using C++ extern inline function to be data members of the `QueryOptimizer` class. That way easily solves the sub-query optimization problem. The `QueryOptimizer` class also performs as an interface to the optimizer built in the framework.

   Redesign the `Set` class to relieve it from relying on the `Query` object pointed by the `GlobalVariable()` method. The `Set` class has a pointer pointing to the target `Query` object, which has to be initialized while constructing a `Set`. The `TotalNumber()` method and the `NthNumber(int N)` method in the set element classes are modified to take the `Query` object from a parameter of the method. For example, in the `Aptree_t` class, the two methods would be:

```
TotalNumber(Query* q){return q->numoperations ();}

NthNumber(int N, Query* q){return query->operation (N);}
```

   Here is the definition of the `Set` constructor:

```
Aset_t (Query * q) : Abitmap_t (SetElementType::TotalNumber (q))

{this->mydomain=q;}// mydomain is Query pointer
```

3. Reorganize the divisions of functionalities between the `SearchStrategy` class and the `SearchTree` class. We reassign the functionalities according to the natural behaviors of each component. Firstly, we move the implementation of search heuristics to the `SearchStrategy` class from the `SearchTree` class and make the `SearchTree` class a generic operator tree container that provides

GET/SET methods for accessing its elements. The separation of the logic and the data structure makes a more understandable program and a clearer interface between the Search Strategy component and the Search Space component since both components operate on the elements (the operator tree) stored in the `SearchTree` class.

4. Decouple Search Strategy Component and Algebra Component to allow modifying the Algebra without leaving impact on the Search Strategy and the other way around. There is a rule to follow, the search strategy always take an operator tree as a whole. It does not tangle the detail of the operator tree structure and its corresponding algorithm trees. Otherwise, any changes in the `OperatorTree` class or other related classes e.g. the `OperatorTreeProperty` class, the `Operator` class, *etc.*, might cause change requests in the `SearchStrategy` class. Making operator/algorithm information related decision directly in the `SearchStrategy` class results in a higher performance but a less extensibility. So, there is a performance and extensibility trade-off to make in this case. Our rule is to make the not-must-in-strategy decision out of the Search Strategy component, and get the information from the Algebra component via GET/SET method for must-in-strategy decision and add the GET/SET methods to the super class as high as possible.
5. Provide a clear interface between Search Strategy and Search Space. Abstract an interface for the Search Space component. This interface provides a unified

manner for the search strategy to access the search space. Therefore, the modifications made in the implementations of tree generation will not affect anything in the Search Strategy component.

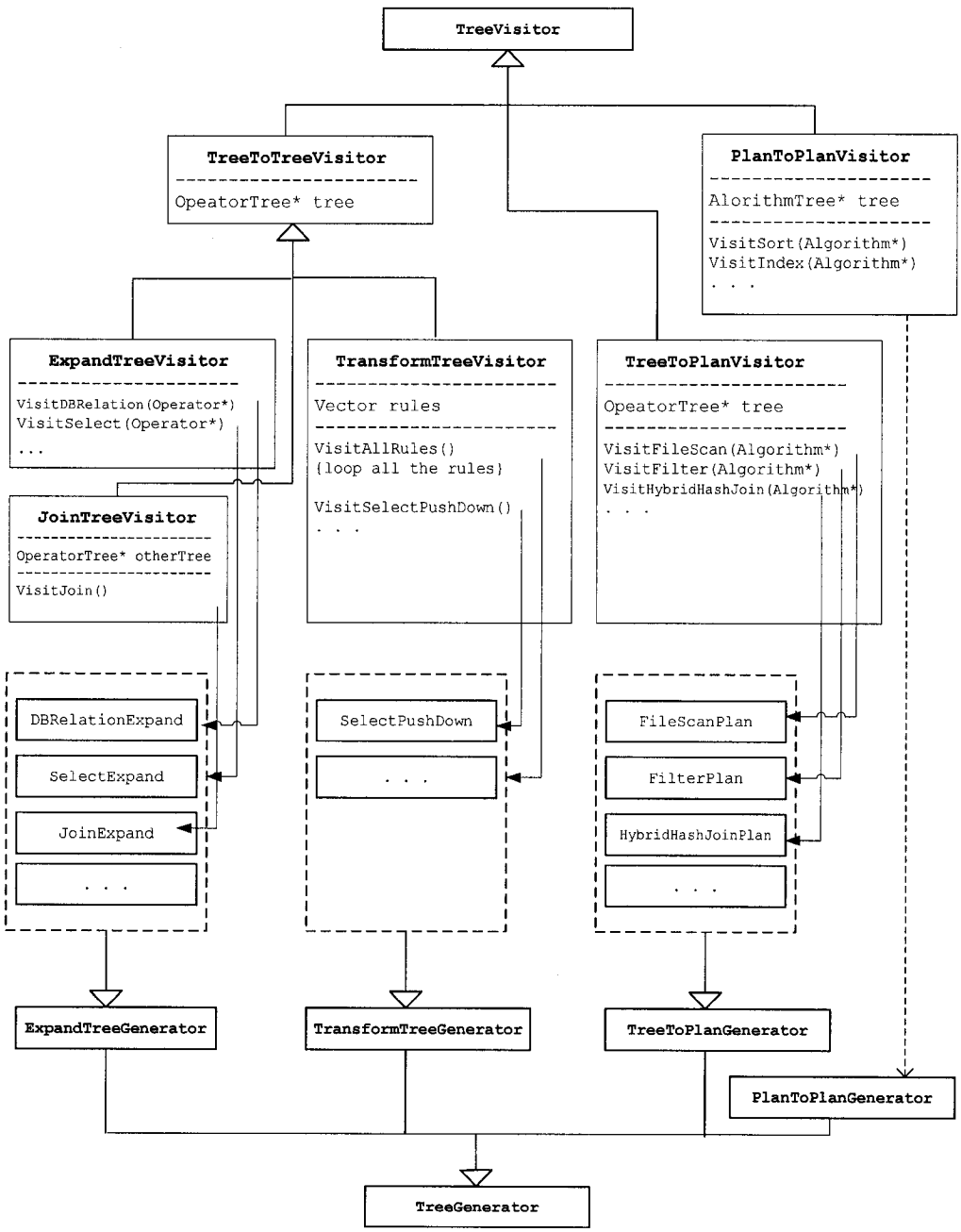

Figure 18: The Improved Visitor & Generator Structure

6. Restructure the visitor and generator hierarchies and refine implementation of the associations between them in the Search Space component. Abstract a new `Treevisitor` subclass called `JoinTreeVisitor` that replaces the `VisitJoin()` method to deal with the special features of joining two trees while expanding the tree. The `JoinTreeVisitor` calls the `JoinExpand` generator class, a subclass of `BinaryOperatorExpand`, to produce a new tree by joining two trees.

Figure 18 demonstrates the new visitor and generator structure. The `PlanToPlanVisitor` that is not abstracted in the previous version defines interfaces for plan to plan generations, which mainly happen when the operation represented an `Enforcer` is needed to be performed on an algorithm tree before certain algorithm can be applied to that algorithm tree, for example, two algorithm trees have to accomplish `Enforcer Sort` before they can do a merge-join. The methods defined in the `PlanToPlanVisitor` class create corresponding `PlanToPlanGenerator` objects, which is not detailed in Figure 18 for space limitation.

3.3.2 The New Framework

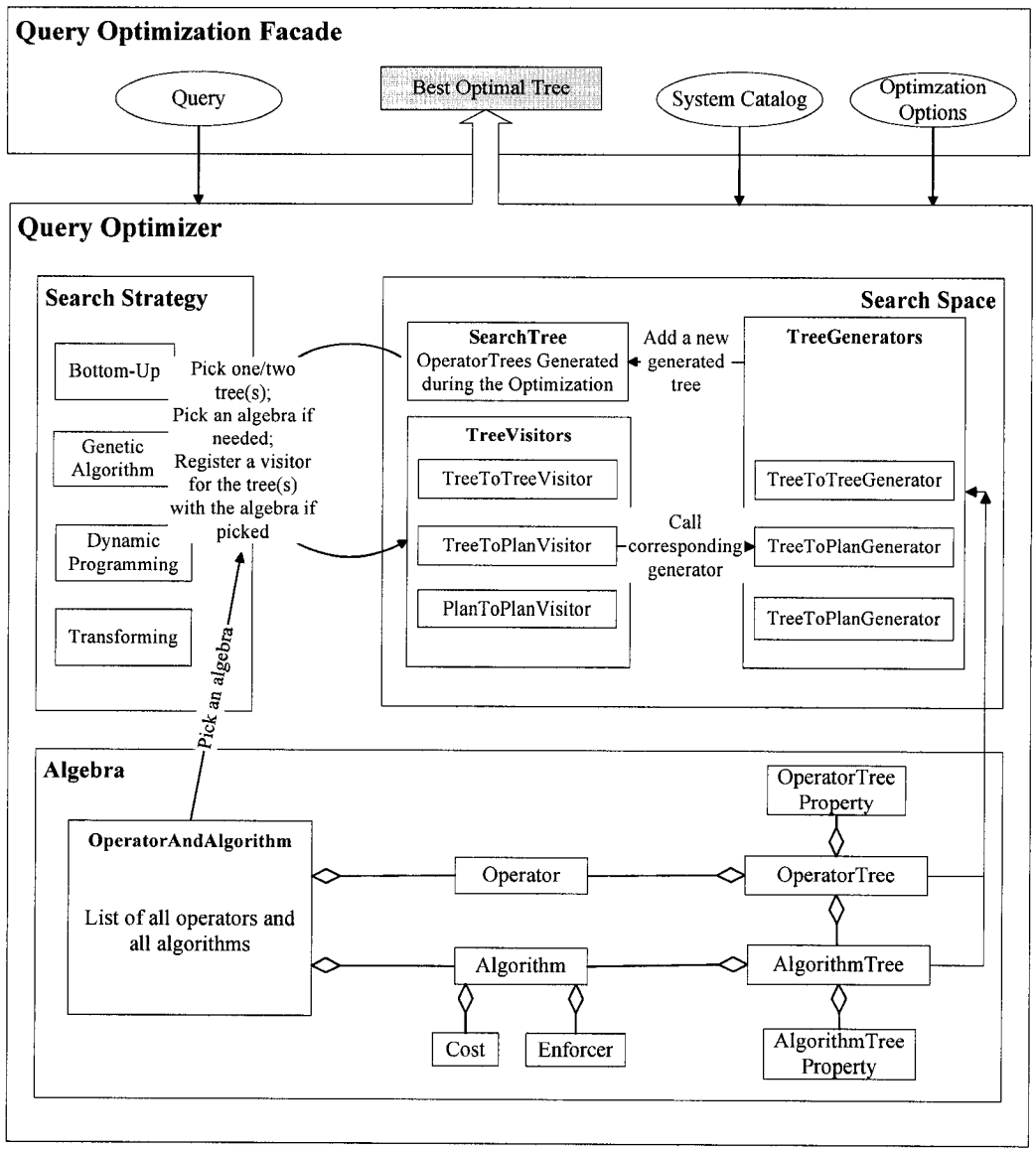

Figure 19: System Overall Diagram of the New Framework

In the third-generation framework, to optimize a query, the `QueryOptimizerFacade` takes the algebraic expression of a user query as the input. It will create an `QueryOptimizer` object and calls the method `Optimize()` of the object to complete the optimization, which will return an optimal physical execution plan. Figure 19 demonstrates the overalls of an optimizer system built in

our framework, which is able to easily experiment different search strategy and easily extend the search space, *e.g.* adding new algebraic operators and its associate algebraic algorithm or transformation rules.

The `QueryOptimizer` class offers a straightforward manner to utilize the flexibility and extensibility of the three-component architecture. It initializes concrete objects of the interfaces to the three components and then calls the `DoOptimize()` method of the Search Strategy object to complete the query optimization. The class definition is as follows. Detailed description of the class is given in *4.1.2*.

```
template <class SearchStrategyType, class SearchSpaceType,
          class AlgebraType >
class QueryOptimizer
{...}
```

As showed in Figure 19, the query optimization of an optimizer system built in the new framework involves the following steps:

→ The *Query Optimization Façade* receives a *Query* object that normally comes from a query parser. It constructs the *System Catalog* object (often by calling a catalog reader) and *Optimization Options* object that defines the optimization flags, *e.g. left_deep_only, do_exhaustive_selects etc.* It then initializes a *Query Optimizer* and passes the above three objects to it.

→ The *Query Optimizer* takes a *Query* to be optimized, the *System Catalog* and the *Optimization Options* as input. It then initializes interfaces to the three components. It calls the *Search Strategy* to complete the optimization, and returns the complete optimal tree to *Query Optimization Façade.*

→ The *Search Strategy* picks one tree (if a unary operator is applied to expand the tree, *e.g. order by*, or transforming rule is used to transform a tree) or two trees (if an binary operator is applied to join two trees, *e.g. join*) from *Search Tree* in the *Search Space*. It picks an algebraic operator if it wants to expand the tree(s) picked from the *DB Algebra*. It then registers a corresponding *Tree Visitor, e.g. TreeToTreeVisitor etc.,* for the tree(s) picked along with the algebra if picked.

→ The *Tree Visitor* in the *Search Space* calls corresponding *Tree Generator* to generate a new tree, which will be put in the *Search Tree.*

→ The above two steps are repeated until the *Search Strategy* stops.

→ The *Search Strategy* retrieves the best optimal tree from the *Search Tree* and returns it to the *Query Optimizer*, which in turn is returned to the *Query Optimization Façade.*

In the new framework, more than one search strategy can be applied in a single optimizer, and they can access the same search space or different search spaces. Furthermore, a search strategy can be a simple search strategy or a compound search strategy, which has sub search strategies. For example, to optimize a compound query: `select foo.name from Cities as c, (select name from Employees e,Persons p where p.name=e.name order by name) as foo where foo.name=c.name`, which can be decomposed into two simple queries: query1 - `select foo.name from Cities as c, foo where foo.name=c.name` and query2 - `select name from Employees e,Persons p where p.name=e.name order by name`. We can apply bottom-up strategy for query1

and dynamic programming for the query2.

The Search Space Component in the new framework manages the whole search space the optimization is carried out. Specifically, the Search Space Component is responsible for holding the operator trees generated during the optimization and offering tree generators to generate new trees. The new abstracted `SearchTree` class is moved to here from the Search Strategy component that holds the operator trees generated during the optimization. It provides GET/SET methods for accessing the operator trees held. And the `SearchSpace` class provides an interface for accessing the `SearchTree` object and creating instances of `TreeVisitor` class. It also implements methods for pruning suboptimal algorithm trees generated to narrow the search space explored by the search strategy.

The Algebra component in the new framework mainly maintains the static data structure of the system, which is a relatively stable component. The `OperatorTree` class, `OperatorTreeProperty` class, `AlgorithmTree` class and the `AlgorithmTreeProperty` class which naturally adhere to the `Operator` class and the `Algorithm` class are moved to the Algebra component from the Search Strategy component, while the `Clone()` method defined in the `Operator` class that is used by the `TreeGenerator` object to fetch the information related to the `Operator` object from a user-input query during the tree generation, is moved to the corresponding `TreeGenerator` class. And the Cost class whose behaviors (the implementation of calculations) tightly associate with the `Algorithm` objects is also shifted to the Algebra component from the Search Strategy component.

3.3.3 Discussion

We can see the following flexibilities among the three main components in the above implementation.

By designing the optimizer as a class template of the three types of classes, which offer interfaces to access the functions of the three components of the framework respectively, it becomes very easy to instantiate an optimizer for different search strategy, search space and DB algebra. We can evaluate different search strategies on the same query in certain search space to retrieve the best physical plan by simply creating optimizers with different types of search strategies.

Search Strategy and *Algebra* are independent of each other. *Search Strategy* only needs to know what algebraic operators and algorithms are available to use for the optimization in the system. It does not need to tangle the detailed data structure of the operator/algorithm tree. Therefore, different *Search Strategy* can easily use the same *Algebra* and the other way around. Actually, the interaction between the search logic and the data structure has been transferred to the *Search Space*.

Furthermore, the *Search Space* offers a very clear interface for the *Search Strategy*, which allows that different *Search Strategies* can easily adapt the same *Search Space,* and modification made in the *Search Space* will not affect the Search Strategy. The `SearchTree` class that resides in the *Search Space* performs as a collection holding the initial operator trees and the operator trees generated in the course of the optimization. It offers GET/SET methods for accessing its elements (the operator trees). *Search Space* offers methods which create corresponding

`TreeVisitor` objects to response the requests a search strategy would ask, *e.g.* to expand a tree with some operator, to transform a tree with some transforming rule, or to convert a logical tree to a physical plan, *etc.*

Also, the collaborations of `TreeVisitor` classes and `TreeGenerator` classes inside the *Search Space* are clear and straightforward. Visitor Design Pattern is used between the tree structures and the tree generators, which gains the extensibility of adding new operator/algorithm in the *Algebra* and its corresponding implementation in the *Search Space*.

Chapter 4

Implementation

In this chapter, we describe the implementation of the third-generation query optimization framework. We only emphasize on the new aspects of this generation and do not detail previous implementations. The PostgreSQL-like optimizer application is integrated to the new framework, and a performance testing on it is performed. We report the performance testing result of the PostgreSQL-like optimizer application in this chapter.

4.1 The Search Strategy Component

The Search Strategy component implements the search logics, *e.g.* bottom-up, dynamic, *etc.* that guide the query optimization. The entry to a query optimizer built in the framework is enclosed in this component. As shown in Figure 20, this component mainly maintains three classes: the `QueryOptimizationFacade` class, the `QueryOptimizer` class and the `SearchStrategy` class. Detailed descriptions of these classes are given as follows.

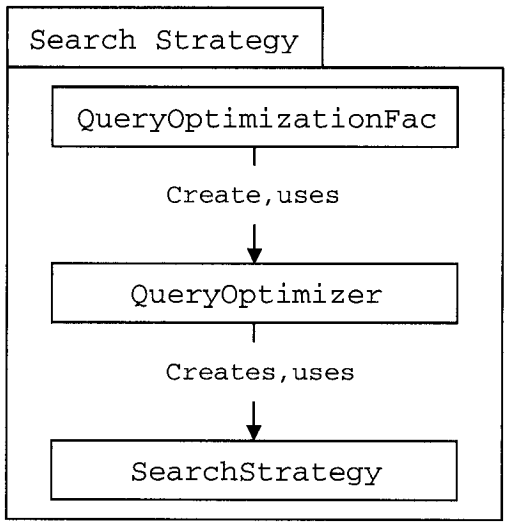

Figure 20: Search Strategy Component in the New Framework

4.1.1 The QueryOptimizationFacade Class

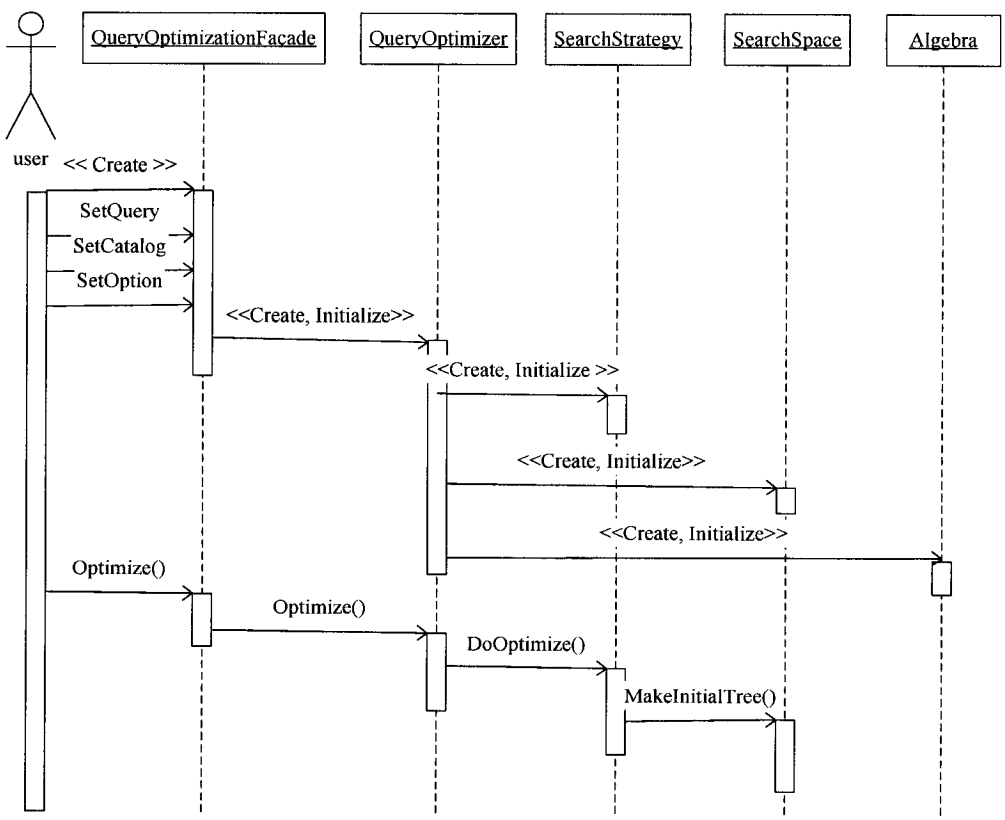

Figure 21: Sequence Diagram of Query Optimization Initialization

The `QueryOptimizationFacade` class employs Facade Design Pattern and performs as an interface to the query optimization process in an optimizer built in the framework. The `Optimize()` method of this class controls the whole optimization flow and does the whole work.

As shown in Figure 21, the user initializes a `QueryOptimizationFacade` object and sets the query, system catalog and optimization options to the facade object. Instead of binding the whole system with a set of global variables, *e.g.* the query to optimize, the optimization options and the hash table for pruning, *etc.*, and initializing an instance of the `SearchStrategy` class to process the query optimization as done in the previous version, the facade object creates an instance of the `QueryOptimizer` class and assigns these global variables as the data members of the `QueryOptimizer` instance. The `QueryOptimizer` instance introduced in next paragraph will create and initialize instances of the `SearchStrategy` class, the `SearchSpace` class, and the `Algebra` class and call the `SearchStrategy` instance to complete the query optimization. Finally, the `QueryOptimizationFacade` passes the optimal execution plan of the query from the `QueryOptimizer` instance to the outsider, *e.g.* query execution component. This design allows multi-optimization. In other words, the system can maintain more than one query optimization at the same time, and it leads to an elegant implementation for sub-query case, which will be demonstrated *4.1.3*.

4.1.2 The QueryOptimizer Class

As introduced in *Chapter 3*, we abstract an interface for the optimizers built in framework. The interface called `QueryOptimizer` is a class template of types of a `SearchStrategyType` class, a `SearchSpaceType` class and an `AlgebraType` class. The class template is declared as follows:

```
template <class SearchStrategyType, class SearchSpaceType,
          class AlgebraType >
class QueryOptimizer
{...}
```

`SearchStrategyType` – the class that implements the optimization logic, which would be a subclass of the `SearchStrategy` abstract class.

`SearchSpaceType` – the class that performs an interface for the search strategy to access to the optimization space. In our framework, it is the `SearchSpace` class.

`AlgebraType` – the class that defines objects of the data structures, *i.e.* a list of algebra operators and the associated algebra algorithms, on which the query optimization is carried on.

To create a query optimizer, the user just needs to decide the types of the search strategy, the search space and the query algebra. For example, the following statement creates an optimizer with the strategy of type of `PostgreSQLSearchStrategy`.

```
QueryOptimizer<PostgreSQLSearchStrategy, SearchSpace,
               OperatorAndAlgorithm >*
   optimizer = new QueryOptimizer<PostgreSQLSearchStrategy,
                       SearchSpace, OperatorAndAlgorithm >;
```

Attribute	Description
strategy	An instance of the specified SearchStrategyType class that implements the search strategy used in the optimizer. Initialized by the optimizer.
space	An instance of the specified SearchSpaceType class that maintains the search space of the optimization. Initialized by the optimizer.
opalgo	An instance of the specified AlgebraType class that interfaces to the operator objects and the algorithm objects on which the optimization is carried on. Initialized by the optimizer.
cat	An instance of the Catalog class that represents the catalog information of the underlying database. Passed from the QueryOptimizationFacade.
oopt	An instance of the OptimizationOption class that defines the values of the optimization options. Passed from the QueryOptimizationFacade.
query	An instance of the Query class that represents the user input query to be optimized. Passed from the QueryOptimizationFacade.

Table 1: Attributes of the QueryOptimizer Class

As shown in Table 1, the QueryOptimizer class has six attributes. The QueryOptimizer initializes attribute strategy, attribute space and attribute algebra itself by creating instances of the SearchStrategyType, the SearchSpaceType and the AlgebraType, while the other three attributes – the query, the options and the catalog – are either passed as parameters of the QueryOptimizer constructor or assigned via Set methods from outside, *e.g.* the QueryOptimizationFacade. The optimizer notifies the SearchStrategyType instance the interfaces to the Search Space component and the Algebra component by passing it the instances of the SearchSpaceType and the AlgebraType. The optimizer then calls the DoOptimize() method of the SearchStrategyType instance to complete the query optimization.

4.1.3 The SearchStrategy Class

The `SearchStrategy` class is an abstract class. It defines abstract method `DoOptimize()` to perform the optimization search. Specific search strategies, *e.g.* bottom-up search strategy, dynamic search strategy, *etc.*, must subclass the `SearchStrategy` class and implement the `DoOptimize()` method. While the detailed search logic of different search strategies varies a lot, the collaborations of the three components are basically constant, as illustrated in Figure 22. The `SearchStrategy` picks one operator tree (if a unary operator is applied to expand the tree, *e.g.* `order by`, or transformation rule is used to transform a tree) or two operator trees (if a binary operator is applied to join two trees, *e.g.* `join`) from the `SearchSpace` and picks an algebraic operator if it wants to expand the tree(s) from the `Algebra`. It then registers a corresponding `TreeVisitor`, such as `TreeToTreeVisitor`, via the `SearchSpace` interface to generate a new tree.

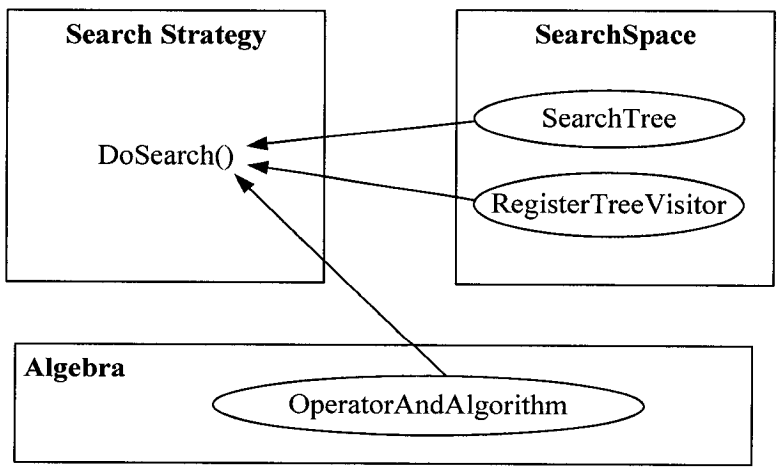

Figure 22: Basic Collaborations of the Three Components

As introduced before, the abstraction of the `QueryOptimizer` class template which localizes the global variables makes it easy to deal with sub-query. Figure 23 shows optimization of query that contains another query in the PostgreSQL-like optimizer. For example, `select foo.name from Cities as c, (select name from Employees e, Persons p where p.name=e.name order by name) as foo where foo.name=c.name;`

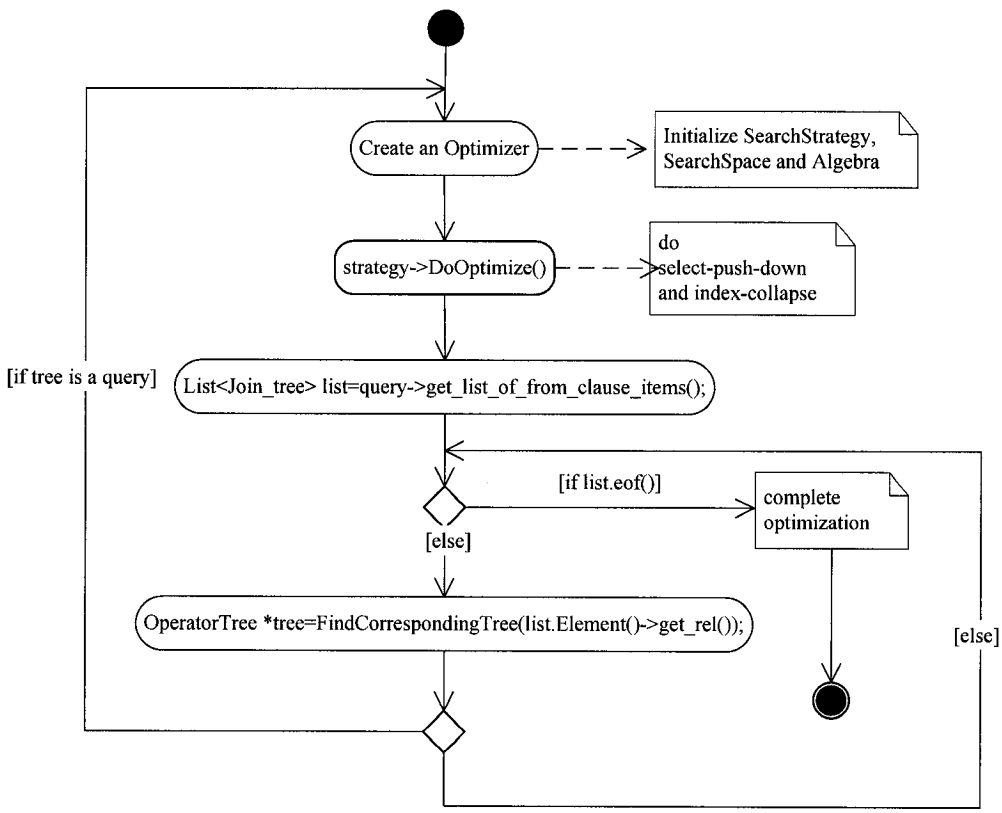

Figure 23: Activity Diagram of Sub-query Optimization

4.2 The Search Space Component

The Search Space component manages the search space the search strategy explores. Major classes in this component include: the `SearchSpace` class, the

SearchTree class, the TreeVisitor class and the TreeGenerator class as shown in Figure 24.

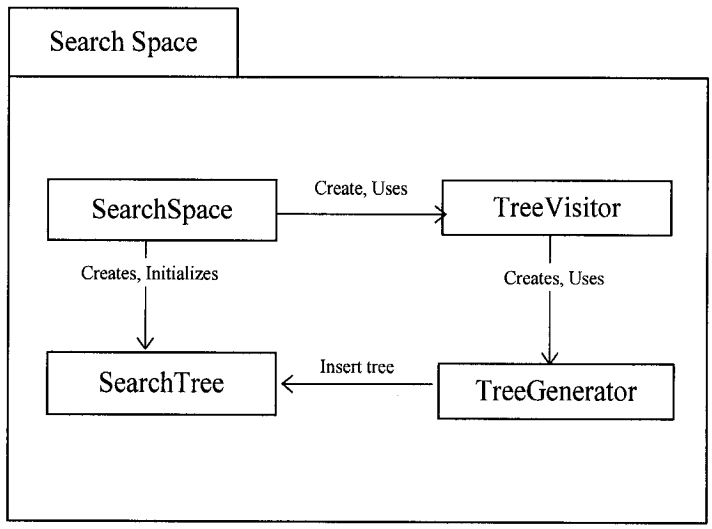

Figure 24: Package Diagram of the Search Space Component

4.2.1 The Search Space Class

The SearchSpace class is the interface for the SearchStrategy class to the Search Space component. It implements the Prune() method delegated by the SearchStrategy class, which is used to narrow the search space to improve the optimization performance. The hash table used by the pruning that was a global variable is now a local data member of the SearchSpace class and is initialized by the class itself. The class also maintains the SearchTree object which holds the OperatorTree instances generated during the course of the optimization. Table 2 lists the attributes of the class and how they are initialized.

Attribute	Description
search	An instance of the SearchTree class that contains the operator trees generated during the optimization. Initialized by the search space.
hashtable	An instance of the HashTable class that is used to prune the algorithm trees associating with the operator trees contained by the **search** attribute.
opalgo	An instance of the specified AlgebraType class that interfaces to the operator objects and the algorithm objects on which the optimization is carried on. Passed from the QueryOptimizer.
cat	An instance of the Catalog class that represents the catalog information of the underlying database. Passed from the QueryOptimizer.
oopt	An instance of the OptimizationOption class that defines the values of the optimization options. Passed from the QueryOptimizer.
query	An instance of the Query class that represents the user input query to be optimized. Passed from the QueryOptimizer.

Table 2: Attributes of the SearchSpace Class

Method	Description
RegisterExpandTreeVisitor (OperatorTree* tree)	Method to create an ExpandTreeVisitor instance to expand the specified operator tree.
RegisterJoinTreeVisitor (OperatorTree* tree1, OperatorTree* tree2)	Method to create a JoinTreeVisitor instance to join the two specified operator trees.
RegisterTransformTreeVisitor (OperatorTree* tree)	Method to create a TransformTreeVisitor instance to transform the specified operator tree to another algebraically-equivalent operator tree.
RegisterTreeToPlanVisitor (OperatorTree* tree)	Method to create a TreeToPlanVisitor instance to convert the specified operator tree to a list of algorithm trees.
RegisterPlanToPlanVisitor (AlgorithmTree* tree)	Method to create a PlanToPlanVisitor instance to transform the specified algorithm tree to another algebraically-equivalent algorithm tree.

Table 3: Tree Register – Methods in the SearchSpace Class

The optimization search space is accessed via calling tree visitor registering methods defined in the `SearchSpace` class to register `TreeVisitor` instances, which call corresponding `TreeGenerators` to complete the request – tree generation. Table 3 lists the tree visitor registering methods defined in the `SearchSpace` class.

4.2.2 The SearchTree Class

The `SearchTree` class is an `OperatorTree` instance container. It maintains the operator trees generated during the course of the optimization. Table 4 lists the data members of the `SearchTree` class. The main data structure is an array of lists - `listofunexpandednodes`. Each list contains operator trees of which the number of nodes equals to the index of the list in the array. There is also a list that maintains the database entities involving in the query called `listofrootnodes`, which compose the leaf nodes of the operator trees built in the optimization. Data member `level` is an index into the `ListofUnexpandedNodes`. The `optimumnode` is the place to put the optimization result. The GET/SET methods are defined to encapsulate these data members.

Table 5 shows a simple search tree example in the PostgreSQL optimizer for optimizing query *Select * from A , B, C where A.a=B.b and B.c = C.c*. Assume the best plan is {{A,B},C}.

Attribute/Method	Description
`listofrootnodes`	A list of operator trees that represent the entities involved in the query. They fill the leaves of other operator trees.
`listofunexpandednodes`	An array of lists of operator trees that generated during the optimization. The operator trees in each list contain the same number of nodes.
`level`	An integer as an index into the `listofunexpandednodes`.
`optimumnode`	Place to save the optimization result.

Table 4: Attributes of the `SearchTree` Class

`optimumnode`	Join /\ join C /\ A B			
`listofunexpandednodes`	Join /\ join C /\ A B	Join /\ join A /\ B C	Join /\ join B /\ A C	`level=2`
	Join /\ A B	Join /\ B C	Join /\ A C	`level=1`
	A	B	C	`level=0`
`listofrootnodes`	A	B	C	

Table 5: A Simple Search Tree Example in the PostgreSQL-like Optimizer

4.2.3 Visitors and Generators

Table 6 lists the types of `TreeVisitor` classes implemented in the framework. The tree visitors implement the algorithms on the data structure residing in the Algebra component. To achieve a cleaner layout, the implementation of the visitors is delegated to a set of `TreeGenerator` classes. Figure 18 shows the `TreeVisitor` and `TreeGenerator` hierarchies and their collaborations.

Visitor	Behavior
`ExpandTreeVisitor`	Subclass of `TreeToTreeVisitor` with one `OperatorTree` attribute that creates `ExpandTreeGenerator` object to expand an operator tree via the `Operator` object, which will be added to the operator tree as the root.
`JoinTreeVisitor`	Subclass of `TreeToTreeVisitor` with one more `OperatorTree` attribute than its super class. It deals with the specific characteristics of the `Join` operator. It creates `BinaryOperatorExpand` generator to join two operator trees.
`TransformTreeVisitor`	Subclass of `TreeToTreeVisitor` with one `OperatorTree` attribute that creates `TransformTreeGenerator` object to transform an operator tree to an alternative.
`TreeToPlanVisitor`	Class defines interfaces for converting an operator tree to an algorithm tree via the selected algorithm that associates with the operator that roots the operator tree. It initializes the corresponding `TreeToPlanGenerator` object to do the job.
`PlanToPlanVisitor`	Class defines interfaces for plan to plan generations, which mainly happen when the operation represented by an `Enforcer` is needed to be performed on an algorithm tree before certain algorithm can be applied to that algorithm tree in the framework.

Table 6: Definitions of Tree Visitors

4.3 The Algebra Component

The Algebra component maintains the query algebra – `Operator` classes and their associated `Algorithm` classes. It also contains the main data structure of the framework – the `OperatorTree`, the `AlgorithmTree` and their property classes – the `OperatorTreeProperty`, and the `AlgorithmTreeProperty`. The `Cost` class that calculates the times consumed while executing the `Algorithm` objects is also implemented in this component. Figure 25 details the component.

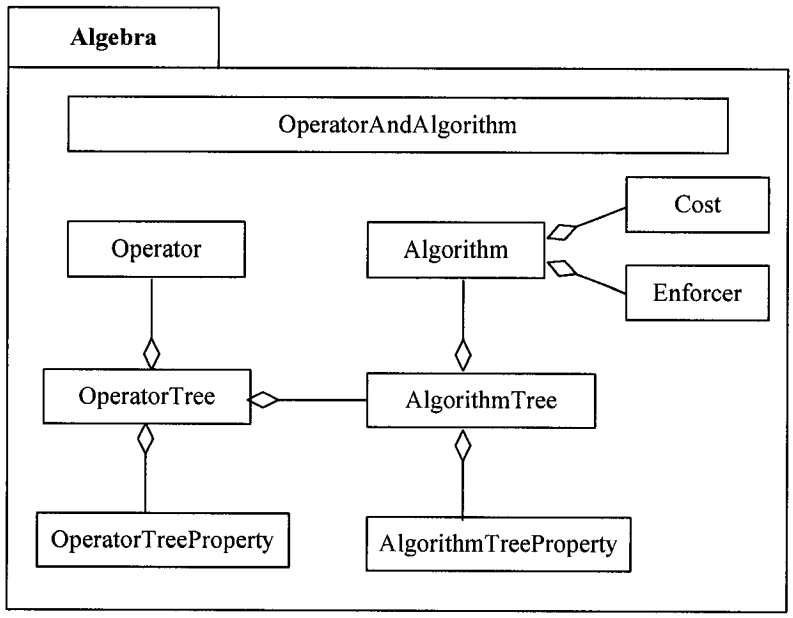

Figure 25: Package Diagram of the Algebra Component

4.3.1 Operator and Algorithm

The `Operator` class defines a general interface of an operator in the query algebra, and the `Algorithm` class defines a general interface of an algorithm that represents the implementation of an operator on data in the underlying database. In the new framework, the `Clone()` method defined in the `Operator` class that

fetches run-time information from the user input query associated with the operator are delegated to the corresponding `TreeGenerator` class.

The `OperatorAndAlgorithm` class is the interface for the Search Strategy component to access the query algebra defined in the Algebra component. It creates concrete objects of the `Operator` and `Algorithm` classes that will be used by the `SearchStrategy` object.

There are three types of Set object in terms of the elements of the Set in the framework: a `Set` holding attributes of the `Query`, a `Set` holding relations of the `Query` and a `Set` holding operations of the `Query`. The sizes of a `Set` object has to be determined when it is initialized, which is obtained from the `Query` object that represent the query to be optimized. In the previous version, the `Set` attributes of the concrete objects of the `Operator` and `Algorithm` classes are initialized by getting the `Query` object from hard coding `GlobalVariable()->query`, which obviously is very limited. In the new framework, a new method `InitDBAlgebraSets(Query * query)` is added to class `OperatorAndAlgorithm` to enhance the flexibility of initializing the `Set` attributes of the concrete objects of the `Operator` and `Algorithm` classes.

4.3.2 OperatorTree and AlgorithmTree

The `OperatorTree` class (with the `OperatorTreeProperty` class) aggregated of the `Operator` class describe a logic plan and the `AlgorithmTree` class (with the `AlgorithmTreeProperty` class) aggregated of the `Algorithm` class that

describe a physical plan are moved to this component from the Search Strategy component in the new framework.

A list of `AlgorithmTree` of the `OperatorTree` object is generated during the construction of the `OperatorTree` object by calling the corresponding `TreeToPlanGenerators`. With the old visitors and generators structure, the implementation of this process is very complicated, as shown in Figure 26. With the new visitors and generators structure, the algorithm tree generation process is simplified and shown in Figure 27.

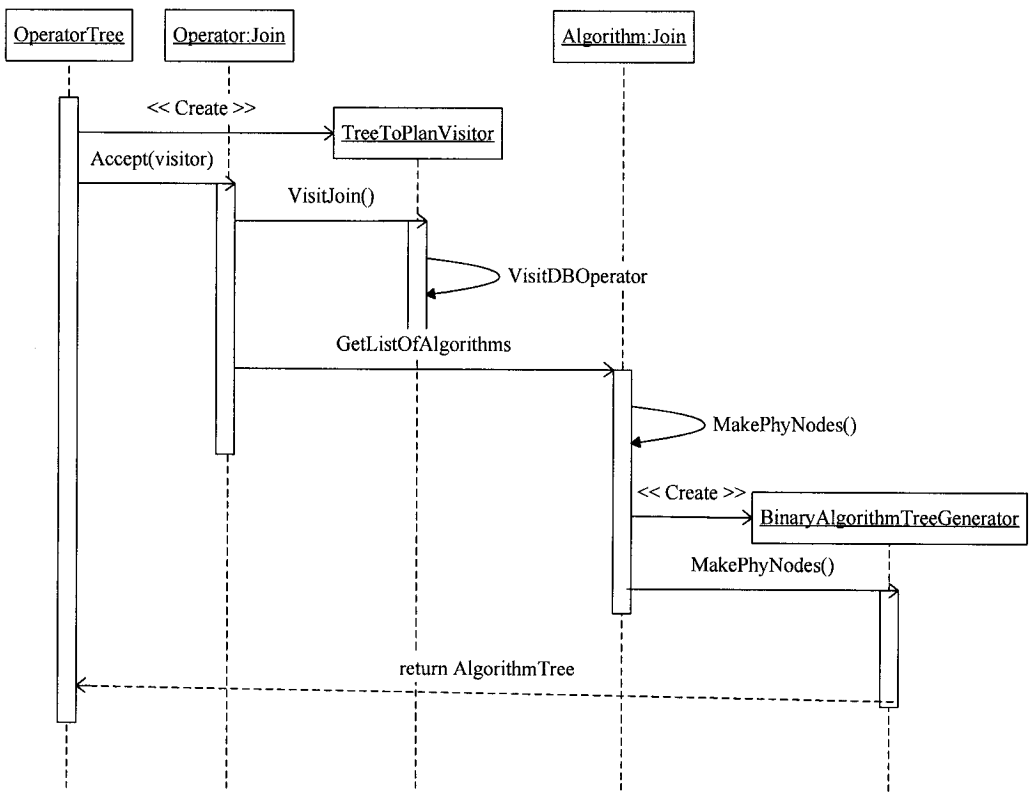

Figure 26: Implementation of Tree to Plan Conversion in the Old Framework

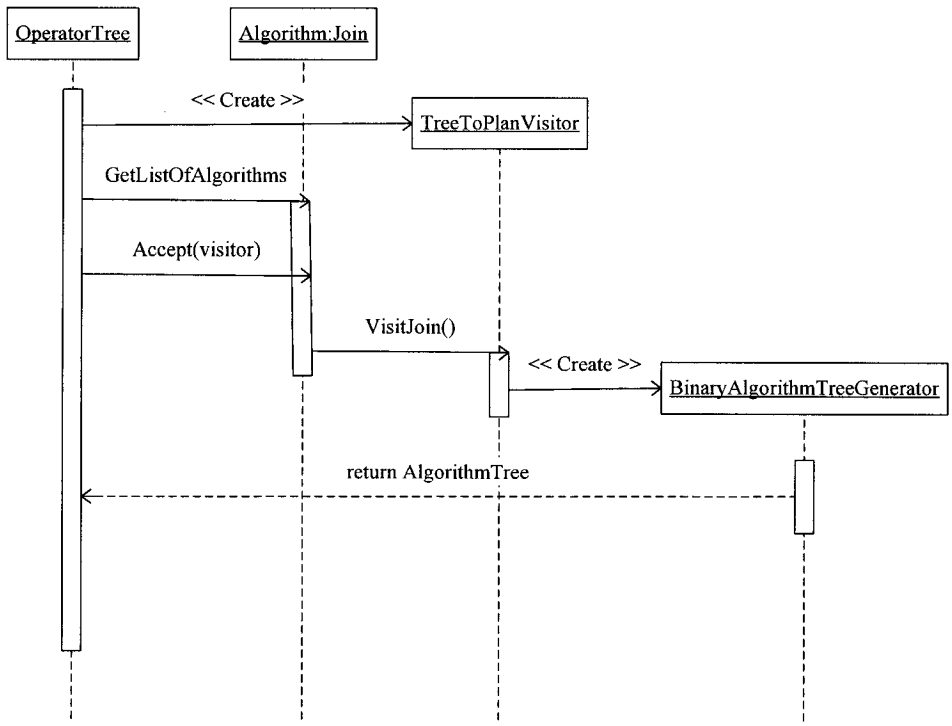

Figure 27: Implementation of Tree to Plan Conversion in the New Framework

4.3.3 The Cost Model

The `Cost` class associated with the `Algorithm` class implements the cost mechanism of the framework and is moved to this component from the Search Strategy component. The `Algorithm` class has a `Cost` object as its data member, which is in charge of computing the cost in terms of times used to execute the algorithm. This process is performed while a new algorithm tree is created.

During our study of the performance issue of the PostgreSQL-like optimizer, we found out that constructing a new tree is very expensive. Furthermore, a lot of algorithm trees will be constructed during the optimization. In the previous version, the `TreeToPlanGenerator` class constructs an algorithm tree when its cost is

calculated and then performs the cost-based pruning. The newly created algorithm tree might happen be the target of the pruning. If this is the case, the construction of the algorithm tree is actually a waste.

In the new framework, we enhance the `Cost` class and modify the `TreeToPlanGenerator` classes to perform the cost calculation of an algorithm tree that is intended to create before really create one, which provides around ten percent better performance on average.

4.4 Performance Test of the PostgreSQL-like Optimizer

# Joins	Native PostgreSQL Optimizer *	Original PostgreSQL-like Optimizer *	Improved PostgreSQL-like Optimizer *
1	300	341	241
2	553	915	617
3	873	2944	1527
4	1880	6506	3462
5	2917	13528	6764
6	4279	24425	11016
7	6025	42138	18093
8	8232	68185	27947
9	11051	103955	41284
10	647101	2957896	1242340
11	722162	3509801	1366913
12	799932	4064398	1499023
13	881062	4613137	1630084
14	966972	5298748	1790419
15	1050195	6043457	1909302
16	1139038	6861692	2082104
17	1227845	7734251	2233596
18	1322223	8541668	2396811
19	1417649	9472771	2549617
20	1496809	9962640	2628904
*: the numbers are in Microsecond.			

Table 7: Performance Testing Result of the PostgreSQL-like Optimizer

Table 7 reports the performance comparison among the native PostgreSQL optimizer, the original PostgreSQL-like optimizer and the improved PostgreSQL-like optimizer in terms of the optimization times in microseconds per query. The testing is performed on the "haida" server at Concordia University. The operating system is linux2.4.20. The C++ compiler is GNU g++ 3.2.2. And the native PostgreSQL version is 7.2.1.

As listed in Table 7, the new PostgreSQL-like optimizer gains an obvious improvement in performance. For instance, the time used to optimize a query with nine joins drops from ten times slower than the native PostgreSQL optimizer to around four times. When there are more than 10 joins in a query, the genetic algorithm is used (in both systems). The optimizer needs to initialize a fixed size (1024 by default) pool of chromosomes, so more trees are needed to create and the time increases drastically. For this case, the new PostgreSQL optimizer gets around five times improvement when the number of joins reaches 20.

The following factors contribute to the remaining performance difference (about two times) between the native PostgreSQL optimizer and the PostgreSQL-like optimizer built in our framework.

- The native PostgreSQL optimizer does not generate logical operator trees, while the PostgreSQL-like optimizer in our framework does. Since the operator tree and the algorithm tree have similar data structures as shown in Figure 4 and Figure 5, initializing an operator tree needs the same effort of initializing an algorithm tree.

- The generic framework design does result in minor degradation of performance. The native PostgreSQL optimizer written in C is a "custom-made" optimizer, while the PostgreSQL-like optimizer is written in C++ and built from an extensible framework that aims to be flexible and extensible.

Chapter 5

Conclusion

Query optimization has been studied over twenty years. However, it is still an active research subject due to the expansion of database management system. New algebra and new search techniques are continuously introduced. Therefore, there is a need to build an extensible query optimizer, which allows easily adding and modifying query algebra and meanwhile can easily switch among different search strategies.

In this thesis, we describe a third-generation extensible query optimization framework, which addresses the issues arising from building a simple bottom-up query optimizer and an instance of PostgreSQL query optimizer in the previous generation. The third-generation framework also improves the reusability, the extensibility and the performance of the framework.

Frameworks are reuse technology that have attracted the attention in building query optimizer. OPT++, on which our work is based, is a well-designed extensible query optimization framework. On the whole, the three-component decomposition of OPT++ conforms to the nature of query optimization. There is a good separation between responsibilities of the three major components and the relationships among the classes are well defined. However, previous studies show limitations of OPT++

and substantial needs to improve it in the detailed design and implementation.

Understanding a framework is a key problem in framework-based development. In this thesis, we gained experience in understanding and reengineering a query optimization framework. We benefited from the valuable guidance provided by Jinmiao Li's documentation and cook books for studying the query optimization framework. We also confirmed that once learned a framework significantly increases the productivity of application developers.

The reengineering improves the reusability, the extensibility and understandability of the query optimization framework. An interface to the optimizer built in the framework has been abstracted, which offers a straightforward way to experience the flexibility of the system (switch between different search strategies and different sets of query algebra). Global variables that tightly bind the three components have been localized to the optimizer object, which makes it possible to have more than one optimization process with different optimization context (*e.g.* different search strategy, different optimization options, *etc.*) in the system. That in turn leads to an elegant implementation of sub-query processes. The interfaces to the three components are abstracted or refined, and the functionalities among the components are reassigned, which clearly decouple the three components and produce more extensibility.

The reengineering also addresses the performance issue of the framework. The performance of the PostgreSQL-like optimizer has been improved obviously. For example, to optimize a query with nine joins, the time used drops from 10 times

slower than the native PostgreSQL optimizer to around 4 times. And the time drops from 6.6 times to 1.7 times when the number of joins reaching 20, where the genetic algorithm is used and a fixed size (1024 by default) pool of join plans are needed to be initialized.

While the two applications, the simple bottom-up optimizer and the PostgreSQL-like optimizer are both for relational databases, we believe the framework also works for object-oriented databases. In the future, we need to build an object-oriented query optimizer in the framework to further prove the fitness of the query optimization framework as a sub-framework in a database management system framework.

Bibliography

[1] Scott W. Ambler. *The Object Primer*. Second Edition. Cambridge University Press, 2001.

[2] José A. Blakeley, William J. McKenna, Goetz Graefe. *Experiences Building the Open OODB Query Optimizer.* Proceedings of the 1993 ACM international conference on management of data, Volume 22, Issue 2, Page 289-296.

[3] G. Butler, R.K.Keller, H.Mili. *A framework for framework documentation.* ACM Computing Surveys 32,1 (March 2000) electronic symposium.

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

[5] Guido Cardinot, Francesco Baruehellit, Andrea Valerio. *The Evaluation of Framework Reusability*. ACM SIGAPP Applied Computing Review, Volume 5 Issue 2, Sep. 1997.

[6] Peter Coad, David North, Mark Mayfield. *Object Models Strategies, Patterns, and Applications*. Prentice Hall, 1995.

[7] Stephen C. Dewhurst. *C++ Gotchas Avoiding Common Problems in Coding and Design*. Addison Wesley, 2003.

[8] Anton Eliens. *Principles of Object-Oriented Software Development.* Addison Wesley, 1995.

[9] Mohamed Fayad, Douglas C. Sdhmidt. *Object-Oriented Application Frameworks.* Communications of the ACM. Vol. 40, No. 10, October 1997.

[10] M. E. Fayad, D. C.Schmidt, and R. E.Johnson, *Building Application Frameworks.* Addison-Wesley Pub Co, 1st edition, 1999.

[11] Johann Christoph Freytag. *The Basic Principles of Query Optimization in Relational Database Management Systems.* ACM, IFIP Congress,1989, 801-807.

[12] Johann Christoph Freytag. *A Rule-Based View of Query Optimization.* Proceedings of the 1987 ACM SIGMOD international conference on Management of data, Volume 16, Issue 3, Page 173-180.

[13] Erich Gamma, Richard Helm, Ralph Johnson, Jon Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[14] Hector Garcia-Molina, Jeff Ullman, Jennifer Widom. *Database System: The Complete Book.* Prentice Hall, 2002.

[15] David Garlan, Robert Allen, John Ockerbloom. *Architectural Mismatch or Why it's hard to build systems out of existing parts.* Proceedings of the Seventeenth International Conference on Software Engineering, Seattle WA, April 1995, Page 179-185.

[16] G. Graefe, W.J. McKenna. *The Volcano Optimizer Generator: Extensibility and Efficient Search.* Proceedings of the ninth International Conference on Data Engineering, 1993, Page 209-218.

[17] G. Graefe, David J. DeWitt. *The EXODUS Optimizer Generator.* Proceedings of the 1987 ACM SIGMOD international conference on Management of data, Volume 16 Issue 3, Page 160-172.

[18] L. M. Haas, J. C. Freytag, G. M. Lohman, H. Pirahesh. *Extensible query processing in Starburst.* Proceedings of the 1989 ACM SIGMOD international conference on Management of data, Volume 18 Issue 2, Page 377-388.

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

[20] Navin Kabra, David J. DeWitt. *OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization.* VLDB Journal, Volume 8, No.1, Pages 55-78, May 1999.

[21] Mavis K. Lee, Johann Christoph Freytag, Guy M. Lohman. *Implementing an Interpreter for Functional Rules in a Query Optimizer.* Proceedings of the fourteenth VLDB Conference, 1988, Page 55-78.

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

[23] Ray Lischner. C++ In a Nutshell: *A Language & Library Reference.* First Edition. O'Reilly, 2003.

[24] Marcus Eduardo Markiewicz, Carlos J.P. Lucena. *Object Oriented Framework Development.* ACM, Crossroads, Volume 7, Issue 4, July, 2001.

[25] Scott Meyers. *Effective C++--50 Ways to Improve Your Programs and Designs.* Second Edition. Addison-Wesley, 1998.

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

[27] Tomasz Muldner. *C++ Programming with Design Patterns Revealed.* Addison Wesley, 2002.

[28] David R. Musser, Gillmer J. Derge, Atul Saini. *STL Tutorial and Reference Guide.* Second Edition. Addison Wesley, 2001.

[29] PostgreSQL 7.3.3 Documentation. http://www.PostgreSQL.org/idocs/

[30] PostgreSQL 7.3.3 Source Code. http://www.PostgreSQL.org/

[31] Raghu Ramakrishnan, Johannes Gehrke. *Database Management Systems*. Second Edition. Mc-Graw-Hill Higher Education, 2000.

[32] Robert Robson. *Using the STL: the C++ standard template library.* Second edition. Springer-Verlag, 2000.

[33] Giedrius Slivinskas, Christian S. Jensen. *Enhancing an Extensible Query Optimizer with Support for Multiple Equivalence Types.* Http://www.cs.auc.dk/~csj, 2001.

[34] David Vandevoorde, Nicolai M. Josuttis. *C++ Templates the Complete Guide.* Addison Wesley, 2003.

[35] Ju Wang, G. Butler, Jinmiao Li. *Implementing the PostgreSQL Query Optimizer within the OPT++ Framework.* Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference, 2003. Page 262-271.

[36] Ju Wang. *Implementing the PostgreSQL Query Optimizer Within The OPT++ Framework.* Master Thesis. Concordia University, 2002.

Appendix

A. Search Strategy Component

A.1 QueryOptimizerFacade.h

```
#ifndef QUERYOPTIMIZERFACADE_H
#define QUERYOPTIMIZERFACADE_H

#include <optdef.h>
#include <QueryOptimizer.h>
#include <Aquery.h>
#include <Aphynode.h>

/*
 *A thread waiting for query from the parser. It packs the query received
 * into a queue and triggers method Optimize(), which will initialize an
 * optimizer to optimize the queries in the queue. The result of the
 * optimization will be pack into another queue for further use.
*/

class QueryOptimizerFacade
{
public:
  QueryOptimizerFacade();
  virtual ~QueryOptimizerFacade();
  virtual void Optimize();
  void SetQuery(Query* query) {this->thequery = query;}
  Query* GetQuery(void) {return this->thequery;}
  AlgorithmTree* GetResult(void) {return this->theresult;}

protected:
  virtual void PreprocessQuery() = 0;
  virtual AlgorithmTree* DoOptimize(Aquery_t* query);

private:
  virtual void InitializeVariables();
```

```
  virtual void CleanUpVariables();
  virtual void ReadSystemCatalog();

public:
  Query* thequery; //the query passed from the parser
                   //should change to a queue in the future

  AlgorithmTree* theresult; //the result of optimizing the query
                            //should change to a queue in the future
};

class QueryOptimizerFacadeWithParser: public QueryOptimizerFacade
{
protected:
  virtual void PreprocessQuery();
};

class QueryOptimizerFacadeWithFormatedFile: public QueryOptimizerFacade
{
protected:
  virtual void PreprocessQuery();
};
#endif /* QUERYOPTIMIZERFACADE_H */
```

A.2 QueryOptimizer.h

```
#ifndef QUERYOPTIMIZER_H
#define QUERYOPTIMIZER_H

#include <optdef.h>
#include <SearchStrategy.h>
#include <SearchSpace.h>
#include <Aopdefs.h>
#include <SearchTree.h>
#include <Acost.h>
#include <Aquery.h>
#include <Acat.h>
#include <Ahash.h>
#include <Aoptions.h>
#include <Alognode.h>
#include <Aphynode.h>
```

```
template <class SearchStrategyType, class DBAlgebraType, class SearchSpaceType>
class QueryOptimizer
{
private:
    SearchStrategyType* strategy;
    DBAlgebraType* opalgo;
    SearchSpaceType* space;
    Catalog* cat;   //DB-specific
    OptimizerOptions* oopt;//user-specific
    Query* query; //user-specific

public:
    QueryOptimizer(){}
    QueryOptimizer(OptimizerOptions* oopt, Catalog* cat)
    {
        this->oopt = oopt;
        this->cat = cat;
    }
    ~QueryOptimizer()
    {
        if(strategy) delete strategy;
        if(space) delete space;
        if(opalgo) delete opalgo;
        if(oopt) delete oopt;
        if(query) delete query;
    }
    SearchStrategyType* GetSearchStrategy(void) {return strategy;}
    SearchSpaceType* GetSearchSpace(void) {return space;}
    DBAlgebraType* GetDBAlgebra(void) {return opalgo;}

    void SetOption(OptimizerOptions* oopt) {this->oopt = oopt;}
    void SetCatalog(Catalog* cat){this->cat = cat;}
    void SetQuery(Query* query) {this->query = query;}

    virtual void Initialize();
    virtual OperatorTree* Optimize(Query* query);
};

#endif /* QUERYOPTIMIZER_H */
```

A.3 SearchStrategy.h

```
#ifndef SEARCHSTRATEGY_H
```

```
#define SEARCHSTRATEGY_H

#include <optdef.h>
#include <SearchTree.h>
#include <SearchSpace.h>
#include <Aquery.h>
#include <Acat.h>
#include <Alognode.h>
#include <Aphynode.h>
#include <Ahash.h>
#include <stack.h>
#include <Geqo.h>
#include <Acost.h>

class SearchStrategy
{
public:
  SearchStrategy(){}
  virtual ~SearchStrategy(){}

  virtual SearchSpace* GetSearchSpace(){return this->space;}
  virtual void SetSearchSpace(SearchSpace* space){this->space = space;}
  virtual Query* GetQuery() {return this->query;}
  virtual void SetQuery(Query* query){this->query = query;}
  virtual Catalog* GetCatalog() {return this->cat;}
  virtual void SetCatalog(Catalog* cat){this->cat = cat;}
  virtual OperatorAndAlgorithm* GetDBAlgebra(){return this->opalgo;}
  virtual void SetDBAlgebra(OperatorAndAlgorithm* opalgo)
  {this->opalgo = opalgo;}
  virtual OptimizerOptions* GetOption(void){return this->oopt;}
  virtual void SetOption(OptimizerOptions* oopt) {this->oopt = oopt;}
  virtual OperatorTree* DoOptimize()=0;
  virtual OperatorTree* DoSearch()=0;

protected:
  OperatorAndAlgorithm* opalgo;
  OptimizerOptions* oopt;
  Catalog* cat;
  SearchSpace* space;
  Query* query;
};

class BottomupSearchStrategy: public SearchStrategy
{
```

```
public:
  BottomupSearchStrategy():SearchStrategy(){}
  ~BottomupSearchStrategy(){}

  virtual OperatorTree* DoOptimize();
  virtual OperatorTree* DoSearch();
  virtual void ExpandNode(OperatorTree* node);
};

class TransformativeSearchStrategy: public SearchStrategy
{
public:
  TransformativeSearchStrategy():SearchStrategy(){}
  ~TransformativeSearchStrategy(){}

  virtual OperatorTree* DoOptimize();
  virtual OperatorTree* DoSearch();
  void ConstructInitialTree (void);
  virtual void ExpandNode (OperatorTree* node);
};

class PostgresqlSearchStrategy: public SearchStrategy
{
protected:
    stack<SearchSpace*> space_garbage_bin;//clean up at the end of optimizaiton.
                        //cannot delete space which will delete SearchTree
                       //during subquery optimization. Must wait to the end.

    stack<HashTable*> hash_stack;//used by Genetic Algorithm

public:
  PostgresqlSearchStrategy():SearchStrategy() {}
  ~PostgresqlSearchStrategy()
  {
     while(!space_garbage_bin.empty())
     {
        SearchSpace* tempSpace = space_garbage_bin.top();
        space_garbage_bin.pop();
        delete tempSpace;
      }
  }

  virtual OperatorTree* DoOptimize();
  virtual OperatorTree* DoSearch();
```

```
    Alognode_t* GimmeTree(Gene *tour, int num_gene, int rel_count, Alognode_t*
old_tree);
    int GimmePoolSize(int nr_rel);
    Cost GeneticEvaluate(Gene *tour, int num_gene);

protected:
    virtual void PreprocessExpression(void);
    virtual void SetInitPlans(OperatorTree* optimal_tree);
    void TraverseAnOperation(Aptree_t* ptree);

private:
   // just for convenience. in geqo, it is more convenient to use an array
  //to genete a tree than to use a list.
    Alognode_t **root_rel_array;

private:
  //wj: rels in from-clause have been expanded by select and indexcollapse
  //operators, so i have to find which tree contains the rel I want.
    Alognode_t *  FindCorrespondingTree( Arel_t * rel);

  // wj: in a from-clause, there may be explicite join ().
  //This is function is to convert such a join into a log node.
  //NOTE: Each of inputs of a join could be a join or a rel.
    Alognode_t * MakeOneExplicitJoin(Join_tree * join_tree);

    void DynamicProgramming(int level);
    int IsLeftJoinTree(Alognode_t *log_node);
    int IsRightJoinTree(Alognode_t *log_node);
    int IsIntersect(Alognode_t* lower_log_node,Alognode_t* upper_log_node);
    void GeneticOptimize(void);
    void DeleteGeneticTree(Alognode_t* logtree,int num_gene);
};
#endif /* SEARCHSTRATEGY_H */
```

B. Search Space Component

B.1 SearchSpace.h

```
#ifndef SEARCHSPACE_H
#define SEARCHSPACE_H
```

```
#include <optdef.h>
#include <Alist.h>
#include <Aoptions.h>
#include <Ahash.h>
#include <Acat.h>
#include <Aquery.h>
#include <Aopalgos.h>
#include <Alognode.h>
#include <SearchTree.h>
#include <TreeVisitor.h>

class SearchSpace
{
public:
  OperatorAndAlgorithm* opalgo;     // operators and algorithms
  OptimizerOptions* oopt;   // option controlling the optimizer
  Catalog* cat;         // the catalog object
  Query* query;     // the query being optimized
  SearchTree* search;
  HashTable* hashtable; // APG internal hashtable
  double lowestCost; //for performance, used by Alognode_t::MakePhyNodes
          // and BinaryAlgorithmTreeGenerator::MakePhyNodes to
        // precompute the cost before really generate a new algorithm tree
  int firstTry; //for performance used by Alognode_t::MakePhyNodes
          // and BinaryAlgorithmTreeGenerator::MakePhyNodes to
        // precompute the cost before really generate a new algorithm tree
public:
  SearchSpace()
  {
        this->search = new SearchTree;
        this->hashtable = new HashTable(1021);
  }
  SearchSpace(OperatorAndAlgorithm* opalgo = 0, OptimizerOptions* oopt = 0,
Catalog* cat = 0, Query* query = 0)
    {
        this->opalgo = opalgo;
        this->oopt = oopt;
        this->cat = cat;
        this->query = query;
        this->search = new SearchTree;
        this->hashtable = new HashTable(1021);
    }
    SearchSpace(OperatorAndAlgorithm* opalgo, OptimizerOptions* oopt,
                Catalog* cat, Query* query,
```

```
                SearchTree* search, HashTable* hashtable)
    {
        this->opalgo = opalgo;
        this->oopt = oopt;
        this->cat = cat;
        this->query = query;
        this->search = search;
        this->hashtable = hashtable;
    }
    virtual ~SearchSpace()
    {
        if (this->search) delete this->search;
        if (this->hashtable)
        {
            this->hashtable->EmptyHashTable();
            delete this->hashtable;
        }
    }
    void SetDBAlgebra(Aopalgo_t* opalgo) {this->opalgo = opalgo;}
    void SetOption(Aoptimizeroptions_t* oopt) {this->oopt = oopt;}
    void SetCatalog(Catalog* cat) {this->cat = cat;}
    void SetQuery(Query* query) {this->query = query;}
    void SetSearchTree(SearchTree* search) {this->search = search;}
    void SetHashtable(Ahashtable_t* hashtable) {this->hashtable = hashtable;}

    virtual void MakeInitialTree(void)
    {
        search->SetOptimumNode(0);
        OperatorTree* tree = 0;
        RegisterExpandTreeVisitor(tree).VisitDBRelation(this->opalgo->get);
    }
    List<DBOperator> &GetListOfOperatorsToApply (OperatorTree *node)
    {
      return static_cast<List<DBOperator>&> (opalgo->all_operators);
    }

    virtual ExpandTreeVisitor RegisterExpandTreeVisitor(OperatorTree* tree)
    {
      ExpandTreeVisitor visitor(tree, this);
      return visitor;
    }
    virtual JoinTreeVisitor RegisterJoinTreeVisitor(OperatorTree* tree,
OperatorTree* otherTree)
    {
```

```
        JoinTreeVisitor visitor(tree, otherTree, this);
        return visitor;
    }
    virtual TransformTreeVisitor RegisterTransformTreeVisitor(OperatorTree*
tree)
    {
        TransformTreeVisitor visitor(tree, this);
        return visitor;
    }
    virtual TreeToPlanVisitor RegisterTreeToPlanVisitor(OperatorTree* tree)
    {
        TreeToPlanVisitor visitor(tree, this);
        return visitor;
    }
    virtual void Prune (AlgorithmTree *phynode);
};

//Called by Prune
inline int mcond (int condition, int anti)
{
  return anti ? !condition : condition;
}
#endif /* SEARCHSPACE_H */
```

B.2 SearchTree.h

```
#ifndef SEARCHSTREE_H
#define SEARCHSTREE_H

#include <optdef.h>
#include <Alognode.h>
#include <Aquery.h>
#include <Alist.h>

//#define ADEBUG

const int MaximumOperations = 100;

class SearchTree {
protected:
  List<OperatorTree> listofrootnodes;
                // should be renamed
                // listofatomicnodes someday.
```

```
  List<OperatorTree> listofunexpandednodes[MaximumOperations];
                // array of lists.
                // each list contains a list of unexpanded
                // nodes with that many operations.
  int level;    // index into above array.
                // keeps track of which list is being expanded currently
  OperatorTree *optimumnode;//to put the final result

public:
#ifdef ADEBUG
  NodeCounter lognode_ctr; // some performance statistics
  NodeCounter phynode_ctr;
  NodeCounter subopt_lognode_ctr;
  NodeCounter subopt_phynode_ctr;
#endif

  SearchTree (void);
  virtual ~SearchTree (void);

  void Initialize (void);   // call before each query
  void CleanUp (void);      // call after each query

  int GetLevel(void)  {return this->level;}
  void SetLevel(int l) {this->level = l;}
  OperatorTree* GetOptimumNode(void) {return optimumnode;}
  void SetOptimumNode(OperatorTree* node) {this->optimumnode = node;}

  int GetLengthOfRootNodes(void) {return listofrootnodes.Length();}
  List<OperatorTree> &GetListOfRootNodes (void);
  void DeleteListOfRootNodes(void);
  void SetListOfRootNodes(Alist_t<Alognode_t>& nodes);
  int GetLengthOfNodes(void) {return listofunexpandednodes->Length();}
  int GetLengthOfNodesByLevel(int level)
    {return listofunexpandednodes[level].Length();}
  List<OperatorTree> &GetListOfNodesCurrent(void);
  List<OperatorTree> &GetListOfNodesByLevel(int level);
  void DeleteListOfNodesByLevel(int level);
  void SetListOfNodesByLevel(int level, List<OperatorTree>& nodes);

  void AddNodeToTree (OperatorTree *node);
  virtual void NewNode (OperatorTree *node);
  virtual void DeleteLogNode (OperatorTree *lognode); // remove from tree.

  int IsExisting(Alognode_t*right_log_node,Alognode_t*left_log_node);
```

```
                    // in Dynamic Programming, if i don't test this, unnecessary
                    // repeations will occur, because the emelents of
                    // listofunexpandednodes[1]
                    // are actually joined with the elements of the same list.
  Alognode_t* GetBestLogTree(void);
  void print(void);
};

class NodeCounter
{
private:
  int tot_nodes;        // total number of nodes created
  int cur_nodes;        // number of nodes in tree currently
  int max_nodes;        // maximum nodes coexisting at a given time
public:
  NodeCounter (void) {reset ();}
  ~NodeCounter (void) {}

  void reset (void) {tot_nodes = cur_nodes = max_nodes = 0;}
  void add_node (void) {
    tot_nodes++; cur_nodes++;if (cur_nodes>max_nodes) max_nodes=cur_nodes;
  }
  void delete_node (void) {cur_nodes--;}
  void write (ostream &os) const {
    os << "Total: " << tot_nodes << "\tMax: " << max_nodes;
  }

  int tot (void) const {return tot_nodes;}
  int max (void) const {return max_nodes;}
};

//called by node counter
inline int is_real_lognode (OperatorTree *node)
{
  return node->GetOp ()->GetNumber () != Aidx_collapse;
}
inline ostream &operator<< (ostream &os, const NodeCounter &n)
{
  n.write (os); return os;
}
#endif /*SEARCHSTREE_H*/
```

B.3 Ahash.h

```
#ifndef AHASH_H
#define AHASH_H

#include <Alist.h>
class Aphynode_t;
class Alogprop_t;
class Aphyprop_t;

/*****************************************************************************
  BEWARE!!!
  do not make any changes to this file unless you are very
  very sure of what you are doing.
*****************************************************************************/
/*****************************************************************************
  PRUNING.
  whenever a new physical node is created
  there is potential for ii_cost based pruning.

  remember that each physical node stands for an access plan
  (for whatever partial operator tree it is supposed to be
  implementing).

  so if there exists a physical node in our search tree which
  produces the the same output as the new node (i.e. all the
  logical properties are same AND the physical properties are also
  the same) then we keep only the the less expensive one around
  to be considered for further optimization. the more expensive
  one is pruned out. we call this an EXACT MATCH.
  in this case, if the new node is less expensive then it
  REPLACEs the older one in the search tree or if it is more
  expensive then (since it is not useful to anyone) it commits SUICIDE
  by deleting itself.

  if no exact match is found we continue with pruning.

  if there is a physical node whose logical properties are the
  same as the new node and its physical properties are not
  but its physical properties are NOT INTERESTING
  and it is more expensive than the new node then that node
  can be pruned out (because the new node can provide everything
  that the old one provided (and more) at a lesser ii_cost).
```

```
  the desicion of whether the physical properties of a node are
  INTERESTING or not is left to the DBI (database implementor)
  through the use of the Aphynode_t::IsInteresting member funtion.

  in such a situation we call the new node a REPLACER (because
  it is going to remove/KILL the older node from the search tree) and
  the new node a REPLACEE.

  the situation is reversed if the physical properties if the new
  node are NOT INTERESTING. in that case if there is any already
  existing node which has the same logical properties (obviously
  it has INTERESTING physical properties) and is less expensive than
  the new node, then the new node need not be kept around for the
  later phase of optimization. hence the new node commits SUICIDE.

**************************************************************************/

class Ahashnode_t {
friend class Ahashtable_t;
friend class Ahashid_t;
private:
  Aphynode_t *node;
  Alogprop_t *logprops;
  Aphyprop_t *phyprops;
  Alist_t<Ahashnode_t> *listofreplacers; // nodes which can replace this node.

private: // this is a private class
  Ahashnode_t (Aphynode_t *);
  Ahashnode_t (Alogprop_t *);
  ~Ahashnode_t (void);
  void NewPhyNode (Aphynode_t *);
};

/**************************************************************************

  A note on hashing.

  since the hashing is required to locate nodes with equivalent
  logical properties and physical properties and since these
  classes are supplied by the DBI, the hash function has to
  be implemented by the DBI.

  to do this, we require the DBI to provide us with two functions
  1) the Alogprop_t.Hash () function.
```

```
      the DBI is expected to guarantee that whenever two
      instances of the Alogprop_t class are deemed to be
      equivalent by the Alogprop_t.IsEqualTo function then
      their Alogprop_t.Hash () values must be equal.
      further, the DBI should attempt to write this function such that
      different logprops objects return different values.
   2) the Aphyprop_t.Hash (Alogprop_t *) function.
      here two (logprops,phyprops) should give
      the same hash values if they are equivalent according to the
      Alogprop_t.IsEqualTo and the Aphyprop_t.IsEqualTo
      functions.

   these hashed numbers are rehashed by our hashtable to produce an integer
   from 0 to (arraysize - 1). in case of nodes which are not INTERESTING,
   the Alogprop_t.Hash value is used for rehashing (because the
   physical properties are not interesting). for other nodes the
   Aphyprop_t.Hash value is used for rehashing since the physical
   properties are also important in this case.

*****************************************************************************/

class Ahashid_t {
private:
  Aphynode_t *node;
  Ahashnode_t *exactmatch;
  Alist_t<Ahashnode_t> *replacers;
  Ahashnode_t *replacee;

  int logicalhashnumber; // value generated by Alognode_t.Hash
  int physicalhashnumber; // value returned by the Aphynode_t.Hash

  void FindReplacee (void);

public:
  Ahashid_t (Aphynode_t *newnode);

  static void InitializeHashTable (void);
  static void EmptyHashTable (Ahashtable_t* hashtable);
  Aphynode_t *GetExactMatch (Ahashtable_t* hashtable);
                        // there'll be only one exact match
  Aphynode_t *GetNextReplacer (Ahashtable_t* hashtable);
                       // there will be a list of replacers
  Aphynode_t *GetReplacee (Ahashtable_t* hashtable);
                      // there can be only one replacee
```

```
  void Replace (void);
  void KillReplacee (void);
  void Suicide (void);
  void Insert (Ahashtable_t* hashtable);
};

class Ahashtable_t {
private:
  int arraysize; // number of buckets in the hashtable
  Alist_t<Ahashnode_t> *hasharray; // this is an array of lists
                  // each element is a list of Ahashnode_ts
  static int abs (int x) {return x < 0 ? -x : x;}
  int Rehash (int logicalorphysicalnumber) {
    return abs (logicalorphysicalnumber) % arraysize;
  }

public:
  int end_of_replacers;
public:
  Ahashtable_t (int thearraysize = 67);
  ~Ahashtable_t () { if (hasharray) delete [] hasharray; }
  Ahashnode_t *FindMatch (Aphynode_t *, int hashnumber);
        // in this, for INTERESTING nodes the hashnumber
        // should be the value returned by the Aphyprop_t.Hash function
        // for the non-INTERESTING nodes the Alogprop_t.Hash value should be used.

  void Insert (Ahashnode_t *, int hashnumber);
  Ahashnode_t *FindReplacee (Aphynode_t *, int hashnumber);
  void EmptyHashTable (void); // call between queries.
};
#endif /* AHASH_H */
```

B.4 TreeVisitor.h

```
#ifndef TREEVISITOR_H
#define TREEVISITOR_H

#include <optdef.h>
#include <vector>
#include <Aquery.h>

class TransformTreeGenerator;
```

```
class SearchSpace;

//This class just forms a complete hierarchy and provides get/set methods
//to common attributes.
class TreeToTreeVisitor {
public:
  TreeToTreeVisitor(OperatorTree* tree = 0, SearchSpace* space = 0);
  virtual ~TreeToTreeVisitor();

  OperatorTree* GetCurrentTree() {
    return this->currentTree;
  }
  void SetCurrentTree(OperatorTree* tree) {
    this->currentTree = tree;
  }

  SearchSpace* GetSearchSpace() {
    return this->currentSpace;
  }
  void SetSearchSpace(SearchSpace* space) {
    this->currentSpace = currentSpace;
  }

protected:
  OperatorTree* currentTree;
  SearchSpace* currentSpace;
};

class ExpandTreeVisitor: public TreeToTreeVisitor {
public:
  ExpandTreeVisitor(OperatorTree* tree=0, SearchSpace* space=0);
  virtual ~ExpandTreeVisitor();
  virtual void VisitDBRelation(DBRelation* op);
  virtual void VisitMaterialization(Materialization* op);
  virtual void VisitMaterializationCollapse(MaterializationCollapse* op);
  virtual void VisitSelect(Select* op);
  virtual void VisitSelectCollapse(SelectCollapse* op);
  virtual void VisitIndexCollapse(IndexCollapse* op);
  virtual void VisitJoin(Join* op);
  virtual void VisitUnnest(Unnest* op);
  virtual void VisitOutput(Output* op);
  virtual void VisitOrder(Aorder_t* op); //added by wj:
  virtual void VisitSubquery(Asubquery_t* op);
};
```

```
class JoinTreeVisitor: public TreeToTreeVisitor {
public:
  JoinTreeVisitor(OperatorTree* tree=0, OperatorTree* otherTree=0,
SearchSpace* space = 0);
  virtual ~JoinTreeVisitor();
  virtual Alist_t<Abinop_t> CloneJoin(Join* op);
  virtual Join* CloneJoin(Join* op, Join_tree *join_tree);

protected:
  OperatorTree* otherTree;
};

class TransformTreeVisitor: public TreeToTreeVisitor {
public:
  TransformTreeVisitor(OperatorTree* tree=0, SearchSpace* space=0);
  virtual ~TransformTreeVisitor();

  // Loop for all rules and apply each of them
  void VisitRules();
 // void VisitSelectPushDown(SelectPushDown* rule);

private:
  vector <TransformTreeGenerator*> rules;
};

class TreeToPlanVisitor {
public:
  TreeToPlanVisitor(OperatorTree* tree = 0, SearchSpace* space = 0);
  virtual ~TreeToPlanVisitor();

  OperatorTree* GetCurrentTree() {
    return this->currentTree;
  }
  void SetCurrentTree(OperatorTree* tree) {
    this->currentTree = tree;
  }
  SearchSpace* GetSearchSpace() {
    return this->currentSpace;
  }
  void SetSearchSpace(SearchSpace* space) {
    this->currentSpace = currentSpace;
  }

  virtual void VisitUnaryAlgorithm(Aunaryalgo_t* algorithm);
```

```
  virtual void VisitBinaryAlgorithm(Abinalgo_t* algorithm);
  virtual void VisitFileScan(Afilescan_t* algorithm);
  virtual void VisitFilter(Afilter_t* algorithm);
  virtual void VisitIndexScan(Aindexscan_t* algorithm);
  virtual void VisitSort(Asort_t* algorithm);

protected:
  OperatorTree* currentTree;
  SearchSpace* currentSpace;
};

class PlanToPlanVisitor {
public:
  PlanToPlanVisitor(AlgorithmTree* tree = 0, SearchSpace* space = 0);
  virtual ~PlanToPlanVisitor();

  AlgorithmTree* GetCurrentTree() {
    return this->currentTree;
  }
  void SetCurrentTree(AlgorithmTree* tree) {
    this->currentTree = tree;
  }

  SearchSpace* GetSearchSpace() {
    return this->currentSpace;
  }
  void SetSearchSpace(SearchSpace* space) {
    this->currentSpace = currentSpace;
  }
  virtual void VisitConstrain();

protected:
  AlgorithmTree* currentTree;
  SearchSpace* currentSpace;
};
#endif
```

B.5 TreeGenerator.h

```
#ifndef TREEGENERATOR_H
#define TREEGENERATOR_H

#include <optdef.h>
```

```
#include <SearchSpace.h>

class Generator
{
protected:
    SearchSpace* space;
public:
    Generator(SearchSpace* space = 0){this->space = space;}
    void SetSearchSpace(SearchSpace* space){this->space = space;}
    SearchSpace* GetSearchSpace(){return this->space;}
};

//
// class ExpandTreeGenerator Hierarchy
//
class ExpandTreeGenerator: public Generator {
public:
  ExpandTreeGenerator(SearchSpace* space = 0):Generator(space){}
  virtual ~ExpandTreeGenerator(){}

  // know type of operator in advance
  virtual void Apply(DBOperator* op, OperatorTree *input);
  // do not know type of operator in advance
  virtual void Apply(DBOperator* op, OperatorTree *input, OperatorTree*
&output);
  virtual Alist_t<Aop_t> Clones(DBOperator* op, OperatorTree **input=0);
};

class UnaryOperatorExpand: public ExpandTreeGenerator {
public:
  UnaryOperatorExpand(SearchSpace* space = 0):ExpandTreeGenerator(space){}
  virtual ~UnaryOperatorExpand(){}
 // know type of operator in advance
  virtual void Apply(DBUnaryOperator* op, OperatorTree *input);
  // do not know type of operator in advance
  virtual void Apply(DBOperator* op, OperatorTree *input, OperatorTree*
&output);
  virtual Alist_t<Aunaryop_t> Clones (DBUnaryOperator* op, Alognode_t *input);
};

class BinaryOperatorExpand: public ExpandTreeGenerator {
public:
    BinaryOperatorExpand(SearchSpace* space = 0):ExpandTreeGenerator(space){}
  virtual ~BinaryOperatorExpand(){}
```

```
  // know type of operator in advance
  virtual void Apply(DBBinaryOperator* op, OperatorTree *input);
  // do not know type of operator in advance
  virtual void Apply(DBBinaryOperator* op, OperatorTree *input, OperatorTree*
&output);
protected:
  virtual void DfsNode (DBBinaryOperator* op, OperatorTree *input, OperatorTree
*othernode);
  virtual void DfsNode (DBBinaryOperator* op, OperatorTree *input, OperatorTree
*othernode,
                        OperatorTree* &output);
  virtual Alist_t<Abinop_t> Clones (DBBinaryOperator* op, Alognode_t
*leftinput, Alognode_t *rightinput);
};

class DBRelationExpand: public ExpandTreeGenerator {
public:
    DBRelationExpand(SearchSpace* space = 0):ExpandTreeGenerator(space){}
  virtual ~DBRelationExpand(){}
  virtual void Apply(DBRelation* op, OperatorTree* input);
  virtual void Apply(DBRelation* op, OperatorTree* input, OperatorTree*
&output);
  virtual void compute_operations (DBRelation* op);
    virtual Alist_t<Aop_t> Clones (DBRelation* op, Alognode_t **inputs = 0);
};

class MaterializationExpand: public UnaryOperatorExpand {
public:
    MaterializationExpand(SearchSpace* space = 0):UnaryOperatorExpand(space){}
  virtual ~MaterializationExpand(){}
  //virtual void Apply(Amat_t* op, OperatorTree* input);
  //virtual void Apply(Amat_t* op, OperatorTree* input, OperatorTree* &output);
    virtual void compute_operations (Amat_t* op);
    virtual Alist_t<Aunaryop_t> Clones (Amat_t* op, Alognode_t *inputs);
};

class MaterializationCollapseExpand: public UnaryOperatorExpand {
public:
    MaterializationCollapseExpand(SearchSpace* space =
0):UnaryOperatorExpand(space){}
  virtual ~MaterializationCollapseExpand(){}
  virtual void Apply(MaterializationCollapse* op, OperatorTree* input);
```

```
  virtual void Apply(MaterializationCollapse* op, OperatorTree* input,
OperatorTree* &output);
};

class SelectExpand: public UnaryOperatorExpand {
public:
  SelectExpand(SearchSpace* space = 0):UnaryOperatorExpand(space){}
  virtual ~SelectExpand(){}
  //virtual void Apply (Aselect_t* op, OperatorTree *input);
  //virtual void Apply (Aselect_t* op, OperatorTree *input, OperatorTree*
&output);
  virtual Alist_t<Aunaryop_t> Clones (DBUnaryOperator* op, Alognode_t *input);
};

class SelectCollapseExpand: public UnaryOperatorExpand {
public:
  SelectCollapseExpand(SearchSpace* space = 0):UnaryOperatorExpand(space){}
  virtual ~SelectCollapseExpand(){}
  virtual void Apply (SelectCollapse* op, OperatorTree *input);
  virtual void Apply (SelectCollapse* op, OperatorTree *input, OperatorTree*
&output);
};

class IndexCollapseExpand: public ExpandTreeGenerator {
public:
  IndexCollapseExpand(SearchSpace* space = 0):ExpandTreeGenerator(space){}
  virtual ~IndexCollapseExpand(){}
  virtual void Apply (IndexCollapse* op, OperatorTree *input);
  virtual void Apply (IndexCollapse* op, OperatorTree *input, OperatorTree*
&output);
};

class JoinExpand: public BinaryOperatorExpand {
public:
  JoinExpand(SearchSpace* space = 0):BinaryOperatorExpand(space){}
  virtual ~JoinExpand(){}
  virtual void Apply (Join* op, OperatorTree *input);
  virtual void Apply (Join* op, OperatorTree *input, OperatorTree* &output);
  virtual Alist_t<Abinop_t> Clones (Join* op, Alognode_t *leftinput, Alognode_t
*rightinput);
  virtual Ajoin_t * Clones(Join* op, Join_tree *join_tree,
                           Alognode_t *leftinput, Alognode_t *rightinput);
private:
  void DfsNode (Join* op, OperatorTree *input, OperatorTree *othernode);
```

```
  void DfsNode (Join* op, OperatorTree *input, OperatorTree *othernode,
OperatorTree* &output);
};

class UnnestExpand: public UnaryOperatorExpand {
public:
    UnnestExpand(SearchSpace* space = 0):UnaryOperatorExpand(space){}
};

class OutputExpand: public UnaryOperatorExpand {
public:
    OutputExpand(SearchSpace* space = 0):UnaryOperatorExpand(space){}
};

class SortExpand: public UnaryOperatorExpand {
    // Do something here. if the input aleardy sorted in a required attr,
    //just take the exiting physical plan as my physical plan.
    // Otherwise, we have to do an explicit order.
public:
    SortExpand(SearchSpace* space = 0):UnaryOperatorExpand(space){}
    virtual void Apply (Aorder_t* op, OperatorTree *input);
    virtual Alist_t<Aunaryop_t> Clones (Aorder_t* op, Alognode_t *input);
};

class SubqueryExpand: public UnaryOperatorExpand {
    // Do something here. if the input aleardy sorted in a required attr,
    //just take the exiting physical plan as my physical plan.
    // Otherwise, we have to do an explicit order.
public:
    SubqueryExpand(SearchSpace* space = 0):UnaryOperatorExpand(space){}
    virtual Alist_t<Aunaryop_t> Clones (Asubquery_t* op, Alognode_t *input);
    static Alist_t<Aunaryop_t> FindAlias (Asubquery_t* op, Aquery_t* query, char
*rel_var);
};

//
// class AlgorithmTreeGenerator Hierarchy
//
class AlgorithmTreeGenerator : public Generator{
public:
    AlgorithmTreeGenerator(SearchSpace* space = 0):Generator(space){}
  virtual int CanBeApplied (OperatorTree *, AlgorithmTree **inputs = 0);
            // returns true if this algo can be applied to these inputs.
```

```
  virtual void Apply (DBAlgorithm* algo, OperatorTree *lognode, AlgorithmTree
**inputs);
        // this function should construct all the physical nodes
        // of the given logical node which can result from the
        // application of this algorithm to the given physical nodes.
        // the DBI will usually NOT need to redefine this function
  virtual void MakePhyNodes (DBAlgorithm* algo, OperatorTree *) = 0;
};

class UnaryAlgorithmTreeGenerator: public AlgorithmTreeGenerator {
public:
    UnaryAlgorithmTreeGenerator(SearchSpace* space =
0):AlgorithmTreeGenerator(space){}
  virtual int CanBeApplied (OperatorTree *, AlgorithmTree *input);
           // returns true if this algo can be applied to these inputs.
  virtual void Apply (DBAlgorithm* algo,OperatorTree *, AlgorithmTree *input);
        // DBI will usually NOT need to redefine this function
  virtual void MakePhyNodes (DBAlgorithm* algo, OperatorTree *);
        // DBI will usually NOT need to redefine this function
};

class BinaryAlgorithmTreeGenerator: public AlgorithmTreeGenerator {
public:
    BinaryAlgorithmTreeGenerator(SearchSpace* space =
0):AlgorithmTreeGenerator(space){}
  virtual int CanBeApplied (OperatorTree *, AlgorithmTree *leftinput,
              AlgorithmTree *rightinput);
           // returns true if this algo can be applied to these inputs.
  virtual void Apply (DBAlgorithm* algo, OperatorTree *,
             AlgorithmTree *leftinput,
             AlgorithmTree *rightinput);
               // DBI will usually NOT need to redefine this function
  virtual void MakePhyNodes (DBAlgorithm* algo, OperatorTree *);
               // DBI will usually NOT need to redefine this function
};

class FileScanPlan: public AlgorithmTreeGenerator {
public:
  FileScanPlan(SearchSpace* space = 0):AlgorithmTreeGenerator(space){}
  virtual void MakePhyNodes (DBAlgorithm* algo, OperatorTree *node);
};

class EnforcerPlan: public UnaryAlgorithmTreeGenerator {
```

```
public:
    EnforcerPlan(SearchSpace* space = 0):UnaryAlgorithmTreeGenerator(space){}
};

class AssemblyPlan: public EnforcerPlan {
public:
    AssemblyPlan(SearchSpace* space = 0):EnforcerPlan(space){}
    Alist_t<Aphynode_t> Enforce (Aphynode_t *node, Aphyprop_t *reqd_props);
};

class FilterPlan: public UnaryAlgorithmTreeGenerator {
public:
    FilterPlan(SearchSpace* space = 0):UnaryAlgorithmTreeGenerator(space){}
  int CanBeApplied (OperatorTree *node, AlgorithmTree *input);
  void MakePhyNodes (DBAlgorithm* algo,OperatorTree *lognode);
  void Apply(DBAlgorithm* algo, OperatorTree *lognode, AlgorithmTree *input);
};

class IndexScanPlan: public AlgorithmTreeGenerator {
public:
    IndexScanPlan(SearchSpace* space = 0):AlgorithmTreeGenerator(space){}
  virtual void MakePhyNodes (DBAlgorithm* algo, OperatorTree *node);
};

class HybridHashJoinPlan: public BinaryAlgorithmTreeGenerator {
public:
    HybridHashJoinPlan(SearchSpace* space =
0):BinaryAlgorithmTreeGenerator(space){}
  int CanBeApplied (OperatorTree *node, AlgorithmTree *leftinput,
            AlgorithmTree *rightinput);
};

class PointerHashJoinPlan: public BinaryAlgorithmTreeGenerator {
public:
    PointerHashJoinPlan(SearchSpace* space =
0):BinaryAlgorithmTreeGenerator(space){}
  int CanBeApplied (OperatorTree *node, AlgorithmTree *leftinput,
            AlgorithmTree *rightinput);
};

class UnnestAlgorithmPlan: public UnaryAlgorithmTreeGenerator {
public:
    UnnestAlgorithmPlan(SearchSpace* space =
0):UnaryAlgorithmTreeGenerator(space){}
```

```
  int CanBeApplied (OperatorTree *node, AlgorithmTree *input);
};

class OutputAlgorithmPlan: public UnaryAlgorithmTreeGenerator {
public:
    OutputAlgorithmPlan(SearchSpace* space =
0):UnaryAlgorithmTreeGenerator(space){}
  int CanBeApplied (OperatorTree *node, AlgorithmTree *input);
};

class SortPlan: public UnaryAlgorithmTreeGenerator {
public:
    SortPlan(SearchSpace* space = 0):UnaryAlgorithmTreeGenerator(space){}
  virtual void MakePhyNodes (Asort_t* algo, OperatorTree *node);
};

//
// class TransformTreeGenerator Hierarchy
//
class TransformTreeVisitor;

class TransformTreeGenerator : public Generator{
public:
  TransformTreeGenerator(SearchSpace* space = 0):Generator(space){}
  virtual bool CanBeApplied(OperatorTree* tree);
  virtual void Apply(TransformTreeVisitor& visitor);
};

class SelectPushDown: public TransformTreeGenerator {
public:
  SelectPushDown(SearchSpace* space = 0):TransformTreeGenerator(space){}
  virtual bool CanBeApplied(OperatorTree* tree);
  virtual void Apply(TransformTreeVisitor& visitor);
};
#endif
```

C. Algebra Component

C.1 Alognode.h

```
#ifndef ALOGNODE_H
#define ALOGNODE_H
```

```
#include <Alist.h>
#include <Aop.h>

class Aphynode_t; // forward reference
class Alogprop_t;
class ExpandTreeVisitor;
class TreeToPlanVisitor;
class SearchSpace;
extern void printTree(Alognode_t*node);

/*****************************************************************************
  LOGICAL NODES.

  a search tree is made up of logical nodes and physical nodes.

  A logical node represents a part of an operator tree which can be
  used in the evaluation of the given query.
  the logical node has a pointer the operator which is being applied,
  and the lognodes which serve as inputs to it.

  There is also a pointer to Alogprop_t of the output
  of the partial operator tree associated with the logical node.
  the Alogprop_t class is supplied by the DBI.

  a logical node also has pointers to all the physical nodes
  associated with this logical node. i.e. each physical node
  represents a different possible physical implementation which
  can be used to evaluate the partial operator tree associated
  with this logical node.

  Also there is a list of children of this logical node.
  Children are those logical nodes which result from the application
  of some Operator::Apply to this logical node.
*****************************************************************************/
class Alognode_t {
private:
  Aop_t *op;
  Alogprop_t *logprops;
  Alognode_t **inputs; // array of input Alognode_ts
  Alist_t<Aphynode_t> phynodes; // list of physical nodes
  Alist_t<Alognode_t> children; // logical nodes generated from this one
  Alist_t<Aphynode_t> suboptimal_phynodes;
                // see comment in Aphynode.h
```

```
  Alist_t<Aphynode_t> enforcednodes;
                // list of nodes created by enforcers
                // being applied to phynodes of this node
  Alist_t<Alognode_t> dependent_nodes;
                // list of log nodes that MUST be deleted
                // when this node is deleted. i.e. nodes
                // that use this node as an input and
                // will have to be forcefully deleted if
                // this node is suboptimal.
                // Note: in general, if this node is suboptimal
                // nodes that use this node as an input
                // will get automatically deleted. but this is
                // not true if they are interesting. in this
                // case we have to delete them forcefully...
      .
  int done; // a flag indicating whether this logical node is in
        // the process of being created or it is completed.
        // this is used to decide whether a logical node is to be
        // deleted or not when the number of physical nodes becomes 0.
 // Alognode_t * parent;    // the lognode that takes me as an input.
           // wj: in course of adding an enforcer, we need to konw the
          // phy node(to be enforced)'s parent's parent.

  SearchSpace* space; //the searchsapce where the tree is built.

public:
  int expanded; // true if this node has already been expanded.
  int my_postion; // for print only
  int num_of_printed_children;// for print only
  int print_all;// for print only

public:
  int IsDone (void) const {return done;}
  ~Alognode_t (void);
  //To get rid of global variables, we need to know in which search space
  //a new node is constructed. Also the property class needs to know
  //the number of operations of the current query in the search space
  //to initialize the Aset_t attributes.
  Alognode_t (Aop_t *, SearchSpace* space);
  Alognode_t (Aunaryop_t *, Alognode_t *input, SearchSpace* space);
  Alognode_t (Abinop_t *, Alognode_t *leftinput,
          Alognode_t *rightinput, SearchSpace* space);
```

```
Aop_t *GetOp (void) const {return op;}
Alogprop_t *GetLogProps (void) const {return logprops;}
void SetLogProps (Alogprop_t *l) {logprops = l;}
SearchSpace* GetSearchSpace() {return this->space;}
  void SetSearchSpace(SearchSpace* space) {this->space = space;}

int CanRemoveFromSearch (void);
              // TRUE if this lognode can be removed
              // from the search tree. i.e. it is
              // suboptimal
int CanDelete (void); // TRUE if this lognode can be deleted safely Children()

Alist_t<Alognode_t> &Children (void) {return children;}
void AddChild (Alognode_t *newchild) {children.Insert (newchild);}
void DeleteChild (Alognode_t *child) {children.FindAndDelete (child);}
void AddDependent (Alognode_t *node) {dependent_nodes.Insert (node);}
void DeleteDependent (Alognode_t *n) {dependent_nodes.FindAndDelete (n);}

Alist_t<Aphynode_t> &GetPhyNodes (void) {return phynodes;}
int NumPhyNodes (void) {return phynodes.Length ();}
void AddPhyNode (Aphynode_t *node) {phynodes.Insert (node);}
void DeletePhyNode (Aphynode_t *node);
void DeleteSubOptimalPhyNode (Aphynode_t *node);
void AddEnforcedNode (Aphynode_t *node) {enforcednodes.Insert (node);}

Alognode_t *Input (int N = 0) const {return inputs[N];} // get the Nth input
Alognode_t *LeftInput (void) const {return inputs[0];}  // special cases
Alognode_t *RightInput (void) const {return inputs[1];} // of the above
Alist_t<Aphynode_t> &MakePhyNodes (void);
int Contain(Alognode_t *other_log_node);   // added by wj.
               //To judge if a tree contain another tree.
              //Called in Dynamic Programming to prevent a tree
              //to make Cartesian product with a node contained by itsefl.

//Alognode_t * GetParet(void){return parent;}

//methods moved from SearchTree.c
void ADeleteTree (void);
void DeleteAllSubOptimalPhyPlan(void);
double GetCheapestCostOfLognode(void);
// added by wj: for test
void print(void){reset_print_info();printTree(this);}// for print only
void print_phy_tree(void);
void print_node(void);// for print only
```

```
  int nargs(void){return op->Arity();}// for print only
  Alognode_t* arg(int n){return Input(n);}// for print only
  void reset_print_info(void)
  {
      my_postion=0; //
      num_of_printed_children=0;
      print_all=0;
      for (int i=0;i<nargs();i++)
      {
          arg(i)->reset_print_info();
      }
  }
};
#endif /* ALOGNODE_H */
```

C.2 Alogprop.h

```
#ifndef LOGPROP_H
#define LOGPROP_H

#include <Aset.h>
#include <Arel.h>
#include <Aattr.h>
#include <Aptree.h>
#include <Apred.h>
#include <Aopdefs.h>

class Alognode_t;
class SearchSpace;

class Alogprop_t {
    friend class Aphyprop_t; //wj: i need to set _is_interesting and
private:
  Aset_t<Aptree_t>* _operations;   // operations applied so far
  Aset_t<Aptree_t>* _undone_tuplerefs;
                // keeps track of any pointer join
                // predicates which should have been
                // applied but haven't yet been...
  Aset_t<Aptree_t>* _need_unnesting;
                // keeps track of which attributes
                // are set valued and havent yet been unnested.

  double _numtuples;        // number of tuples
```

```
  int outputop_applied;     // true if output operator has been applied.
                // these two variables are used in the
                // IsEqualTo function. they are very important
  char *_index_path;        // if this tree is part of a
                // get-(mat)*-select combination which can
                // be used for an indexscan then this variable
                // holds the pathname materialized so far...
  int _is_interesting;      // is this an interesting lognode?

  Alogprop_t (void) {
    _operations = _undone_tuplerefs = _need_unnesting = 0;
  }
  SearchSpace* space; //just for convenience, actually can get from the lognode.

 public:
  ~Alogprop_t (void) {if(_index_path) delete _index_path;}
  Alogprop_t (Aget_t *, Alognode_t *);
  Alogprop_t (Amat_t *, Alognode_t *);
  Alogprop_t (Amat_collapse_t *, Alognode_t *);
  Alogprop_t (Aselect_t *, Alognode_t *);
  Alogprop_t (Aselect_collapse_t *, Alognode_t *);
  Alogprop_t (Aidx_collapse_t *, Alognode_t *);
  Alogprop_t (Ajoin_t *, Alognode_t *);
  Alogprop_t (Aunnestop_t *, Alognode_t *);
  Alogprop_t (Aoutputop_t *, Alognode_t *);
  Alogprop_t (Aorder_t *, Alognode_t *); // by wj
  Alogprop_t (Asubquery_t *, Alognode_t *); // by wj

  Alogprop_t *Duplicate (void) const;

  Aset_t<Aptree_t> &operations (void) const {return *_operations;}
  const Aset_t<Aptree_t> &undone_tuplerefs (void) const;
  const Aset_t<Aptree_t> &need_unnesting (void) const;

  double numtuples (void) const {return _numtuples;}
  const char *index_path (void) const {return _index_path;}

  int IsEqualTo (const Alogprop_t *other) const;
  int IsInteresting (void) const;
  int NumOperations (void) const;
  int IsCompleteQuery (void);
  int Hash (void) const;
};
```

```
/*****************************************************************************

  inline functions

*****************************************************************************/
#include <Aoptions.h>

inline const Aset_t<Aptree_t> &Alogprop_t::undone_tuplerefs (void) const
{
  return *_undone_tuplerefs;
}
inline const Aset_t<Aptree_t> &Alogprop_t::need_unnesting (void) const
{
  return *_need_unnesting;
}
inline int Alogprop_t::IsCompleteQuery (void)
{
  return _operations->IsFull () && _need_unnesting->IsEmpty ();
}
inline int Alogprop_t::NumOperations (void) const {
  return space->oopt->dont_split_lists ? 0 : _operations->Cardinality ();
}
#endif /* LOGPROP_H */
```

C.3 Aphynode.h

```
#ifndef APHYNODE_H
#define APHYNODE_H

#include <Alist.h>
#include <Aalgo.h>
#include <Alognode.h>
#include <Aopalgos.h>

class Aphyprop_t; // forward reference
void printTree(Aphynode_t*node);

/*****************************************************************************

  PHYNODE
```

```
  there can be a number of phynodes associated with each logical
  node in a search tree. each Aphynode_t is a representation of a
  different way of physically computing the same operation (operator tree)
  described in the given ALogNode. these differences arise due to
  different algorithms existing for computing the operators.

  SUBOPTIMAL NODES

  consider a node (n1) which has been used as an input node by
  another node (n2). now, if a third node (n3) is created and it
  happens to be equivalent to node n1 and also happens to be cheaper
  than n1, then we would delete n1. but if, in this situation, it so
  happens that miraculously n2 turns out to be part of the ultimate
  optimal plan (unlikely, but possible in the current scheme) then we
  are in trouble. because n2 will try to access n1, but n1 has already
  been deleted.

  currently i see no easy way out of this problem. so this is what i
  do: each node has a usedflag variable which will be zero if and only
  if a particular node is not used as an input by any other node. and
  when we are trying to delete a node we check if usedflag is 0. if it
  is not we dont delete it. we just remove it from the hashtable and the
  search tree.

*****************************************************************************/

class Aphynode_t {
private:
  Aalgo_t *algorithm;
  Aphynode_t **inputs; // array of pointers to input Aphynode_ts
  Aphyprop_t *phyprops;
  Alognode_t *parent;
  int usedflag; // this flag is true if this node has been used by
        // some other node as an input. in this case we should
        // not delete this node. we just mark it as suboptimal.
  int suboptimal; // this marks nodes which are suboptimal but cannot be
          // deleted because they are used as inputs to other nodes.
          // these nodes should not considered as inputs for other
          // nodes
  int enforcednode; // true if this node is the result of the application
            // of an enforcer to another. such nodes have to be
            // handled differently in some cases.
  Alist_t<Aphynode_t> init_plan_list; // wj: a list of pointers to the subplans
              //generated by subqueries in where-clauses; these plans have to
```

```
            // be executed first before the current plan is executed!

public:
  int my_postion; // for print only
  int num_of_printed_children;// for print only
  int print_all;// for print only

  ~Aphynode_t (void);
  static void DeletePhyNode (Aphynode_t *node);
  Aphynode_t (Alognode_t *parent, Aunaryalgo_t *, Aphynode_t *input);
  Aphynode_t (Alognode_t *parent, Abinalgo_t *,
          Aphynode_t *leftinput, Aphynode_t *rightinput);
  Aphynode_t (Alognode_t *parent, Aalgo_t *, Aphynode_t **inputs = 0);
  Aphynode_t (Alognode_t *parent, Aenforcer_t *, Aphynode_t *input);

  Aalgo_t *GetAlgo (void) const {return algorithm;}
  int GetNumInputs (void) const {return algorithm->Arity ();}
  void SetPhyProps (Aphyprop_t *p) {phyprops = p;}

  Aphynode_t *Input (int N = 0) const {return inputs[N];} // get the Nth input
  Aphynode_t *LeftInput (void) const {return inputs[0];}  // special cases
  Aphynode_t *RightInput (void) const {return inputs[1];} // for binary nodes

  Aphyprop_t *GetPhyProps (void) const {return phyprops;}
  Alogprop_t *GetLogProps (void) const {return parent->GetLogProps ();}
  Alognode_t *GetParent (void) const {return parent;}

  int IsInteresting (void) const; // is this physical node interesting.
  int SubOptimal (void) const {return suboptimal;}
        // true if this is a suboptimal node.
  int IsUsed (void) const {return usedflag;}
              // not for general use.
              // used only by Aalgo_t::Apply
  int IsEnforcedNode (void) const {return enforcednode;}
  void InsertIninPlan(Aphynode_t* node){init_plan_list.Insert(node);}//  wj
  Alist_t<Aphynode_t> GetInitPlan(void){ return init_plan_list;}

  void print(void){reset_print_info(); printTree(this);}
  void print_node(void)
  {
      cout<<algorithm->GetName();
      cout<<"  (COST="<<this->GetPhyProps()->GetCost().GetCost()<<")";
      Aop_t *op=this->GetParent()->GetOp();
```

```
        if(op->GetNumber()==Aget)
        {
            Aget_t *get=(Aget_t *)op;
            cout<<"{"<<get->rel()->name()<<"}";
        }
        cout<<endl;
    }// for print only
    int nargs(void){return algorithm->Arity();}// for print only
    Aphynode_t* arg(int n){return Input(n);}// for print only
    void reset_print_info(void)
    {
        my_postion=0; //
        num_of_printed_children=0;
        print_all=0;
        for (int i=0;i<nargs();i++)
        {
            arg(i)->reset_print_info();
        }
    }
};

#endif /* APHYNODE_H */
```

C.4 Aphyprop.h

```
#ifndef APHYPROP_H
#define APHYPROP_H

#include <Aopdefs.h>
#include <Aattr.h>
#include <Acost.h>
#include <Aptree.h>

class Alogprop_t;
class Aphynode_t;

class Aphyprop_t {
private:
  Acost_t _cost;
  Aset_t<Aptree_t>* _ops_not_in_memory;
                // operations which the Alogprops thinks
                // are completed but which are not in
                // memory (due to idx_collapse)
```

```
                // these will have to be brought in by
                // the assembly enforcer.
  long int _inmem_ass_obj_size;// size of assembled obj in memory
  char* _required_sort_path;
  int _is_interesting; // added by wj. if a phy node is interesting,
        //its log node must be interesting. However, its brother phy nodes are
        //not necessarily interesting, and could be deleted. Therefore,
        //I have to mark if a phynode is interesting!
  Alist_t<char> _sort_order;
        // wj: i have to record the sort-order as results of this phynode!

public:
  ~Aphyprop_t (void) {
    if(_required_sort_path)
        delete [] _required_sort_path;
  }
  Aphyprop_t (Aphyprop_t *other) {*this = *other;_required_sort_path=0;}

  Aphyprop_t (Afilescan_t *, Aphynode_t *);
  Aphyprop_t (Aindexscan_t *, Aphynode_t *);
  Aphyprop_t (Aassembly_t *, Aphynode_t *);
  Aphyprop_t (Afilter_t *, Aphynode_t *);
  Aphyprop_t (Ahh_join_t *, Aphynode_t *);
  Aphyprop_t (Ahhptr_join_t *, Aphynode_t *);
  Aphyprop_t (Aunnestalgo_t *, Aphynode_t *);
  Aphyprop_t (Aoutputalgo_t *, Aphynode_t *);
  Aphyprop_t (Asort_t *, Aphynode_t *);    // wj
  Aphyprop_t (Amerge_join_t *, Aphynode_t *);  // wj
  Aphyprop_t (Anested_loop_t *, Aphynode_t *); // wj
  Aphyprop_t (Asubplan_t *, Aphynode_t *); // wj

  Aset_t<Aptree_t> &ops_not_in_memory (void) const {return
*_ops_not_in_memory;}
  long int inmem_ass_obj_size (void) const {return _inmem_ass_obj_size;}
  void need_inmem (Aset_t<Aptree_t> &need_inmem);
  Acost_t GetCost () const {return _cost;}
  int IsEqualTo (const Aphyprop_t *other) const;
  int Hash (const Alogprop_t *logprop) const;
  int IsInteresting (const Alogprop_t *logprop) const;
  void SetRequiredSortOrder(const char *rel_name,const char *attr_name){
      _required_sort_path=new char[strlen(rel_name)+strlen(attr_name)+2];
      strcpy(_required_sort_path,rel_name);
      strcat(_required_sort_path,".");
      strcat(_required_sort_path,attr_name);
```

```
}
  char * GetRequiredSortOrder(void) {return _required_sort_path;}
  Alist_t<char> GetSortOrderList(void){return _sort_order;}
};

/*****************************************************************************
  inline functions
*****************************************************************************/
inline void Aphyprop_t::need_inmem (Aset_t<Aptree_t> &need_inmem)
{
  _ops_not_in_memory->Minus (need_inmem);
}
#endif /* APHYPROP_H */
```

C.5 Aop.h

```
#ifndef AOP_H
#define AOP_H

#include <Aopdefs.h> // supplied by DBI
#include <stdlib.h>
#include <Alist.h>

class Alogprop_t; // to take care of
class Aalgo_t;      // the forward
class Alognode_t;       // references.
class ExpandTreeVisitor;

/*****************************************************************************
  THE OPERATOR BASE CLASS

  from this base class will be derived classes for all the operators
  in the algebra.
  we have derived classes Aunaryop_t and Abinop_t
  to factor out common code for these classes of operators.

  associated with each operator we have a listofalgorithms which
  can implement this operator.
  everytime a new algorithm is defined it is expected to call
  the AddAlgorithm function and add itself to the list of algorithms
  which implement that operator.
*****************************************************************************/
```

```
class Aop_t {
private:
  char *name;   // name of the operator. DBI-supplied.
  AopNumber number; // a unique number associated with this operator.
                // we believe that this function will be useful
                // for writing code where the DBI wants to do
                // a switch based on operator "type"
protected:
  Alist_t<Aalgo_t> listofalgorithms; // this is a list of instances
                     // of the algorithms which can
                     // be used to implement this operator.
public:
  virtual ~Aop_t (void) {}
  Aop_t (char *newname, AopNumber n, Alist_t<Aalgo_t> l) {
    name = newname; number = n; listofalgorithms = l;
  }
  virtual void Accept(ExpandTreeVisitor& visitor) = 0;
  virtual Aop_t *Duplicate (void) const = 0;
        // this virtual function should return a newly
        // allocated object which is an exact replica of this one.
        // read comment in file ./README to find out why it is
        // required. the DBI has to provide this function

  char *GetName (void) {return name;}
  int GetNumber (void) {return number;}
  virtual Alist_t<Aalgo_t> GetListOfAlgorithms (void) const
     {return listofalgorithms;}

  virtual int Arity (void) const = 0;
        // e.g. 1 for unary and 2 for binary ops.
  virtual Alogprop_t *MakeLogProps (Alognode_t *) = 0;
        // make the logical properties for the given logical node
        // this just involves a call to the Alogprop_t
        // with the right parameters.
};

class Aunaryop_t : public Aop_t {
private:
public:
  virtual ~Aunaryop_t (void) {}
  Aunaryop_t (char *name, AopNumber n, Alist_t<Aalgo_t> a)
    : Aop_t (name, n, a) {}

  int Arity (void) const {return 1;} // because its a unary operator
```

```
};

class Abinop_t : public Aop_t {
public:
  virtual ~Abinop_t (void) {}
  Abinop_t (char *name, AopNumber n, Alist_t<Aalgo_t> a)
    : Aop_t (name, n, a) {}

  int Arity (void) const {return 2;} // because it a dyadic operator.
};

#endif /* AOP_H */
```

C.6 Aalgo.h

```
#ifndef AALGO_H
#define AALGO_H

#include <Aopdefs.h>        // provided by user

#include <Alist.h>
#include <string.h>
#include <Aop.h>
#include <iostream.h>
#include <stdlib.h>

class TreeToPlanVisitor;
class Aenforcer_t;
class Aphyprop_t;
class Aphynode_t;
class Bexec_info_t;     // forward reference
class Alognode_t;

/*****************************************************************************
  ALGORITHMS

  algorithms are different ways to physically implement a given
  operator. thus, for each logical node, we can have different
  physical implementations (depending on different algorithms)
  giving rise to the same output. This gives rise to to multiple
  Aphynode_ts associated with the same logical node.

  each algorithm has a MakePhyNodes function which supposed to
```

```
  generate all the Aphynode_ts that can possibly arise from
  implementing the operator of a given Alognode_t using that
  algorithm.

  this MakePhyNodes function makes use of an overloaded
  MakePhyNode function to do the job.
*****************************************************************************/

class Aalgo_t {
private:
  char *name;   // name of the algorithm. supplied by DBI.
  AalgoNumber number; // a unique number supplied by DBI.

protected:
  Alist_t<Aenforcer_t> *enforcers; // array of list of enforcers.
                                   // one list for each input.
  int interesting; //for performance,
                  //used by BinaryAlgorithmTreeGenerator::MakePhyNodes
                  //check if interesting when precomputing the cost
                 //before really generate a new algorithm tree
public:
  Aalgo_t (char *newname, AalgoNumber n,
        Alist_t<Aenforcer_t> *e = 0) {
    name = newname; number = n; enforcers = e;
  }

  virtual ~Aalgo_t (void) {}
  virtual Aalgo_t *Duplicate (void) const = 0;
               // this virtual function should return a newly allocated object
               // which is an exact replica of this one.
               // read comment in file ./README to find out why it is required.
               // the DBI has to provide this function

  char *GetName (void) {return name;}
  AalgoNumber GetNumber (void) {return number;}
  int IsInteresting(void) {return interesting;}
  virtual int Arity (void) const = 0;
  //virtual void MakePhyNodes (Alognode_t *) = 0;
  virtual void Accept(TreeToPlanVisitor& visitor) = 0;

  virtual Aphyprop_t *Constraint (Alognode_t *,Aphynode_t *,int inputnumber =
0) {  return 0; }
        // default == 0. unless DBI specifies otherwise
        // this function shoule return the physical properties
```

```
        // required of the given physical node if it needs to be
        // used as the Nth input of this algorithm. should
        // return 0 if the given node already satisfies
        // the constraints or if the node cannot be used as
                // an input to this algorithm at all.
                // DBI will usually redefine this function.

  virtual Alist_t<Aalgo_t> Clones (Alognode_t *, Aphynode_t **inputs = 0);
        // this is supposed to give a list of algorithm objects
        // which represent different ways of applying this algorithm
        // to these inputs, but with different parameters
        // DBI is supposed to redefine this function if s/he
        // doesnt like the default behaviour

  virtual Alist_t<Aphynode_t>
    EnforceNthConstraint (Alognode_t *,
                  Aphynode_t *input, int N = 0);
                // apply the proper enforcer(s) to the Nth input and
                // return a list of resulting phynode(s)
                // (which will satisfy the input constraints).
                // under normal circumstances the DBI should NOT have to
                // redefine this function.
                // the DBI will usually NOT need to redefine this function

  virtual Aphyprop_t *MakePhyProps (Aphynode_t *)=0;
                // make the physical properties for the given physical node.
                // assume that the rest of the members of the physical node
                // have already been filled in...
                // the DBI will have to provide this function

  virtual void Execute (Bexec_info_t &exec_info) = 0;
        // to be used by the DBI for execution of an access
        // plan generated by APG. Bexec_info_t is a DBI supplied
        // class
};

class Aunaryalgo_t : public Aalgo_t {
private:
public:
  Aunaryalgo_t (char *name, AalgoNumber n, Alist_t<Aenforcer_t> *e = 0)
    : Aalgo_t (name, n, e) {}
  virtual ~Aunaryalgo_t (void) {}
  int Arity (void) const {return 1;}
```

```
  virtual void Accept(TreeToPlanVisitor& visitor);
  virtual Alist_t<Aunaryalgo_t> Clones (Alognode_t *, Aphynode_t *input);
        // this is supposed to give a list of algorithm objects
        // which represent different ways of applying this algorithm
        // to this input, but with different parameters
        // DBI is supposed to redefine this function if s/he
        // doesnt like the default behaviour
};

class Aenforcer_t : public Aunaryalgo_t {
private:
public:
  Aenforcer_t (char *name, AalgoNumber n, Alist_t<Aenforcer_t> *e = 0)
    : Aunaryalgo_t (name, n, e) {}
  virtual ~Aenforcer_t (void) {}

  virtual Alist_t<Aphynode_t> Enforce (Aphynode_t *,
                      Aphyprop_t *) = 0;
        // apply this enforcer to the given phynode so that it
        // finally has the given physical properties.
        // the DBI will have to provide this function

        //  virtual void Apply (Alognode_t *, Aphynode_t *input);
        //  an Enforcer does not need an Apply function
        // this is different from Aunaryalgo_t::Apply because
        // the resultant nodes should not be Pruned in case of
        // enforcers DBI will usually NOT need to redefine this
        // function.
};

class Abinalgo_t : public Aalgo_t {
private:
public:
  Abinalgo_t (char *name, AalgoNumber n,
              Alist_t<Aenforcer_t> *e = 0)
    : Aalgo_t (name, n, e) {}
  virtual ~Abinalgo_t (void) {}

  int Arity (void) const {return 2;}
  virtual void Accept(TreeToPlanVisitor& visitor);
  virtual Alist_t<Abinalgo_t> Clones (Alognode_t *,
                        Aphynode_t *leftinput,
                        Aphynode_t *rightinput);
        // this is supposed to give a list of algorithm objects
```

```
        // which represent different ways of applying this algorithm
        // to this input, but with different parameters
        // DBI is supposed to redefine this function if s/he
        // doesnt like the default behaviour
    virtual double precompute (Aphynode_t *left, Aphynode_t *right){}
};
#endif /* AALGO_H */
```

C.7 Acost.h

```
#ifndef ACOST_H
#define ACOST_H

#include <Aopdefs.h>
#include <iostream.h>

class Aphynode_t;

class Acost_t {
private:
  double _cost;
public:
  Acost_t (void) {_cost = -1;}
  ~Acost_t (void) {}

  int operator< (const Acost_t &other) {return _cost < other._cost;}
  int operator<= (const Acost_t &other) {return _cost <= other._cost;}
  int operator> (const Acost_t &other) {return _cost > other._cost;}
  int operator>= (const Acost_t &other) {return _cost >= other._cost;}
  int operator== (const Acost_t &other) {return _cost == other._cost;}
  int operator!= (const Acost_t &other) {return _cost != other._cost;}

  void compute (Afilescan_t *algo, Aphynode_t *node);
  void compute (Aindexscan_t *algo, Aphynode_t *node);
  void compute (Afilter_t *algo, Aphynode_t *node);
  void compute (Aassembly_t *algo, Aphynode_t *node);
  void compute (Ahh_join_t *algo, Aphynode_t *node);
  void compute (Ahhptr_join_t *algo, Aphynode_t *node);
  void compute (Aunnestalgo_t *algo, Aphynode_t *node);
  void compute (Aoutputalgo_t *algo, Aphynode_t *node);
  void compute (Asort_t *algo, Aphynode_t *node);
  void compute (Amerge_join_t *algo, Aphynode_t *node);
  void compute (Anested_loop_t *algo, Aphynode_t *node);
```

```
  //Alternatives to computer a join, but computer the cost
  //before create a new algorithm tree for better performance.
  double Acost_t::precompute (Ahh_join_t *algo, Aphynode_t *left, Aphynode_t
*right);
  double Acost_t::precompute (Amerge_join_t *algo, Aphynode_t *left, Aphynode_t
*right);
  double Acost_t::precompute (Anested_loop_t *algo, Aphynode_t *left,
Aphynode_t *right);

  double  GetCost(void){return _cost;}// added by wj for test
  void write (ostream &os) const;
};

inline ostream &operator<< (ostream &os, const Acost_t &cost)
{
  cost.write (os); return os;
}
#endif /* ACOST_H */
```

C.8 Aopdefs.h

```
#ifndef AOPDEFS_H
#define AOPDEFS_H

#include <Aquery.h>

class Aop_t;
class Aalgo_t;
class Aenforcer_t;

enum AopNumber {
  Aget,
  Amat,             // materialize operator
  Amat_collapse,
  Aselect,
  Aselect_collapse,
  Aidx_collapse,        // collapse to index scan
  Ajoin,
  Aunnestop,
  Aaggrop,          // aggregate operator
  Aoutputop,            // output the result of the query
  Aorder,
```

```
  Asubquery
};

class Aget_t;
class Amat_t;
class Amat_collapse_t;
class Aselect_t;
class Aselect_collapse_t;
class Aidx_collapse_t;
class Ajoin_t;
class Aunnestop_t;
class Aaggrop_t;
class Aoutputop_t;
class Aorder_t;
class Asubquery_t;

enum AalgoNumber {
  Afilescan,
  Aindexscan,
  Afilter,
  Aassembly,
  Ahh_join,             // hybrid hash join
  Ahhptr_join,          // hybrid hash pointer join
  Aunnestalgo,
  Aoutputalgo,          // output the result of the query
  Asort,
  Amerge_join,          // merge join for join
  Anested_loop,
  Asubplan
};

class Afilescan_t;
class Aindexscan_t;
class Aassembly_t;
class Afilter_t;
class Ahh_join_t;
class Ahhptr_join_t;
class Aunnestalgo_t;
class Aoutputalgo_t;
class Asort_t;
class Amerge_join_t;
class Anested_loop_t;
class Asubplan_t;
```

```
#include <Alist.h>

// this class stores the association between the
// operators and the various algorithms used to implement them.
class Aopalgo_t {
public:
  // lists of algorithms associated with various operators
  Alist_t<Aalgo_t> get_algos;
  Alist_t<Aalgo_t> mat_algos;
  Alist_t<Aalgo_t> mat_collapse_algos;
  Alist_t<Aalgo_t> select_algos;
  Alist_t<Aalgo_t> select_collapse_algos;
  Alist_t<Aalgo_t> idx_collapse_algos;
  Alist_t<Aalgo_t> join_algos;
  Alist_t<Aalgo_t> unnest_algos;
  Alist_t<Aalgo_t> output_algos;
  Alist_t<Aalgo_t> order_algos;  //wj:
  Alist_t<Aalgo_t> subquery_algos;  //wj:

  // array of lists of enforcers associated with various algorithms
  Alist_t<Aenforcer_t> *enforcer_array;
  Alist_t<Aenforcer_t> *enforcer_array1;

  // operators and algorithms.
  Aget_t *get;
  Amat_t *mat;
  Amat_collapse_t *mat_collapse;
  Aselect_t *select;
  Aselect_collapse_t *select_collapse;
  Aidx_collapse_t *idx_collapse;
  Ajoin_t *join;
  Aunnestop_t *unnestop;
  Aoutputop_t *outputop;
  Aorder_t *order;  //wj: order-by operator
  Asubquery_t *subquery;

  Afilescan_t *filescan;
  Aassembly_t *assembly;
  Afilter_t *filter;
  Aindexscan_t *indexscan;
  Ahh_join_t *hh_join;
//  Ahhptr_join_t *hhptr_join;
```

```
  Aunnestalgo_t *unnestalgo;
  Aoutputalgo_t *outputalgo;
  Asort_t *sort;    //wj: sort algorithm for order-by operator
  Amerge_join_t *merge_join;    //merge join algorithm
  Anested_loop_t *nested_loop;
  Asubplan_t* subplan;

  Alist_t<Aop_t> all_operators;

public:
  //Aopalgo_t (void);
  Aopalgo_t (Aoptimizeroptions_t* oopt);
  Aopalgo_t (Aoptimizeroptions_t* oopt, Aquery_t* query);
  ~Aopalgo_t (void);
  void CreateDBAlgebra (Aoptimizeroptions_t* oopt);
  void InitDBAlgebraSets(Aquery_t* query);
};
#endif /* AOPDEFS_H */
```

D. Others

D.1 optdef.h

```
#ifndef OPTDEF_H
#define OPTDEF_H
#include <stack.h>

//
//re-definitions for some class names in OPT++
//

class Aop_t;
class Aunaryop_t;
class Abinop_t;
class Aget_t;
class Amat_t;
class Amat_collapse_t;
class Aselect_t;
class Aselect_collapse_t;
class Aidx_collapse_t;
class Ajoin_t;
class Aunnestop_t;
```

```
class Aoutputop_t;
class Aorder_t;
class Asubquery_t;

class Aalgo_t;
class Aunaryalgo_t;
class Abinalgo_t;
class Afilescan_t;
class Aassembly_t;
class Afilter_t;
class Aindexscan_t;
class Ahh_join_t;
class Ahhptr_join_t;
class Aunnestalgo_t;
class Aoutputalgo_t;
class Anested_loop_t;
class Amerge_join_t;
class Asort_t;
class Asubplan_t;

class Alognode_t;
class Aphynode_t;

class Aglob_vars_t;
class Acat_t;
class Ahashtable_t;
class Ahashid_t;
class Bparser_state_t;
class Aquery_t;
class Aopalgo_t;
class Aoptimizeroptions_t;
class Arusage_t;
class Afunc_t;
class Aptree_t;
template <class ListElementType> class Alist_t;
template <class SetElementType> class Aset_t;
class Bquery_stmt_op_t;
class Brqg_t;

typedef Aglob_vars_t        GlobalVariables;
typedef Brqg_t              QueryGenerator;
typedef Bquery_stmt_op_t    QueryStatement;
typedef Aop_t               DBOperator;
typedef Aunaryop_t          DBUnaryOperator;
```

```
typedef Abinop_t             DBBinaryOperator;
typedef Aget_t               DBRelation;
typedef Amat_t               Materialization;
typedef Amat_collapse_t      MaterializationCollapse;
typedef Aselect_t            Select;
typedef Aselect_collapse_t   SelectCollapse;
typedef Aidx_collapse_t      IndexCollapse;
typedef Ajoin_t              Join;
typedef Aunnestop_t          Unnest;
typedef Aoutputop_t          Output;

typedef Aalgo_t              DBAlgorithm;
typedef Aunaryalgo_t         DBUnaryAlgorithm;
typedef Abinalgo_t           DBBinaryAlgorithm;
typedef Afilescan_t          FileScan;
typedef Aassembly_t          Assembly;
typedef Afilter_t            Filter;
typedef Aindexscan_t         IndexScan;
typedef Ahh_join_t           HybridHashJoin;
typedef Ahhptr_join_t        PointerHashJoin;
typedef Aunnestalgo_t        UnnestAlgorithm;
typedef Aoutputalgo_t        OutputAlgorithm;

typedef Alognode_t           OperatorTree;
typedef Aphynode_t           AlgorithmTree;
typedef Alogprop_t           OperatorTreeProperty;
typedef Aphyprop_t           AlgorithmTreeProperty;

typedef Acat_t               Catalog;
typedef Ahashtable_t         HashTable;
typedef Ahashid_t            HashId;
typedef Bparser_state_t      ParserStates;
typedef Aquery_t             Query;
typedef Aopalgo_t            OperatorAndAlgorithm;
typedef Aoptimizeroptions_t  OptimizerOptions;

typedef Arusage_t            ResourceUsage;
typedef Afunc_t              Expression;
typedef Aptree_t             PredicateTree;
#endif
```