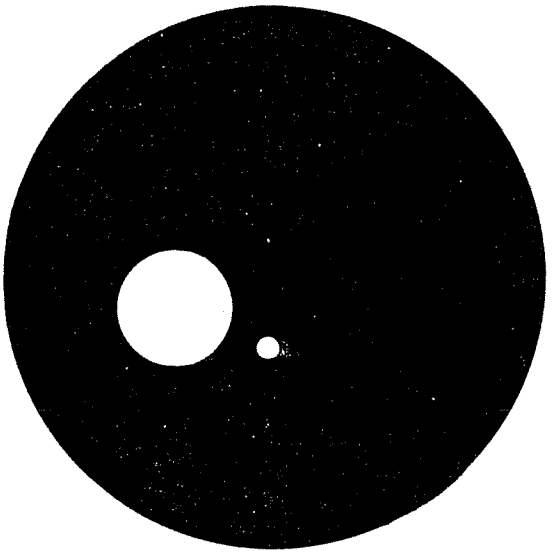
COMPUTER SCIENCES
DEPARTMENT

University of Wisconsin-
Madison



Rule-Based Query Optimization in Extensible Database Systems

by
Goetz Graefe

Computer Sciences Technical Report # 724
November 1987

RULE-BASED QUERY OPTIMIZATION IN EXTENSIBLE DATABASE SYSTEMS

by

GOETZ GRAEFE

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

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1987

Table of Contents

List of Figures

List of Tables

List of Examples

Abstract

This thesis presents the problems of query optimization in extensible database systems and proposes a solution. It describes the design and an initial evaluation of the query optimizer generator developed for the EXODUS extensible database system. The goal of the EXODUS system is to provide software tools and libraries to structure and to ease the task of implementing or extending a database system for a new data model. Our basic model of optimization is to map a query tree, which consists of operators at the nodes and stored data at the leaves, to an access plan, which is a tree with implementation methods at the nodes and scans at the leaves. The optimizer generator translates algebraic equivalence rules into an executable optimizer. The equivalence rules are specific to the data model. The generated optimizer reorders query trees and selects implementation methods according to cost functions associated with the methods. The search strategy of the optimizer avoids exhaustive search by learning from past experience.

We report on two operational optimizers. Experiments with a restricted relational system show that the generated optimizer produces access plans of almost the same anticipated execution cost as those produced by exhaustive search, with the search time cut to a small fraction. Another set of experiments shows that a generated optimizer is able to handle large queries.

An optimizer currently under development for a new query evaluation method shows the power and flexibility of the approach. Other researchers have decided to use the optimizer generator for their database implementation work.

Independently from the EXODUS project, the optimizer generator proved to be a valuable tool for exploring the trade-offs between left-deep execution trees and general execution

trees in relational database systems. Our experiments show that for bushy trees, the higher optimization cost and the cost for creating and reading temporary files can be more than compensated by the reduction in processing cost.

Acknowledgements

It has been my privilege to work and study with Professor David DeWitt during my four years in Wisconsin. He has provided a stimulating and productive research environment in both the GAMMA and the EXODUS projects. His enthusiasm and guidance have made this thesis possible.

Mike Carey has always been willing to answer my questions or to listen to my ideas. I appreciate his encouragement very much.

I would like to thank the other members of my committee: Yannis Ioannidis, Debby Joseph, and John Beetem. Their efforts and suggestions have improved the quality of this thesis. Susan Horwitz has been the first to use my software. Her suggestions regarding my thesis have contributed to this work.

Special thanks are due to Professor G. Stiege, Dr. J. Spiess, and Dr. K. Andresen. Without their help I would not have had the initiative to move from the Technical University Braunschweig to the University of Wisconsin. After I arrived, Bob Holloway went out of his way to give me a good start in this new environment.

The other members of the GAMMA and EXODUS groups have helped me to enjoy the good times and to get through the tough times of graduate studies. Special thanks go to Bob Gerber and to Helen and Joel Richardson. Their friendship makes a big difference.

My family has given me much; their encouragement to go overseas has been important to me. The Matz family in Wisconsin has taken good care of me ever since we learned about our distant relationship.

During the last two summers in Madison, my sailing friends Bill Shelton and Brian Pinkerton helped me to keep a good balance of work and play. Monica Guyette has been a

good friend and running partner. Finally, Barb Peters has been outstandingly caring and supportive during the process of finishing up, even though she has been very busy with her own dissertation. Her encouragement is very much appreciated.

CHAPTER 1

Introduction

1.1. Extensible Database Systems

In recent years, a number of new data models have been proposed in the database literature that extend the modelling and query processing power of the relational model (Codd, 1970). For business style applications, the relational data model has been used very successfully. Services provided by relational database systems have significantly enhanced programmer and data processing productivity. Such services include high level query and data manipulation languages, transparent maintenance of secondary search structures, data independence, data sharing, control of access privileges, concurrency control, and recovery from program, media, and system failures. While the relation-tuple paradigm with fixed-format records provides sufficient modelling flexibility for most record-keeping applications, it is not well-suited for other database application areas which are starting to emerge. Consider, for example, the database needs of an interactive programming environment (Reps, 1984,Horwitz, 1985). Large program segments are stored for extended periods of time, and need to be manipulated efficiently during an editing session. To store a program segment, it is necessary to store the source text, which is usually not in a fixed format; to support a software tool like a syntax-based editor, it is desirable to maintain a parse tree in the database which is manipulated while the program is edited. It is obvious that fixed-format records cannot support these operations with high performance and low resource consumption. Relational database systems are too limited in their capabilities to store, manipulate, and query data structures that are not easily encoded in fixed-format records. Other application areas, like CAD/CAM (Lorie, 1983), office automation (Tagg, 1984), and text process-

ing (Pavlovic-Lazetic, 1986), also need to store massive amounts data permanently on secondary storage devices. The file systems typically used for these application areas, however, do not provide the desired services or do so ony in rudimentary form.

1.1.1. Proposed Concepts

To provide the database services mentioned above to non-traditional application areas, several approaches have been proposed in recent years. We call these approaches the bridge approach, the extended relational approach, and the toolkit approach.

The bridge approach is based on the observation that there are several relational database systems available and ready to use. Instead of implementing a new database system, it is proposed to design an interface layer that serves as a bridge between the data modelling desired for the application and a relational database. The advantages of such an approach are obvious: The largest part of the data management software can be used reliably without modifications, and functionality like concurrency control and recovery is immediately available. The cost, however, can be substantial. Depending on the application, the translation between application specific data structures and relations can be quite cumbersome. If the application has performance constraints, as in the interactive programming environment mentioned above, this approach might be unacceptable.

The extended relational approach is taken by two project teams. The POSTGRES data model and database system (Stonebraker, 1986) and the Starburst system (Schwarz, 1986) are two designs that are based on the relational model. Several significant extensions, however, are incorporated in the designs. The hope is that these extensions will suffice for the application domains that have been identified for the next generation of database systems. Both systems, however, allow for some amount of extensibility. It is possible to define new abstract data types and to introduce new access methods into the run-time system (Stonebraker, 1983, Schwarz, 1986). These extensions provide for much of the desired flexi-

bility and are not trivial to implement. Besides the obvious changes to the run-time system, such extensibility poses new problems for the user interface, the schema or data dictionary management, and the query optimizer design. At this point, it is unknown whether the extended relational systems will be the most successful of the three approaches to extensibility in database systems. On one hand, it seems very likely that the systems will exhibit good performance, since most of the design for the query execution components can rely heavily on the experiences with relational systems. On the other hand, using the record based relational model as a the starting point might turn out to be the biggest obstacle for providing all desired data modelling capabilities.

The third approach, which we call the toolkit approach, is the approach to database extensibility taken by the EXODUS project.

1.1.2. The EXODUS Concept

Instead of having a single database system and refitting it for different application domains, the principal idea in EXODUS is to design and implement a new database system for each application domain, possibly by using existing parts. This has some clear advantages. First, the database system can be made to fit exactly to the situation's needs. Concepts of the application domain can be built into the database system and do not have to be provided by some interface or added-on extension. Second, only those capabilities that really are needed are implemented. There is no unnecessary code to be written, maintained, stored, or executed. Third, the database system can be tuned according to the application's requirements and characteristics. Finally, the user interface and the query optimizer can be designed with a maximum amount of knowledge about the application domain. The user interface is most important to the acceptance of the final product, and query optimization needs all the help it can get to estimate parameters of the stored data and the query, e.g. to anticipate a predicate's selectivity or an operator's execution cost.

The apparent disadvantage of this approach is that it requires substantial effort to implement a database system. It is certainly a conservative estimate that a database system consists of more than 50,000 lines of code, which corresponds to about $2,500,000 at $50 per line of code (Gray, 1987). The goal of the EXODUS project is to ease the implementation of a database system for a new data model. We plan to provide several software tools and program libraries that structure and simplify this immense task. The initial idea was outlined by Carey and DeWitt (Carey, 1985) and an overview is given in (Carey, 1986). The basic idea is that all database systems consist of a fairly standard set of modules. Building a database system should be significantly simplified if these modules are standardized, and if there are software tools to assist in implementing each module. The optimizer generator presented in this dissertation is one of these EXODUS software tools.

Before we outline more of the details of the EXODUS concept, we would like to address a general concern that arises whenever the EXODUS concept is presented. What happens if there are two application domains that need to share data, e.g., manufacturing data used for both accounting and technical product management? If there are two specialized database systems that are designed independently, how is it possible to share data? The answer is that with EXODUS, it is possible to design and implement a database system that is capable of serving both purposes at the same time. Furthermore, it is possible to implement the elements of this combined database system almost as if one is designing the two systems independently. This is possible because the EXODUS approach leads to a modular design. We will see how this is true for the optimization component.

1.1.2.1. The Database Implementor

The EXODUS software is not a database system in itself. Rather, it assists in building a database system. Clearly, the person who uses the EXODUS software to build a new database system has a crucial role in this concept. We envision a person with some back-

ground in database technology and a fair amount of understanding of the application domain. It is important that only familiarity with database concepts, not sophisticated database expertise, is necessary.

To implement a new database system, the database implementor, henceforth called the DBI, invokes EXODUS software tools with appropriate description files and links the generated code with EXODUS libraries and some of her or his own code into an executable database system, ready to assist the users.

1.1.2.2. EXODUS Tools and Libraries

This is only a short overview of the pieces of EXODUS software under development; a more detailed overview is given in (Carey, 1986). Besides the optimizer generator, there is the implementation language E (Richardson, 1987), the storage system (Carey, 1986), the type and dependency manager (Frank, 1987), and the planned user interface generator.

E combines programming concepts helpful for database implementation which are not found in a single language today. Persistent objects, i.e. objects that are stored permanently on secondary storage, are accessible directly through program variables. Procedure calls to an I/O or buffering system are not required in E, since the compiler inserts calls to the EXODUS storage system at appropriate places. Persistent objects can be updated through program variables, and concurrency control and recovery are provided for such updates. Type generators are provided in E to allow the DBI to write access methods for a variety of key and object types, without rewriting or recompiling the access method code. Iterators are a way of writing loops in a more natural way. Since scanning and processing of object sequences play an important role in database run-time systems, we felt that a more structured notation would simplify writing database systems.

The storage system provides the run-time support for E. It manages files of objects of arbitrary size, and allows recoverable updates, deletions, and insertions in such objects.

Besides high performance, the major emphasis in the design of the storage manager is to reduce space requirements as much as possible, both disk space for versions and log files and buffer space in the running program.

The type manager is a software management tool used to keep track of versions of programs, definition files, etc.. It runs continuously as a daemon process, and ensures that program segments and query code dependent on one another are always up-to-date. It employs a pattern-based technique similar to, but more general than, the UNIX make facility[1].

The user interface generator concept is inspired to some extent by compiler generators, e.g. YACC (Johnson, 1975), and by concurrent work on generating object-oriented user interfaces, e.g. (Maier, 1986). The user interface work in the EXODUS project is just getting started.

1.2. Database Query Optimization

One of the fundamental innovations that made the relational approach to databases a challenge in the beginning and a success today are the query languages based on predicate calculus. In earlier database systems, namely those based on the hierarchical and the network data models, the application program must navigate through the database, which includes links and pointers between data records. When using a relational calculus language, a user only specifies a predicate that the retrieved data should satisfy, and the database system determines the necessary processing steps automatically. Since there might be many possible processing strategies which differ by orders of magnitude in processing costs, the task of the query optimizer is to find the cheapest and fastest query processing strategy.

To illustrate the degree of actual savings that can be obtained by query optimization, we give here an example taken from Jarke and Koch (Jarke, 1984).

---

[1] UNIX is a trademark of AT&T Bell Laboratories.

Example 1.1: Consider the relational schema of a database that describes employees offering computer lectures to departments of a geographically distributed organization:

departments (***dname***, city, street address)
courses (***cnr***, cname, abstract)
lectures (***cnr***, ***dname***, ***enr***, daytime)

Key attributes are given in italics. Assume that a user is interested in

> "the names of departments located in New York offering courses on database management."

There are 100 departments, 5 of which are located in New York. A physical block contains 5 department records. There are 500 courses, 20 of which are on database management. The physical block size is 10 records. There are 2000 lectures, 300 are on database management, 100 are held in New York, and 20 (from 3 departments) satisfy both conditions. The physical block size is 10 records.

Assume that the sorting time is $N*\log_2 N$, where N is the file size in blocks, and that there is a buffer of one block for each relation. All relations are sorted by ascending key values.

Jarke and Koch give three example strategies. Only the I/O cost is considered here. Each step has the number of read (r) and write (w) disk accesses indicated.

Strategy 1

(1) Form the Cartesian product of the courses, lectures, and department relations (r: 200000), and

(2) Retain the dname column of those department records, for which the cnr of courses and lectures match, the dname of lectures and departments match, cname = "database management", and city = New York (w: 1).

total: approximately 200,000 accesses.

Strategy 2

(1) Merge courses and lectures (r: 50+200, w: 400).

(2) Sort the results by dname (r+w: 400 $\log_2 400$).

(3) Merge the result with departments (r: 400+20, w: 400+400).

(4) Select the combinations with city = "New York" and cname = "database management" (r: 800), and

(5) keep only the dname column (w: 1).

total: approximately 6000 accesses.

Strategy 3

(1) Merge courses with lectures (r: 50+200), and

(2) keep only the dnames of combinations with cname = "database management" (w: 2).

(3) Sort the dname list generated (r+w: 2).

(4) Merge the result with the departments relation (r: 2+20), and

(5) keep only those with city = "New York" (w: 1).

total: 277 accesses.

Jarke and Koch compare strategies 1 and 3 and conclude:

> "Thus a reduction by a factor of approximately 700 has been achieved. For larger databases and more complex queries, more sophisticated techniques may result in even higher reductions."

□

This potential for savings, or from another perspective, of waste, was realized soon after the relational model was proposed by Codd (Codd, 1970). Smith and Chang wrote in 1975: "In the long run, the worth of the relational model will be measured by how efficiently

a relational view can be implemented" (Smith, 1975). We believe that this statement is equally true for new data models today.

1.3. Outline of the Thesis

In the following chapter, we review related previous work. One section is devoted to optimization in relational systems, the other section to extensible database systems.

In Chapter 3, we introduce the general problem of extensibility in query optimization. We propose tree-based optimization as a general model for optimization in extensible database systems and present a suitable optimization strategy. Rules are used as the principal means to describe query optimization in this model, augmented with some code to connect the optimizer with the other components of the database system.

In Chapter 4, we present the prototype that was implemented in the EXODUS project. A generator produces executable code from the rules; the code is then compiled and linked with the other components to form a complete database system. Special attention is given to the search strategy and its parameters.

Experiences with this prototype are reported in Chapter 5. An optimizer implementation for a restricted relational model shows the validity of the generator approach, and the effectiveness of the parameterization of the search strategy. The optimizer for a new query evaluation method, developed independently of the EXODUS project, shows the versatility of the proposed optimization model.

The flexibility of the EXODUS optimizer generator inspired us to compare optimization and execution costs for bushy (e.g. INGRES) and left-deep (e.g. System R, GAMMA) trees in relational database systems. The experiments and their results are reported in Chapter 6.

In Chapter 7, we summarize our findings, suggest future work in the area, and offer our conclusions.

The appendix provides a complete example of a model description file.

1.4. A Remark on Examples

The examples in this thesis are mostly taken from the relational model. This is not because the concepts presented pertain only to this model, but because we expect that most of the readers are familiar with the relational data model, and the examples can readily be understood. After studying an example, the reader is asked to consider whether the example can be generalized to other data models as well.

CHAPTER 2

An Overview of Previous Work

Many database query optimization strategies have been proposed and implemented over the past 12 years. A recent survey of optimization algorithms and techniques is given by Jarke and Koch (Jarke, 1984). Most of these algorithms are designed for the relational data model, with some restrictions and extensions. Typical restrictions include omitting universal quantification and requiring the query predicate to be in conjunctive or disjunctive normal form. Typical extensions are nested queries, aggregate functions, and distributed execution. More recently, recursion and transitive closure have attracted significant research interest.

All relational query optimizers perform basically the same task. Given a user query, find the optimal access plan, e.g. the one with the smallest number of disk accesses. Nevertheless, optimizer designs differ considerably. In this chapter, we will review some existing optimizer designs. After looking at two well known optimizers, those of System R and INGRES, we describe an query optimizer design that has some commonality with the optimizers generated by our optimizer generator. Then we will review two other approaches to extensible relational query optimization, Freytag's work at IBM Almaden and a design done for the AIM project in Germany. Finally, we will take a brief look at work on search strategies in the context of Artificial Intelligence, since we borrowed some ideas from this area.

2.1. Relational Query Optimizers

2.1.1. System R

System R was developed at the IBM San Jose Research Laboratory as an experimental relational research vehicle beginning in 1975 (Astrahan, 1976). Numerous publications describe various aspects of the system, e.g. the query language SEQUEL (Chamberlin, 1974) (which was later revised and renamed to SQL), the view and authorization system (Chamberlin, 1975), query compilation into executable access modules and the management of those for application programs with embedded queries (Chamberlin, 1981), and the recovery system (Gray, 1981). An overview and an evaluation of the system is given by Chamberlin et al. (Chamberlin, 1981). More recent research work include extending the system to a distributed relational database system called System R* (Lohman, 1985).

It was one of the objectives of the project to support repetitive queries embedded in application programs without recompilation or reoptimization for each execution. Hence, the optimizer works independently from the query execution system. No intermediate query execution results were to be used in the optimization, and all optimization decisions had to be based on catalog information. The System R optimizer (Selinger, 1979) proceeds in five steps. In the first step, catalog information about the relations referenced in the query is cached. The second step determines interesting sort orders. A sort order of a relation is called interesting if it is on a join column or if it is explicitly requested in the query. The third step determines scanning strategies for all relations in the query, for results in each interesting sort order and for unsorted results. In the fourth step, scanning and join methods are combined to form the complete access plan. If the query is a simple query, i.e. it references only one relation and has no nested queries, the query can be executed with a single scan. In this case, step four is trivial. Generating code for the selected access plan in the access specification language is the fifth step.

Scanning is done in System R by the Research Storage System (RSS). The RSS provides segment scans and index scans. In segment scans, all blocks in a contiguous disk area are scanned, whether or not they contain tuples of the scanned relation. Index scans are possible on clustered and on non-clustered indices. They can be specified with lower and upper bounds, and are guaranteed to return tuples sorted on the index key. All scans support "sargable predicates", which are Boolean expressions in disjunctive normal form with clauses of the form "column comparison-operator value". Thus, simple selection predicates can be evaluated by the RSS. Tuples can also be retrieved in an interesting sort order by a scan with arbitrary sort order and a subsequent explicit call to the sort utility.

After scanning strategies have been determined for all relations and for each interesting sort order (including the unsorted case), join strategies are selected. System R uses two join methods, called nested loops join and merge join. Blasgen and Eswaran found these methods close to optimal among the cases examined in almost all cases (Blasgen, 1977). In the first pass, join methods for all pairs of relations with a join predicate are considered. Next, groups of three relations, four relations, etc., are optimized until all relations are combined. For each decision, results for smaller group sizes can be used. Combinations without a join predicate are not considered, because Cartesian products tend to be large, expensive, and unnecessary.

The cost of access plans is calculated as a weighted sum of disk I/O's and CPU time. The CPU cost is estimated as the number of calls to the RSS under the assumption that most of the time is spent in the RSS. Both measures are modeled more precisely in the newer System R*.

Example 2.1: Let us assume a database with two relations, "professor" and "course". We want to know the subjects of all courses in the physics department which are taught by a professor 30 years old or younger. A SQL query to retrieve this information is:

```
SELECT subject
FROM course
WHERE department = "physics" AND
teacher = SELECT employee-no FROM professor
   WHERE age <= 30
```

The interesting orders are professors sorted on employee-no and courses sorted on the teacher attribute. The scans selected are the least expensive ones to return the relations unsorted or sorted in their respective interesting orders, each with a search predicate on age or on department respectively. If suitable indices exist, they would be considered for scanning. The four possible join strategies are nested loops join of the unsorted relations and merge join on the sorted ones, with professors as the outer relation and courses as the inner relation, or vice versa.

□

A more exhaustive example is given by Selinger et al. (Selinger, 1979).

2.1.2. INGRES

The relational database system INGRES (Interactive Graphics and Retrieval System) and its query and data manipulation language QUEL were developed at the University of California at Berkeley (Stonebraker, 1976). It was first developed for use under the UNIX operating system on Digital Equipment Corporation PDP-11 computers. A significantly modified version is now commercially available for a variety of computers and operating systems.

The original query optimization of INGRES is called Decomposition (Wong, 1976). It was designed at approximately the same time as the System R optimizer, but a number of decisions were made differently.

The optimization strategy is intertwined with query execution. The algorithm simplifies the query predicate by "one variable detachment" and "tuple substitution". It is applied recursively to the query until the resulting predicate has only one range variable.

Queries with one range variable can be executed by a special module, the "One Variable Query Processor". The one variable query processor includes file scans and indexing methods. It roughly corresponds to scans in the System R model, and one variable detachment corresponds to simple queries.

If a range variable has a selection predicate associated with it, e.g. "p.age <= 30" in a QUEL version of the example above, one variable detachment can be applied. A temporary relation is created from the original one, containing only the qualifying tuples. After one variable detachment, the query predicate can be simplified. The range of the variable is changed to be the temporary relation, and the selection clauses applied to create the temporary relation can be removed.

When no further one variable detachment is possible, the algorithm resorts to tuple substitution. First, one of the range variables in the query predicate is selected for substitution. This selection is made by a prediction of further processing cost; it is not necessarily the range variable over the relation with the lowest cardinality. This selection is very important, in fact, variable selection "is the heart of optimization" (Wong, 1976). The relation associated with the selected range variable is scanned, and for ***each*** tuple a modified query is executed. Since the attribute values of each tuple are known at this time, all references to the range variable in the query predicate can be replaced by constants.

This simplified query has one less range variable and at least one selection clause. If there is only one variable left, the query can be handled by the one variable query processor. Otherwise, the decomposition algorithm consisting of one variable detachment and tuple substitution is applied to it.

Example 2.2:

Consider the previous example of a query referring to a professor and a course relation. The equivalent QUEL statement is:

```
range of p is professor
range of c is course
retrieve (c.subject) where
c.department = "physics" and
c.teacher = p.employee_no and
p.age <= 30
```

The first step is variable detachment. This can be done for both variables. The above query is transformed into

```
retrieve into p_temp (p.employee_no) where
p.age <= 30

retrieve into c_temp (c.teacher, c.subject) where
c.department = "physics"

range of p is p_temp
range of c is c_temp

retrieve (c.subject) where
c.teacher = p.employee_no
```

Further one variable detachment is not possible in the simplified query predicate. Thus, tuple substitution is applied. One of the variables c and p is selected, say p. Then for each tuple in p_temp, if its employee_no is ***value***, the query

```
retrieve (c.subject) where c.teacher = value
```

is executed and all resulting tuples are returned to the user.

□

Wong and Youssefi (Wong, 1976) give a more extensive example.

There are several significant differences between the optimization strategies of System R and INGRES. System R starts with processing strategies for simple relations, and builds access plans "bottom-up". INGRES starts with the complete query predicate and decomposes it "top-down", selecting the scan mechanisms as the final step. The System R optimizer considers all reasonable processing strategies (meaning all except those involving a Cartesian product). The number of possible processing strategies is exponential in the

number of relations involved. INGRES Decomposition only considers a linear number of processing strategies. Finally, System R considers two join procedures, nested loops join and merge join. INGRES has only one join mechanism, tuple substitution, which is in effect very similar to nested loops join.

The designers of INGRES are now well aware of the fact that merge join tends to be much faster for large relations, and that query decomposition does not always find the optimal join strategy (Stonebraker, 1980). For the commercially marketed version of INGRES, several changes were implemented. The decomposition procedure was replaced by an optimizer based on Kooi's thesis work (Kooi, 1980). In the new optimizer, a large collection of join trees is enumerated and evaluated. A join tree is a binary tree where each leaf node is a relation or index and each interior node is a join. For each join node, not only are nested loops join and merge join algorithms considered (if necessary with preceding sort), but also the use of permanent or temporary hash or ISAM indices. Since the number of join trees can be extremely large, it seems justified to preempt optimization in some cases before all possible trees are enumerated. When the time spent on optimization reaches a certain fraction of the time estimated for the best access plan found so far, optimization is not further pursued and the best plan found so far is executed.

2.1.3. MICROBE

A very different approach to query optimization has been taken for the MICROBE distributed relational database system. The query language of MICROBE is MIQUEL, which incorporates ideas from both QUEL and SQL.

After a query is entered by a user, the predicate is transformed into a parse tree, which is then transformed into an initial operator tree[1]. The optimization proceeds in two steps

[1] A trivial algorithm for the second transformation is given by Ullman (Ullman, 1982): Build the Cartesian product of all relations involved, let it be followed by appropriate selection clauses, and finally do a single project. The MICROBE algorithm is more sophisticat-

(Nguyen, 1982). First, the operator tree is restructured so that the expected amount of data which is transferred between operators is minimized and the total number of operations involved in the query is reduced. This can be done independently from the distribution of data over the sites participating in the distributed system, and is done before execution of the query. Second, the operators are assigned to sites dynamically during query execution. This aspect of the MICROBE system is not of interest here.

The transformation rules used during optimization are a set of quadruples (X, Y, C, A), where X is the subtree to be transformed, Y is the transformed subtree, C is a condition on X, and A is a set of side effects of the rule. The rules are coded in Pascal, and cannot be modified by the user. Nguyen et al. (Nguyen, 1982) describe the rule application:

> "The optimizer works recursively on the query tree. The top-down scan is used to label the nodes in the tree with Xi and Ci elements of the rules Ri, and the bottom-up scan to apply the appropriate rules. This is repeated on the tree until no more transformation rule applies. This process converges in o(n) scans, where n is the depth of the query tree."

The rules designed for MICROBE are not general algebraic equivalence rules. Rather, they are directed transformation rules which do not allow the reproduction of a query tree from itself, i.e. no composition of rules equals identity.

A more specific description of the rules is given by Nguyen et al. (Nguyen, 1982). They also give an example of rules that would not be allowed. Typical examples for legal rules include pushing selections and projections down in the tree, reordering joins to achieve less data transfer between operators, and combining select and project operators on a single relation to a special procedure called proselect.

---

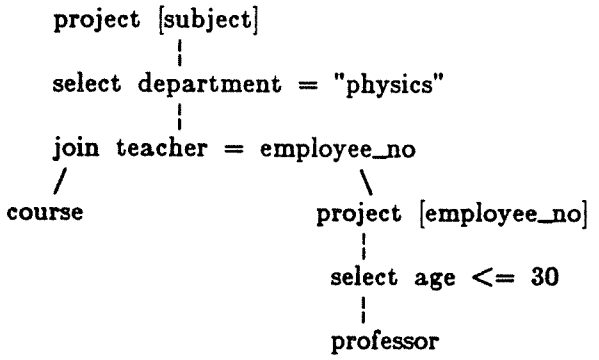
ed, avoiding Cartesian products where possible.

Example 2.3:

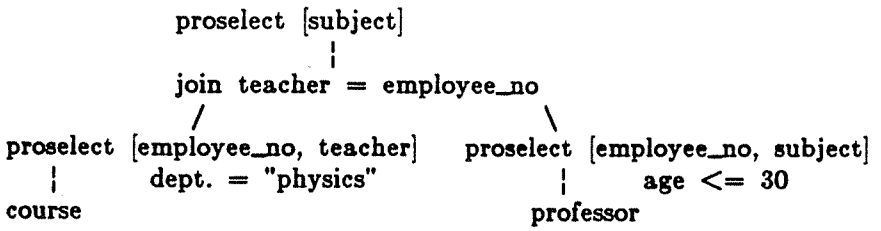
Let us consider the courses and professors in the physics department one more time. In MIQUEL, the query reads

```
SELECT subject
FROM course
WHERE department = "physics" AND
  teacher = SELECT employee-no FROM professor
          WHERE age <= 30
```

which is exactly like the SQL query. This is transformed to the initial operator tree



and to the final, optimized tree



□

The MICROBE optimizer differs radically from the first two optimizers. The System R optimizer and the INGRES Decomposition build on properties that follow indirectly from the properties of the relational data model and the file organizations employed. MICROBE, on the other hand, is designed rather closely along the mathematical properties of the relational model and its operators (Codd, 1970, Codd, 1972). The rules are almost independent

from one another. This modularization, we believe, not only simplifies design, implementation, verification and maintenance, but it also facilitates extensions without major redesign.

Flexibility and the ability to extend the data model and the database software are the major themes in the design of each of the extensible database system projects. This is why the optimizers of database systems based on EXODUS are similar to the MICROBE optimizer. However, EXODUS provides more flexibility than MICROBE. In MICROBE, the user cannot modify the Pascal implementations of rules. Producing Pascal code from the abstract transformation rules, a task performed manually by the MICROBE designers, should be eliminated or automated.

2.2. Extensible Query Optimizers

As pointed out earlier, each of the optimizers reviewed so far were designed to work for a particular data model, query language, and run-time system. Extensibility of the EXODUS optimizer generator means to support query optimization without making restrictive assumptions about the data model, the query language, or the run-time system.

Besides the work presented in this thesis, to the best of our knowledge there are currently two other projects attempting to use rule-based techniques for query optimization. Both are being designed for extensions of the relational model. Freytag is working on rule-based query transformation and optimization at IBM Almaden (Freytag, 1987), and Moeller and Bartels have worked on rule-based optimization for the NF^2-relation model which is being developed at IBM Heidelberg (Moeller, 1986,Bartels, 1986).

Other researchers have identified extensibility as a requirement for the query optimization component, e.g. for PROBE (Dayal, 1985,Manola, 1986) and POSTGRES (Stonebraker, 1986), and have suggested rule-based or description-driven designs, but no implementation has been reported to date.

2.2.1. Freytag's Rule-Based Optimisation

Concurrent with the work presented in this thesis, Freytag began to explore rule-based techniques to transform relational calculus predicates into execution plans (Freytag, 1987) as an extension to his thesis research on translating access plans into iterative programs (Freytag, 1986, Freytag, 1985).

The optimization task is divided into several phases. Each phase has a different purpose and a different rule set. The input into the system is a sorted list of predicates, which are assumed to be connected by AND's in the query. Notice that this requires the transformation of the query into disjunctive normal form. The first phase establishes which relations have to be scanned. The second phase distributes the selection predicates from the list of predicates to the appropriate scans. The third phase examines the indices that could assist in the scans. Then, the possible join orders are generated and the processing costs of the implemented join methods are calculated and compared. Freytag suggests to compile the rule set to achieve faster optimization (Freytag, 1987).

It seems that there are several problems with this approach when it is used in an extensible system. First, it is not clear that the general algorithm is applicable to other data models. In particular, there might be data models and query languages for which there is no normal form that could be used as a starting point for the optimization. For example, Moeller claims that there is no normal form for the NF^2 data model (Moeller, 1986). But it seems that this problem pertains only to the part of the algorithm which is concerned with transforming a relational calculus expression into relational algebra. More importantly, the rules are still a procedural description of the optimization process. We claim that the hard part of specifying an optimizer is not to code it in a particular language, like C or PL/1, but to invent and design the steps and procedures that implement the optimization. In that respect, it is not clear how much is gained by describing the optimization steps in a pro-

cedural rule language instead of in a systems implementation language. Finally, it is not clear how extensible this optimization concept is. Freytag motivates and describes his approach with extensions to the relational model, and uses only conjunctive relational queries for examples (Freytag, 1987).

2.2.2. NF^2-Relations

The Advanced Information Management project (AIM) at the IBM Scientific Center Heidelberg is based on non-first-normal-form (NF^2) relations. The data in a NF^2 database are stored in relations in which an attribute value can be a another (nested) relation. For their Diplom thesisses, Moeller and Bartels designed a rule-based optimization component for this project (Moeller, 1986,Bartels, 1986).

The query language, called NF-SEQUEL, allows the nesting of AND, OR, and NOT operators and existential and universal quantifiers for both tuples at the top level and for nested tuples.

Query evaluation in AIM is done by the ***walk manager***. A walk is a generalized form of a scan. It can result either in materialized tuples or in a list of tuple identifiers, called a ***filter extension table***. A tuple identifier can consist of more than one address, reflecting the nesting of tuples and relation valued attributes. A filter extension table associates a predicate with a list of tuples, or tuple pairs, triplets, etc.

For joins, AIM does not use merge join or nested loops join. Instead, the walk manager intersects filter extension tables or performs nested walks. A nested walk is similar to a nestled loops join, but it does not require the DBMS to materialize the (potentially deeply nested) tuples in main memory.

The optimizer builds the access plan bottom-up from the search predicate. The development of the plan from the predicate is performed in a semantic network called the ***unit net***. Originally, the unit net is very similar to the parse tree of the predicate. In the

first step, bindings of variables are determined, and appropriate pointers are inserted into the net. Next, the unit net is augmented with relevant information from the catalogs. Finally, in a bottom-up traversal of the tree, two plans are created for each node in the predicate tree. One of the plans is designed to produce the tuples that satisfy the predicate, and the other would produce a filter extension table for the predicate.

The logic governing the transformation of predicates into plans is expressed in rules. A rule compiler compiles the rules into Pascal. Before the rule set is applied to a predicate, the predicate is matched against a set of predicate fragments kept in a library for this purpose. If a matching fragment can be found, the precomputed plan associated with the fragment is used.

The purposes of capturing the optimization process in rules are modularity, extensibility, and uniformity. While the goals of modularity and uniformity are well served with this design, we believe that the system provides only limited extensibility. In particular, we believe that requirements frequently mentioned in the current database literature, e.g. transitive closure and other recursive operators, cannot easily be integrated into this design. It is, probably, extensible within the NF^2 relational model, e.g. to include new access methods in the system. It seems that the design of the AIM data model was not finalized when the optimizer was developed. Whether the optimizer satisfies the needs of AIM is not clear, because this software developed at the university in Darmstadt was not integrated with the AIM software at IBM in Heidelberg.

CHAPTER 3

Extensibility in Query Optimization

The goal of the EXODUS project is, as mentioned earlier, to make it as easy as possible to implement a new database system, or to augment one which has been implemented using EXODUS. The challenge is to provide software tools and libraries that make few or no assumptions about the data model to be implemented. For the optimizer component, this means we do not want to restrict the number of operators, or prescribe the operators in the system. This distinguishes EXODUS from other extensible database projects, namely POSTGRES and Starburst. Consequently, it is impossible to provide a single, final query optimizer with the EXODUS software. Since the optimizer is one of the most intricate subsystems of a database system, however, we felt that it is essential to support the DBI in designing and implementing the optimizer.

The design goal of the EXODUS optimizer component is twofold. First, it is imperative to provide a very general and powerful model of optimization. Second, it should be easy to specify the optimization, and the DBI should be encouraged to use a modular design in his or her implementation efforts.

The number of access plans can be very large for a complex query. For example, the number of possible join orders for a multi-relation query in a relational system is O (8^N), where N is the number of equi-join operators in the query (Muralikrishna, 1986). The combinatorial explosion of possible access plans might be even more significant for more advanced data models. Hence, it is necessary to give the DBI a means to limit the search, without requiring too much sophistication from the DBI.

3.1. Tree-Based Optimization

While pondering what a suitable model of optimization would be, we let ourselves be guided by the model of execution that we anticipate for future data models. Incidentally, this model of execution also received special attention in the design of the E programming language (Richardson, 1987,Richardson, 1987). A run-time system for a database typically consists of a limited set of procedures. Each of these procedures transforms a data stream according to an argument which was derived from the original query. A typical example is a selection operator which eliminates from a stream those tuples or records that do not satisfy a predicate provided in the query. To evaluate complex queries, such procedures can be nested, i.e. the output of one of them can be the input of another one. The transfer of data between such procedures we call a data stream or simply a stream, without making assumptions about how this transfer is physically arranged, e.g. by temporary files, shared memory, or messages, and how the procedures are synchronized.

If we assume that this is how queries will be evaluated in EXODUS based databases, we can infer that queries can always be expressed as trees of operators. For the relational model, a trivial algorithm to transform a relational calculus query into an execution tree is given by Ullman (Ullman, 1982). In this algorithm, it is obvious that the tree which is easiest to derive from the original query is not the optimal tree to execute. But it is desirable to leave it to the optimization step to find the cheapest equivalent execution tree, as this minimizes the work required from the user interface.

For EXODUS, we decided to require the user interface to produce a tree of operators that represents one correct sequence of operations to answer a query. The optimizer will transform the presented query tree into one that promises a more efficient execution of the query. This is done step by step, with each step being the transformation of a query tree or a part of it into an equivalent query tree. These transformations include replacement of

operators (e.g. Cartesian product and selection by a join), insertion of new operators (e.g. an additional project to eliminate fields as early as possible), and rearrangement of operators to achieve lower processing cost.

Frequently, there is more than one implementation procedure for a given operation. For example, a number of join methods have been developed for the relational equi-join (Blasgen, 1977), and most database systems have a repertoire of more than one of them. Therefore, the optimizers for EXODUS based database systems will distinguish between operators and methods. Operators are on the logical level, i.e. an operator and its argument determine the mapping from the input stream(s) to the output stream. Methods are on the physical level, i.e. a method specifies the algorithm employed. For example, a relational equi-join is an operator, and nested loops join, merge join, and hash join are corresponding methods for this operator. Part of the optimization process is to find the cheapest set of methods to implement a particular operator tree.

It is important to notice that the correspondence between operators and methods can be complex. A single method can implement more than one operator, or a single operator can require more than one method. Consider a relational equi-join as an example. If duplicate elimination is not considered part of the project operator, it is easy to include a projection in any procedure that implements the join. A single method (algorithm, procedure) can perform more than one operator. Similarly, a merge join is only possible if both inputs are sorted, otherwise extra sort procedures are required. This potentially complex relationship between operators and methods must be captured in the optimization process. In EXODUS, it is part of the optimization to select which of several implementation methods for an operation is the most suitable in each case.

The easiest way to describe tree transformations and the correspondence of operator trees and method trees is by means of rules. Rules for operator reordering are termed

transformation rules, and rules for method selection **implementation rules**. In our framework for optimization, the data model of the target DBMS is described by a set of transformation rules and implementation rules together with a set of cost functions to predict the processing cost for implementation alternatives.

The rule set must have two formal properties — it must be **sound** and **complete**. **Sound** means that it allows only legal transformations. If a rule is not correct, no optimizer working with this rule can work properly. It is impossible for a software tool such as the EXODUS optimizer generator to determine whether a set of rules is sound; this can only be determined by the DBI who defines the data model by specifying the rules. Verifying the soundness of the rules would only be possible if the data model could be described independently, and the two descriptions could be compared. **Complete** means that the rule set must cover all possible cases, such that all equivalent query trees can be derived from the initial query tree using the transformation rules. If the rule set is not complete, the optimizer will not be able to find optimal access plans for all queries. Again, this cannot be verified automatically, because the set of equivalent query trees and access plans is defined only once.

To summarize our optimization model, our optimizers map trees of operators into equivalent trees of methods by operator reordering and method selection. The set of operators and the set of methods are data model dependent, and hence must be specified by the DBI. Similarly, the rules that govern the tree transformations and the operator to method mappings must also be specified by the DBI.

3.1.1. Cost Model

The purpose of the optimization step is to find the cheapest query evaluation plan for a given query. Clearly, the cost model plays a crucial role in the optimization process. It is very important that the cost model accurately anticipates execution cost for a given query.

Cost measures (CPU, I/O, network transfer) and formulas to anticipate query processing cost in database systems have received significant attention (Selinger, 1979,Lohman, 1985,Mackert, 1986,Mackert, 1986). Since we do not want to make assumptions about the operators in the data model, the methods, and the implementation of data streams, we cannot, unfortunately, assist the DBI very much in specifying the cost calculation.

In a database run-time system, cost occurs by processing data, i.e. by executing a procedure or method. Hence, the DBI wishes to associate with each method a cost function which calculates its processing cost as a function of its argument, e.g. join predicate, and the input data stream(s). This design also gives the DBI the flexibility to design his or her own cost measure, e.g. a weighted average of the number of disk I/O's and the number of CPU instructions (Selinger, 1979).

The cost calculation for an entire access plan is rather straightforward in this design. For the methods that are applied in a query evaluation plan, the appropriate cost function is invoked to calculate the cost for this method with the particular arguments and inputs, and the sum of the costs of all methods in a plan is the cost of the entire plan.

3.1.2. Search Strategies

To provide a significant help to the DBI in implementing query optimizers, we believe that it is important to assist in choosing and tuning the search strategy used to find the best query execution plan. Even for the relational model, which is not considered advanced or semantically rich, the number of equivalent operator trees for a given query grows very fast. Considering that there might be more than one method available for each operator, the number of feasible query execution plans can easily become unmanageable, even for only moderately complex queries.

A number of search algorithms has been reported in the literature (Barr, 1981). It is unclear, however, which of the algorithms used in database query optimization is most

suitable for extensible query optimization. Consider the query optimizers reviewed in the last chapter. The System R optimizers build access plans bottom-up, integrating more and more clauses and relations of the query in the plan. INGRES Decomposition, on the other hand, decomposes the query predicate top-down, splitting off and substituting one variable after the other. Microbe, finally, uses transformations of the query tree and specific properties of the data model and the transformations, without regard of the order with which the transformations are applied.

In order to achieve the desired generality, we needed to design an algorithm which does not preclude either one of these approaches, and which requires only the absolute minimum of formal properties from both the data model and the transformation rules. Hence, the desired search strategy allows query tree transformations in any order but also allows a directed search.

The search strategy adopted for EXODUS is a best first search (Barr, 1981). A state in the general formulation of the search strategy is a query tree. Every state can, in general, be expanded (for a query tree, we prefer to say transformed) in several directions. It is quite likely that the benefits of the various expansions differ considerably, hence we decided that not all states will be expanded fully in our search strategy. At any time during the search, there will be states which have been exhaustively expanded, others which are partially expanded, and some which are unexpanded. The expansions which are possible but not applied yet are called the *OPEN* set. Each entry in *OPEN* is a pair consisting of a query tree and a transformation.

At each step in the search, the transformation performed is the one which carries the ***most promise*** that it will eventually, via subsequent transformations, lead to the optimal query evaluation plan. The crucial element in this search strategy is the promise calculation, called the ***promise evaluation function***. In the problem at hand, it must include the

current query and plan, possibly others which have been found already, and information about the transformation rule involved. The most natural measure for promise is cost improvement of the access plans.

3.1.3. Modularization of DBI Code

In an extensible database system, there are always some parts in the optimizer (and in other components as well) which cannot be expressed in a restricted, e.g. rule-based language. We propose to allow these parts to be written in the DBI's implementation language, and to provide a software tool to combine the rules and the DBI's source code.

For easy extensibility, it is very important to assist the DBI in dividing the code into meaningful, independent modules. Not only is a modular optimizer easier to implement, but we also envision this as a help for a database management system that evolves over time.

In this section, we will briefly review optimizer parts that are data model dependent and hence must be provided by the DBI. Furthermore, we propose how to break them into modules. We generally suggest to associate these procedures with one of the concepts that we have introduced earlier, namely operators, methods, and rules. In Chapter 4, we will show how these concepts are realized in the EXODUS optimizer generator.

3.1.3.1. Data Model Dependent Data Structures

There are two kinds of data model dependent data structures which are important in the optimization process. First, there are **arguments** for operators and methods. Second, in almost all cases it is desirable to maintain some dictionary information for intermediate results in a query tree. We term such dictionary information **properties** of the intermediate results. Since defining these data structures is part of customizing an extensible database system, the optimization component of such a system must treat these structures as "black boxes". In this thesis, we propose the use of a procedural interface to maintain and query these properties. Furthermore, we distinguish between operator and method

arguments, and between operator dependent and method dependent properties. As an example from a relational system, cardinality and tuple width are operator dependent properties, whereas sort order is a method dependent property.

3.1.3.2. Rules and Conditions

In the EXODUS optimization concept, the set of operators, the set of methods, the transformation rules, and the implementation rules are the central components that the DBI specifies to implement an optimizer. The rules are non-procedural; they are given as equivalence laws which the generator translates into code to perform tree transformations. Each of these rules should be self-contained. Only then is it possible to expand the rule set safely as the data model evolves.

The transformation rules express equivalency of query trees. Tree expressions, i.e. algebraic expressions, embody the shape of a tree and the operators in it. For some rules, however, applicability does not depend on the tree shape and the operators alone. For example, some transformations might only be possible if an operator argument satisfies a certain condition. Since operator arguments should be defined by the DBI, such conditions cannot be expressed in a data model independent form. We allow the DBI to augment rules with source code to inspect the operator arguments, the data dictionary, etc.

3.1.3.3. Cost Functions

As mentioned earlier, processing cost occurs by executing a particular algorithm. The cost calculation is closely related to the processing method being executed. Hence, we propose to associate cost functions with the methods, and calculate the cost of a query execution plan as the sum of the costs of the methods involved. The parameters of a cost function are the characteristics of the data streams serving as inputs into the method, e.g. the number of data objects in each input data stream, and the method argument, e.g. a predicate.

3.1.3.4. Property Functions

The characteristics of the data stream, which are needed as parameters to the cost functions, are data model dependent. Thus, they must be defined by the DBI. We attach characteristics, which we call properties, to both the operators and the methods. Operators (and their arguments) determine the logical properties of a node in a query tree, e.g. cardinality. The choice of a particular algorithm or method defines physical properties of the intermediate result that this node stands for, e.g. sort order.

Besides the conceptual differentiation of operator and method properties, there is also a practical reason why the two should be separated. The operator properties should be computed as early as possible, in particular before the cost functions are invoked. Determining the cost can be easier if the operator properties, e.g. cardinality, have been derived already. The method properties, on the other hand, can only be computed after the method has been determined, which is after the cheapest method has been found using the cost functions.

3.1.3.5. Argument Transfer Functions

Arguments to operators and methods, e.g. predicates, are also data model dependent, and can, therefore, only be modified by DBI code. If a transformation involves only operator reordering, there is a correspondence between operators in the old query tree and in the new query tree. In these cases, arguments can be copied between corresponding tree nodes. If this is not possible, we suggest that the DBI be allowed to provide a function along with the transformation rule which is called when the rule is applied and which determines the arguments of the new operator nodes. For example, if a complex selection predicate must be broken up into smaller pieces, a function provided by the DBI should be associated with the transformation rule. When the rule is applied, this function is called to perform the necessary manipulations on the argument data structure.

3.1.3.6. Promise Estimation Functions

In the search strategy, a promise evaluation or estimation function is used to anticipate how beneficial the application of a rule will be and to decide which transformation to apply next. It is not easy to design a general scheme to do this, and any general scheme will suffer from its generality; sometimes it might be necessary to consider data model dependent aspects of the query tree to be transformed. For example, join commutativity will, on the average, have a neutral effect. If there are asymmetric join methods like hybrid hash join (DeWitt, 1984), however, the benefit of join commutativity depends on whether one or both of the relations will fit into a main memory hash table. Therefore, an extensible database system should provide a general scheme to estimate the benefit of applying a transformation to a query tree, and leave the option to the DBI to augment this scheme with specialized estimation procedures.

3.1.4. Limitations of Our Work on Tree Based Optimization

The optimizers discussed in this thesis have only limited scope. They optimize a single query at a time, base their decisions solely on information available prior to execution, and are meant for a centralized system. Most existing relational query optimizers have the same restrictions, and we feel that more research needs to be done before it is warranted to remove these restrictions in an extensible query optimizer design.

Regardless of the data model, we would like to recognize and exploit common subexpressions. This is the essential feature for global optimization, i.e. optimizing several queries at a time to achieve greater savings in answering the queries. Therefore, it is essential to find common subexpressions, and to optimize and to execute them only once. It is not clear, however, how to decide when to transform a subexpression that is used more than once in a query. Consider two subtrees in a query that share a common subexpression. If, after a transformation, only a small part of the common subexpression is shared among the queries,

the cost reduction that has been achieved for one of the uses might be more than offset by the reduction of sharing.

For distributed database systems, we offer as an initial idea to include a "transfer" operator in the algebra, which would represent data transfer from one site to another and which could be moved in the query tree by appropriate tree transformation rules. If data are replicated in the system, we suggest an operator similar to the "choose1" operator used in System R* (Selinger, 1980).

3.2. Relation to Other Database Functions

There are a number of functional units that are typically found in all database systems with which the optimizer interacts. In this section, we would like to outline some general assumptions about the relationship of the optimizer to these components.

3.2.1. User Interface, Preprocessing, and Parsing

The optimizer components of EXODUS based database systems expect the query to be transformed into an initial query tree before it is passed to the optimization procedure. There are several advantages and disadvantages to this design. The main reason for this decision is that it isolates the optimizer from all interface design issues. For optimizing a query, it does not matter whether the query was entered using an interactive interface, be it graphical (Zloof, 1977, Wong, 1982, Cruz, 1987) or command oriented (Chamberlin, 1974, Stonebraker, 1976), or whether it was part of a program with an embedded query (Chamberlin, 1981). There are, of course, issues involving the thoroughness of the search, but those can be communicated to the optimizer in other forms as well.

A disadvantage might be that there are optimizations that can be applied easily before the query is transformed into an operator tree, e.g. semantic optimization of the predicate (King, 1981). While we recognize this fact, we believe that a major emphasis of query optimization will remain with choosing from among many access plans.

After a query has been parsed successfully, the database is queried to ensure that the query is permissible in the sense that it only references existing database objects, and access protection is not violated. We envision that a facility similar to relational views will be provided by most database systems because this a proven way to realize an external schema on top of the conceptual schema. If a view is queried, it has to be combined with the query. Hopefully, other data models and query languages allow for the use of the techniques developed for relational systems (Stonebraker, 1975,Hanson, 1987).

3.2.2. Run-Time Systems

Clearly, the run-time system and the optimizer of a database system have a close correspondence. The optimizer should consider exactly those query execution plans that the run-time system can execute. A significant change in the run-time system almost always requires a change to the optimizer to take advantage of the new or improved features. Even though these two subsystems of a database system are closely related, there are very good reasons to keep them separated.

For commercial database systems, it is a requirement that queries can be run many times without changes or with different parameter values only. In these cases, it is usually not worthwhile to reoptimize the query every time. In fact, in many cases, this would be prohibitively slow. Consider, for example, a high-performance banking application. The money transfers performed by the tellers are equivalent except for the account numbers and the amounts involved. Optimizing each of these transactions separately would make no sense, since it would only waste resources[1].

---

[1] In other cases, in particular when range queries are involved, the situation is more complicated. The author suggests that the data structure that represents the access plan include the original query, possibly in parsed form (Chamberlin, 1981), and a predicate on the actual parameters. When a compiled query is invoked, the predicate is evaluated with the actual parameters. If it is satisfied, the query is reoptimized. Since this predicate can be prepared just as database search predicates are that operate on database records, e.g. compiled into machine language, this method gives a maximum of flexibility for hardly any

Another reason why we would like to separate the query optimizer from the run-time system is that it is not clear on which level these subsystems should communicate. The run-time system could "walk the tree", or the plan might be put into a special data structure, as is done for example in GAMMA (DeWitt, 1986), or it could invoke a procedure which was generated from the plan. Mixed forms are also possible. If the run-time system is not able to interpret directly the output of the optimizer, there is another component in between that transforms a query execution plan into a program in machine language. Interesting research on how to do this step with a rule-based system has been done by Freytag (Freytag, 1986,Freytag, 1985). In EXODUS, query execution plans will be transformed into programs in the database implementation language E (Richardson, 1987,Richardson, 1987), which is a very convenient target language because its iterator construct was designed with this application in mind.

3.2.3. Schema and Data Dictionary Support

As pointed out above, the optimizer depends solely on information about the database which is available prior to execution of the query. That means that schema and data dictionary support is very important to optimizers in EXODUS.

In this thesis, we do not suggest which information should be kept in the data dictionary, or any particular implementation. First, since we do not know the data model that a DBI might want to implement using EXODUS, it is impossible to specify exactly what data dictionary information is required. Second, even for the relational model, for which the alternative execution strategies and the decision rules are reasonably well understood, there is currently no standard as to what information should be kept in the data dictionary or in

---

cost. Designing the algorithms used by the optimizer to build this predicate is an interesting problem that, to the best of our knowledge, has not been investigated yet. In fact, the stability of access plans, i.e. the range of query parameters and database states in which an access plan represents the best choice, has yet to be addressed. Another, possibly complementary, solution to the problem is to produce several access plans or scanning strategies and choose one of them depending on the actual parameters of the query.

the schema, and how closely it should reflect the state of the database[2].

For these reasons, the optimization component of EXODUS must be designed in a way that allows the DBI to use sophisticated data dictionary support without requiring it. The optimizer actually does not depend on the data dictionary itself, it depends on information found in or derived from the dictionary. Instead of concentrating on the data to be kept in the dictionary, we propose a functional interface. For example, we do not require that the data dictionary or schema of an EXODUS based database system contain cardinalities. All the optimizer really depends on are the cost calculations. If the DBI wishes to support the cost calculations with information typically found in the data dictionary, e.g. cardinalities and data distributions, we suggest that the DBI provide property functions that provide this information. The EXODUS software ensures that these functions are called by the optimizer.

3.2.3.1. Abstract Data Types

Frequently, abstract data types (ADT's) to be defined by database users are viewed as an essential component of extensible database systems, e.g. in (Dayal, 1985, Stonebraker, 1986) and (Schwarz, 1986). In programming languages, ADT's are defined externally by their procedural interface, i.e. the interface specifies only how to manipulate and query an ADT. This roughly corresponds to the run-time system in the database world. For query optimization purposes, this presents a problem, because the optimizer has to anticipate processing costs, selectivities, etc. Dayal and Smith suggest to make the optimizer parts that need to know about ADT's description-driven (Dayal, 1985). Stonebraker and Rowe propose

---

[2] The author suggests more investigation of statistical moments for estimating selectivity factors. There are two reasons why moments can be very useful in database systems. First, it is also possible to use co-moments that describe the correlation between attributes, and this could help in estimating join selectivities (Yang, 1985). Second, moments can be maintained quite easily. They can be maintained in a transaction consistent state, and incorporated into the data dictionary in a very efficient manner, e.g. using Escrow methods (O'Neil, 1986).

to include functions in the ADT definition to assist the optimizer in these needs (Stonebraker, 1986).

For extensible optimization, however, ADT's are an orthogonal problem. The extensible optimization component uses a procedural interface to the operators and methods in the system, e.g. join and hash join. Whether or not a selection predicate involves an ADT is of no interest to the extensible optimization component. The extensibility required to deal with ADT's is located in the condition code, the cost functions, the property functions, etc., but not in the actual optimization code.

There is another way to look at ADT's in extensible optimization. The optimization component is not interested in what is stored in the data dictionary; it only needs access to a set of procedures to derive information from the dictionary, e.g. processing costs or selectivities. These procedures are the cost functions and property functions described above. If the data dictionary is only specified by its functional interface, as we propose for the EXODUS optimization component, ADT's are not visible to the optimizer. Rather, they present themselves as schema maintenance and schema interpretation problems. The extensibility required to handle ADT's in the optimizer is located in the data dictionary routines. It rests with the DBI to implement it, and it does not require special action by the actual optimization procedures.

3.3. Implementation Considerations

3.3.1. Generator vs. Interpreter

The concepts outlined so far can be implemented in many different ways. From a practical standpoint, it is an important decision whether to interpret or to compile transformation and implementation rules. An interpreter has one potentially significant advantage. Typically, interpreters allow augmention of the rule set at run-time. For example, the optimizer could be designed in a way that it observes which combination of rules occurs

frequently. To speed up a combination of rules, a new rule representing a combination of these rules could be created by the running optimizer and used in further transformations.

Before we designed the optimizer generator, we experimented with interpretative rule-based optimization. The initial idea was to use one of the languages used in artificial intelligence which typically incorporate rules, pattern matching, and a search engine. We considered Prolog (Warren, 1977,Clocksin, 1981), OPS5 (Forgy, 1981), and Loops (Bobrow, 1983). Based on local availability and expertise, we chose Prolog. We designed two Prolog rule files. One contained those rules to guide the search for the optimal access plan, and to keep track of the cheapest access plan found. The other contained the rules describing the data model, i.e. the operators, the methods, and the cost calculations. While this prototype successfully rearranged relational algebra expressions, it also showed Prolog's limitations for our purpose. Surprisingly, its most prominent disadvantage is the search strategy. At any point, only the current plan can be considered, and the applicable rules are tried in their syntactic order. This is a result of the fact that the search strategy is largely implicit in the Prolog interpreter, and the actual implementation task consisted of writing code to let the interpreter backtrack through the promising expressions, keeping track of the best expression encountered. The invocation of rules by the interpreter also makes it hard to gather information needed to improve the optimizer performance. Implementing and experimenting with alternative search strategies is one of the main objectives of our optimizer research work. For this reason, we decided not to use Prolog in the EXODUS system. Another practical problem is the execution speed of our local Prolog implementation (the C-Prolog interpreter).

As one alternative to using an existing language, we considered briefly to design and implement our own interpreter, possibly by modifying source code of the Prolog interpreter. While this would have alleviated some of the problems encountered, we anticipated that the

performance would still be unsatisfactory. Consequently, in order to achieve maximum performance, we decided to build a program generator. The rules describing the data model (for optimization purposes) would be translated into C code, which in turn would be compiled and linked with other EXODUS components. We believe that the higher execution speed of compiled rules more than outweighs the benefits of being able to create new rules at run-time.

3.3.2. Rule Language

Since the design of the generator is not bound to any particular rule language, we designed a new rule syntax for query tree equivalence. The rules from which the optimizer generator produces code consist of two trees and an optional condition. In the case of a transformation rule, both trees are query trees. In the case of an implementation rule, one tree is a query tree and other is the corresponding part of the access plan.

It seems natural to formulate a tree as an algebraic expression. For our purposes, we decided to use prefix notation. The operators are used as function names, and the input streams appear as function arguments. If the specific characteristics of an input do not matter, a variable is used in its place. It is assumed that multiple occurences of the same variable designate the same subtree. Using variables, it is possible to express equivalence laws easily and concisely. A special symbol separates the two expressions. For example, join commutativity can be expressed as:

join (X, Y) <-> join (Y, X)

In order to distinguish transformation rules from implementation rules, different special symbols are used between the expressions.

In general, it should be sufficient to state equivalence as a logical law without procedural aspects being incorporated in the rule. In those cases in which a rule makes sense only in one direction, a different symbol (arrow) is used. It is important, however, that the

correctness of the system does not depend on procedural instructions. Otherwise, the rules are procedural rules. The purpose of using algebraic equivalence laws to specify the optimization is to free the DBI from designing the control of the optimization.

The rule conditions typically include accesses to dictionary information. Since the DBI designs the dictionary, it is easiest for the DBI to code these conditions in her or his implementation language. Since the rules are translated into the DBI's implementation language, it is easy to include the condition code in the generated code.

In Chapter 4, we will see the exact definition of the rule syntax, and how an optimizer is generated from the rules and the support functions written by the DBI.

CHAPTER 4

The EXODUS Query Optimizer Generator: Design and Implementation

We implemented a prototype of the optimizer generator. It was intended to serve several purposes. First, it shows the feasibility of the approach. Second, it is used to get preliminary performance figures. Third, it helps to identify the important parameters, their influences, and their ranges. Finally, it will constitute the optimization component of the EXODUS extensible database system.

In this chapter, we present the design and prototype implementation of the optimizer generator. The first section describes the generation of an optimizer from a description of the data model, as well as required and optional complementary code written by the DBI. The second section concentrates on how a generated optimizer works, how it searches for the best access plan, and what the parameters of the search strategy are. Most of the issues that were introduced in the last chapter are found here again, together with the chosen solution.

In addition to the concepts introduced in Chapter 3, we also allowed the DBI to organize the optimization into several phases. For each phase, there can be a different rule set, different search parameters, and different stopping criteria. The query tree corresponding to the optimal access plan of one phase serves as the input tree of the next phase.

4.1. Optimizer Generation

To produce an optimizer, the DBI invokes the generator on a **model description file**. This is done only once, at **database system generation time**.

The resulting query optimizer can then be used indefinitely. Figure 4.1 illustrates this paradigm. The output of the generator consists of two files: one with C code and one with C definitions to be included in other source code files written by the DBI. From the model description file, the generator produces 5 procedures. At the end of the code file, approximately 40 more procedures are appended. Providing source code and not a library of compiled object code seemed the most practical approach because much of this code depends on constants defined by the optimizer generator while processing the model description file.

We will describe only those aspects of the system that are essential to understanding the overall operation of the optimizer generator. Further details about the content of the model description file, the translation process, and the DBI code are given in Appendix A.

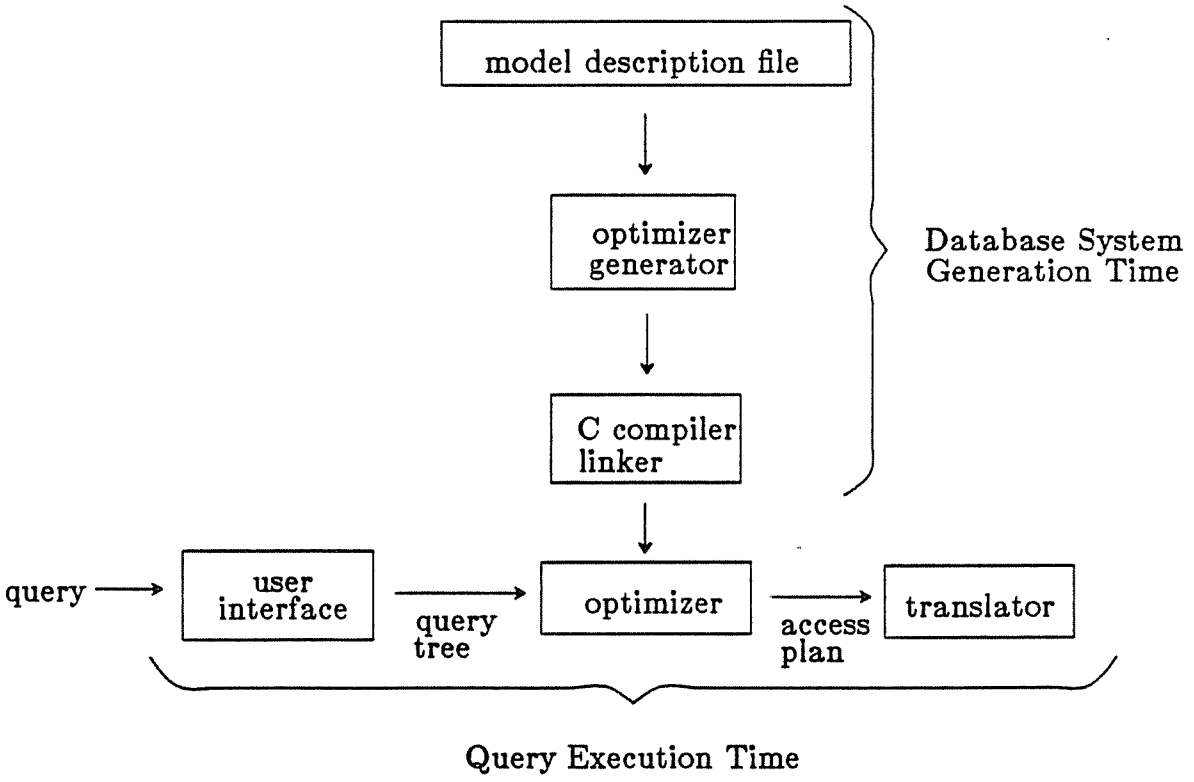


Figure 4.1.
Optimizer Generation.

4.1.1. The Model Description File

The input of the optimizer generator is designed in a style similar to that of YACC (Johnson, 1975). The output file is a set of C procedures. The model description file has two required parts and one optional part. The first required part is used to declare the operators and the methods of the data model. It can include C code and C preprocessor declarations to be used in the generated code. The second required part consists of transformation rules and implementation rules. The optional third part contains C code which is appended to the output file.

The first part of the input file is called the **declaration part**. It is used to declare the number of optimization phases and the operators and methods in the data model. If the number of phases is not explicitly declared, there is only one phase in the generated optimizer. Operators are declared with the keywords *%operator* followed by a number to indicate the arity and by a list of operators with this arity. Methods are declared in the same way using the keyword *%method*.

Besides operator and method declarations, the first part of the description file can also include C code which will be copied into the optimizer code file before the generated code. In particular, it is possible to put type, constant, and macro definitions into these code sections. There are three names that the optimizer sets to a default value if they are not declared in the description file. These are *ARGUMENT*, *OPERATOR_PROPERTY*, and *METHOD_PROPERTY*. They are data model dependent types. Their use will be explained later in this chapter. The order of the entries in the declaration part is immaterial, as is the position and the number of code sections.

Example 4.1:

```
%operator 2 join
%method 2 hash_join merge_join
%{
typedef char string [50];
```

```
# define ARGUMENT string
%}
```

This fragment of a declaration part shows the definition of the binary operator ***join*** and the binary methods ***hash_join*** and ***merge_join***. The code section, limited by %{ and %}, includes a type definition for the type ***string*** and the C preprocessor definition of the name *ARGUMENT*.

□

The second part of the description file, called the **rule part**, contains the transformation rules and the implementation rules. In Chapter 3, we required that the rule set be sound and complete. On the other hand, the rule set can be redundant. In fact, if the DBI foresees that a certain combination of rules will be used frequently, it is recommended (but not required) that this combination be specified as a single rule. While this will speed up the optimization process, it will not affect the access plan produced by the optimizer, unless the search parameters (described below in the section on the search strategy) are set too restrictively.

Transformation and implementation rules can be mixed freely within the rule part. The main part of a rule consists of two expressions and an optional condition. Between the expressions is the keyword ***by*** for implementation rules, or an arrow for transformation rules. Typically, the arrow is double-sided. A double-sided arrow means that the rule is an equivalence that can be used for transformations in both directions. We also call it a bidirectional rule. If the DBI wishes to ensure that the rule is used in one direction only, the arrow can be one-sided in this direction. If a one-sided arrow has an exclamation mark with it, the transformation is a once-only transformation, i.e. it cannot be applied to a query tree generated by this transformation. This feature is useful for enhancing optimizer performance, but it should never be necessary for correctness. A typical situation where it can improve the optimizer performance is a commutativity rule. Using commutativity twice

results in the original query tree[1].

An expression consists of an operator and an argument list. Each argument can either be another expression or an input. An input is indicated by a number. It stands for a subtree that is not affected by the transformation or implementation. For an implementation rule, the second expression consists of a method and a list of inputs. In the current prototype, methods cannot be nested in implementation rules.

Example 4.2:

```
join (1, 2) ->! join (2, 1);
join (1, 2) by hash_join (1, 2);
```

The first line of this example is the join commutativity rule. Since applying it twice results in the original form, the once-only arrow (with exclamation mark) is used. The second line indicates that ***hash_join*** is a suitable implementation method for ***join***.

□

Sometimes the same operator name appears twice in the same expression, e.g. in an associativity rule. In this case, it is necessary to identify the operators so that arguments, e.g. join predicates, can be transferred correctly when the transformation is applied. Operators in an expression can be followed by a number, called **identification**. If the same identification appears with an operator in the other expression of the rule, the arguments are transferred between these two operators. The operator numbering is independent from the input numbering.

Example 4.3:

```
join 7 (join 8 (1, 2), 3) <-> join 8 (1, join 7 (2, 3))
```

Example 4.3 shows a join associativity rule. The input streams not affected by the rule are

[1] When a query tree is generated that is exactly like one generated earlier, the duplication is detected and the new query tree is removed. Thus, allowing commutativity to be applied twice is only a performance and not a correctness issue.

numbered 1, 2, and 3. The ***join*** nodes are distinguished by the identifiers 7 and 8. Since the input numbering and the operator numbering are independent, 1 and 2 could have been used without confusion with the inputs. When this rule is applied in either direction, the join predicates are transferred between the nodes numbered 7 and those numbered 8.

□

Both transformation rules and implementation rules can have a condition. Conditions are written as C code. When the generated optimizer determines which rules are applicable to a given query tree, this code is executed after the pattern match succeeds. Special actions *ACCEPT* and *REJECT* are provided for use in this condition code. If no *REJECT* is executed, it is assumed that all conditions have been met. The condition code can access the arguments and properties of the operators and the inputs of the expression via pseudo-variables defined by the generator. These variables are called respectively OPERATOR_1, OPERATOR_2, etc., INPUT_1, INPUT_2, etc. The numbers in the pseudo-variables are the same as those used to identify operators and to represent the inputs in the rule expressions. Since a bidirectional rule counts as two rules in the generated optimizer, the condition code is inserted twice into the code file. To distinguish the two directions, one of the names *FORWARD* or *BACKWARD* is defined above the condition code in each case.

Example 4.4: The associativity rule in Example 4.3 lacked the condition code. The complete rule with the condition code is:

```
join 7 (join 8 (1, 2), 3) <-> join 8 (1, join 7 (2, 3))
{{
# ifdef FORWARD
if (NOT cover_predicate (OPERATOR_7.oper_argument,
        INPUT_2.oper_property, INPUT_3.oper_property))
   REJECT;
# endif
# ifdef BACKWARD
if (NOT cover_predicate (OPERATOR_8.oper_argument,
        INPUT_1.oper_property, INPUT_2.oper_property))
   REJECT;
# endif
```

}}

Example 4.4 illustrates the join associativity rule and the use of conditions to control the application of a transformation. Since the join operator appears twice in each expression, the numbers 7 and 8 are appended to distinguish the two instances of the operator. This allows the optimizer to transfer correctly the join predicates between the two operators as the transformation rule is applied. The condition code, the lines between {{ and }}, is copied twice into the optimizer code. Nevertheless, only one if statement from the condition code is executed for each direction; the other one is removed by the C preprocessor. The Boolean function ***cover_predicate*** is assumed to determine whether all attributes occurring in a predicate are in one of two relation schemas. The predicate is given as first argument, the schemas are given as second and third argument to the function ***cover_predicate***.

□

A rule can be preceded by the keyword ***phase*** and a number to indicate that this rule is only valid in one of the optimization phases. If a rule is to be used in several, but not all, phases, a range can be indicated with the keywords ***phase*** and ***to***.

Example 4.5: The rule

```
phase 1 to 2
join (1,2) ->! join (2,1);
```

will be applied only in the first two phases of the optimizer.

□

When a transformation is applied, argument fields are transferred between operator nodes in the "old" tree (the one that was transformed) to nodes in the "new" tree (the one resulting from the transformation). By default, the transfer is performed by copying. If an action other than copying is desired, an **argument transfer function** can be specified following the "new" expression in a rule. The new expression is the one that describes the

query tree after the transformation. This function is invoked whenever the rule is applied to fill in the argument fields in the new query tree.

Finally, the DBI can choose to assist the optimizer in estimating the benefit that can be expected from applying a certain rule. The standard method of estimation is described below in the section on the search strategy. To replace the default, the DBI appends the keyword ***estimate*** and a function name to the new expression in a rule.

Example 4.6:

```
join (1, 2) ->! join (2, 1)
    flip_join_argument
    estimate join_comm_benefit
```

After a query tree has been determined to match this rule, the function ***join_comm_benefit*** is called to estimate the execution cost of the query after the transformation. When the join commutativity rule is applied, the procedure ***flip_join_argument*** is called to reverse the join predicate. Using this function allows the DBI to ensure that join predicates are always maintained in a way that attribute on the left of the equation predicate refer to the left input, and right side attributes to the right input. Example 4.4 shows a case where this convention makes it easier to specify condition code for rules. The arguments to benefit estimation functions and argument transfer functions are described below in the section on DBI support functions.

□

4.1.2. Code Generation from Rules

The principal purpose of the optimizer generator is to translate the rules into executable code. A generated optimizer consists of about 45 procedures, three of which are generated from the rules, two of which are generated from the declarations, and the others of which are library procedures.

In this section, we will describe the code that is generated from the model description file, indicating only briefly the purposes of the generated procedures. Section 4.2 will outline the optimization algorithm, and show how the generated code is used.

When the optimizer generator is invoked, it creates a file for the generated source code and three temporary files into which the code for the procedures generated from the rules will be collected. Code sections are directly copied from the model description file into the source code file. After reading the declaration part, the generator appends constant definitions for the operators and the methods, and type definitions for the data structures *QUERY*, *PLAN*, and a number of internal data structures. *QUERY* and *PLAN* are used to build the initial query tree and the final plan tree respectively.

The transformation rules are translated into the procedures *MATCH* and *APPLY*. The former determines which rules can be applied to a particular query tree, the latter performs a transformation. The implementation rules are translated into the procedure *ANALYZE*. This procedure selects the cheapest implementation method for a given query tree. These procedures are created in the temporary files mentioned above, and the code for each rule is appended immediately after the rule has been read from the model description file. Bidirectional rules are considered as two rules in *MATCH* and *APPLY*. The generated procedures are built to work on a network of nodes called *MESH* which contains all query trees and access plans explored so far. *MESH* is described in detail in the section on optimizer operation.

After a rule is parsed, the correspondence between operator nodes in the two expressions is established. Operator nodes with an attached identification can readily be associated. Other operator nodes are associated only if they contain the same operator, e.g. the generator does not associate a join node with a select node.

Example 4.7: Consider the following transformation rule, which has been designed for this example only.

join 7 (join (1, select (2)), 3) <-> select (join (1, join 7 (2, 3)))

First, the join operators with identification 7 are associated. Afterwards, there is only one join operator left in each expression, and the generator assumes that there is a correspondence between them. Finally, there is only one select operator in each expression, and they become associated.

□

In the procedure *MATCH*, the generator produces a number of tests for each transformation rule. When a query is optimized, if all tests for a given rule succeed, the query tree and the rule are inserted into the set of possible transformations. This set is called *OPEN* and is described in the section on optimizer operation. If any one of the tests fails, the macro *REJECT* is executed which is actually a *GOTO* to the first test of the next rule.

Only the tests that are necessary are inserted into the code. The first test ensures that the rule is applicable in the current optimization phase. The second test considers the rule that created the query tree. In a bidirectional rule, this ensures that the query tree produced by one direction of a bidirectional rule will not simply be transformed back in the opposite direction. In a once-only rule, a node will not be transformed by the same rule that created it. The next tests match the pattern of operators in the rule with those in the actual query tree. The operators are encoded as integers in the tree node, ensuring that pattern matching is very fast. Finally, if the rule has condition code attached to it, this code is appended as the final test(s).

Example 4.8: Consider the following transformation rule.

```
phase 1 to 2 join 9 (1, join 8 (2, 3)) -> join 8 (join 9 (1, 2), 3) join_assoc
{{
   if ( /* the right side of join predicate 9 refers to input 3 */ )
```

```
        REJECT;
}}
```

After reading this rule, the generator appends the following code to *MATCH*. (The code lines are numbered in this and the following examples for illustration purposes only; the numbers are not generated by the optimizer generator.)

```
 1 TRANS_2:
 2 # define REJECT goto TRANS_3
 3 # define ACCEPT goto TRANS_2_ACC
 4     if (opt_cur_phase < 1) REJECT;
 5     if (level > 2) REJECT;
 6     if (node->operator != join) REJECT;
 7     if (node->oper_input[1]->operator != join) REJECT;
 8 # define OPERATOR_9 (*(node))
 9 # define OPERATOR_8 (*(node->oper_input[1]))
10 # define INPUT_1 (*(node->oper_input[0]))
11 # define INPUT_2 (*(node->oper_input[1]->oper_input[0]))
12 # define INPUT_3 (*(node->oper_input[1]->oper_input[1]))
13 {
14 # line 11 "model.opt"
15     if ( /* the right side of predicate 9 refers to input 3 */ )
16         REJECT;
17 }
18 # undef OPERATOR_9
19 # undef OPERATOR_8
20 # undef INPUT_1
21 # undef INPUT_2
22 # undef INPUT_3
23 # undef REJECT
24 # undef ACCEPT
25 TRANS_1_ACC:
26     open = (OPEN *) alloc_slot ();
27     open->rule = 1;
28     open->oper_nodes [0] = node;
29     open->oper_nodes [1] = node->oper_input[1];
30     open->input_nodes [0] = node->oper_input[0];
31     open->input_nodes [1] = node->oper_input[1]->oper_input[0];
32     open->input_nodes [2] = node->oper_input[1]->oper_input[1];
33     add_to_OPEN (open);
```

The first line is a label indicating that this is transformation rule 2. This label is used if the preceding rule is rejected. The next two lines (lines 2-3) show the definition of the macros *REJECT* and *ACCEPT*. If the current phase is not phase 1 or 2, the transformation rule is rejected (line 4). Since this rule is neither a bidirectional nor a once-only rule, there is no

test for the rule which produced the query tree under consideration. Next, the level of reanalyzing is checked (line 5). Levels of reanalyzing are described in the section tracing a transformation. The following two lines (lines 6-7) perform pattern matching. The variable ***node*** is an argument to the procedure *MATCH* and points to the root node of the query tree to be matched. To make pattern matching fast, the name ***join*** was defined to be an integer constant after the declaration part was read. The next five lines (lines 8-12) are definitions of pseudo-variables that can be used in the condition code. The condition code follows with a line indicator to ensure that syntax errors in the condition code will be reported by the C compiler with the correct line number (lines 13-17). After all pseudo-variables and the macros *REJECT* and *ACCEPT* have been "undefined" (lines 18-24), an *OPEN* record is allocated from the heap (line 26), filled with the rule number and pointers to the operator nodes and the input nodes of the query tree (lines 27-32), and added to *OPEN* (line 33). For a bidirectional rule, a similar piece of code would have been appended

□

Notice that all transformation rules are tested sequentially. Other strategies could be devised, e.g. using a ***case*** statement based on the root node's operator. However, the machine code produced for the generated tests is extremely short. For the code in example 4.8, the C compiler generates 7 machine instructions before the DBI's condition code. Run-time profiles of an operational optimizer show that only a small fraction of the time is spent in the procedure *MATCH*, indicating that a more sophisticated algorithm would not result in significant savings.

The procedure *APPLY* is called whenever a transformation, i.e. a query tree and a rule, is selected from the above-mentioned set *OPEN*. This procedure consists of a case statement with one case for each rule. First, new tree nodes are allocated for the new query tree and connected with their inputs. Then, arguments are transferred from corresponding

nodes in the old query tree. Finally, in a bottom-up pass through the new nodes, the procedure *REPLACE* is called for each new node. This procedure either replaces a node by an existing equal node or integrates it into *MESH*, which includes calling the procedure *ANALYZE*. Two nodes are equal if they have the same operator, the same argument, and the same input stream(s).

If there is at least one new node which cannot be replaced by an existing one, the new and the old root node are linked into the same class of equivalent nodes, and statistics about rule usage are updated. Notice that the root of a new query subtree is always in this set of nodes, or the set is empty. An equivalence class of nodes consists of the root of all equivalent subqueries, i.e. those that produce the same intermediate results.

Example 4.9: For the join associativity rule in Example 4.8, the generator inserts the following code into *APPLY*.

```
case 1 :
    new = alloc_node (join);
    new->oper_input[0] = alloc_node (join);
    new->oper_input[0]->oper_input[0] = open->input_nodes [0];
    new->oper_input[0]->oper_input[1] = open->input_nodes [1];
    new->oper_input[1] = open->input_nodes [2];
    {
        void join_assoc ();
        join_assoc (open->oper_nodes, open->input_nodes, new);
    }
    new->oper_input[0] = replace (new->oper_input[0], (NODE *) NIL, 0, 0);
    eql = replace (new, open->oper_nodes [0], open->rule, 0);
    if (eql == new) {
        update_rule_stat (open->rule, (REAL)
            ((new->oper_input[0]->local_cost + new->local_cost + TINY_COST) /
            (open->oper_nodes [1]->local_cost +
            open->oper_nodes [0]->local_cost + TINY_COST)), 0);
        if (open->oper_nodes [0]->rule != 0)
            update_rule_stat (open->oper_nodes [0]->rule, (REAL)
                ((new->total_cost + TINY_COST) /
                (open->oper_nodes [0]->total_cost + TINY_COST)), 2);
        new_equiv (new, open->oper_nodes [0], 1, 0);
    }
    else if (eql != (NODE *) NIL  &&
            eql->class != open->oper_nodes [0]->class)
        unify (eql, open->oper_nodes [0]);
```

Since this was the second rule in the model description file, it is case 2 in the ***case*** statement in *APPLY* (line 1). First, two new nodes are allocated which are the join nodes in the new query tree (lines 2-3). In the next three lines, the new join nodes are linked to their inputs (lines 4-6). Then, the argument transfer function which was indicated with the rule is called (line 7-10). In a bottom-up pass over the new nodes, the procedure *REPLACE* is called for each node (lines 11-12). If the new root node cannot be replaced, the rule statistics are updated in a way explained in the section on the search strategy (lines 13-21). A constant cost *TINY_COST* is added to the costs to prevent numeric problems in the case that the cost functions return zero costs. Also, the new root node is inserted into the equivalence class of the old root node (line 22). If the root node can be replaced, but the equal node is not in the same equivalence class as the old root node, then the two classes can be unified into one class (lines 24-26). For a bidirectional rule, two such cases would have been appended to *APPLY*.

□

The procedure *ANALYZE* matches a query tree with the implementation rules and calls the cost functions for each of the matching methods to determine which is the least expensive implementation method. Code generation for implementation rules is somewhat simpler than for transformation rules because the method expressions in implementation rules cannot be nested. After an implementation rule has been read, the generator produces code to match the rule pattern with the actual query tree. At optimization time, if the patterns do not match, the macro *REJECT* is executed which results in a ***goto*** to the next implementation rule. If the patterns match, *ANALYZE* builds the appropriate plan tree and calls the cost function associated with the implementation. If the calculated cost is less than the best cost found so far in *ANALYZE*, the method and the plan tree are stored in the current node of *MESH*.

The argument transfer from the operator argument field to the method argument field must be delayed until the method has been determined, i.e. at the end of *ANALYZE*. If an argument transfer function was specified with the implementation rule, and if the method is the cheapest one found so far, the address of this function is assigned to a pointer variable. At the end of *ANALYZE*, arguments are transferred to the method argument field. As for transformation rules, the default is copying. Another default function can be declared using the name *COPY_METHOD_ARGUMENT*. If the an argument transfer function was explicitly refered to in the implementation rule chosen, this function is called.

Example 4.10: Since the code generation for *ANALYZE* is similar to that of *MATCH* and *APPLY*, we show a fairly short example. Consider the following rule.

```
join (1, 2) by hash_join (1, 2);
```

For this rule, the generator produces the following code.

```
 1 IMPL_1:
 2 # define REJECT goto IMPL_2
 3 # define ACCEPT goto IMPL_1_COST
 4     if (node->operator != join) REJECT;
 5 # undef REJECT
 6 # undef ACCEPT
 7 IMPL_1_COST:
 8     meth_input [0] = node->oper_input[0];
 9     meth_input [1] = node->oper_input[1];
10     local_cost = total_cost = cost_hash_join (node, meth_input);
11     total_cost += meth_input [0]->total_cost;
12     total_cost += meth_input [1]->total_cost;
13     if (total_cost < node->total_cost) {
14         node->method = hash_join;
15         node->meth_input [0] = meth_input [0];
16         node->meth_input [1] = meth_input [1];
17         node->local_cost = local_cost;
18         node->total_cost = total_cost;
19         transfer = (TRANSFER) NIL;
20     }
```

The first lines are just as they are in *MATCH*. The macros *REJECT* and *ACCEPT* are defined (lines 2-3). The pattern of the query tree is matched (line 4), and the macros are undefined (lines 5-6). A local array ***meth_input*** is filled with pointers to the input streams

as they would appear in the access plan (lines 8-9). The cost function is invoked to determine the local cost, i.e. the cost that would occur in this operator, and its return value is stored in a local variable (line 10). The total cost is summed up in a small loop over all inputs (line 11-12). If the total cost compares favorably with the best cost found so far (which is stored in the node and was initialized to a constant *HUGE_COST*), the access plan provided by this implementation rule is copied into the node in consideration (lines 13-18). The pointer to the argument transfer function is set to *NIL* to indicate that no argument transfer function was provided with the rule (line 19).

□

After all rules have been processed, the generator appends the temporary files with the procedures *MATCH*, *APPLY*, and *ANALYZE* to the generated source code file which until then had only type and constant definitions in it. Subsequently, it produces two procedures called *OPERATOR_PROPERTY* and *METHOD_PROPERTY*. Each of them consists of a case statement to invoke the property functions which are associated with the operators and methods. The purpose of property functions and other support functions is described in the following section.

4.1.3. Support Functions

The third part of the model description file is optional. It can be used to append source code to the optimizer code. For instance, it is a good place to put cost functions and other support functions. It is also possible, however, to put the cost functions and the other support functions in separate files. This section describes the support functions and their purposes. For more detailed descriptions, the reader is referred to Appendix A.

A cost function is associated with each method. The name is built by the generator by concatenating the word ***cost*** and the name of the method. To find the cheapest implementation method for a query tree, the procedure *ANALYZE* matches the query against the

implementation rules, and calls the cost functions of the matching methods. The input arguments are the root node of the query tree in *MESH* and the nodes in *MESH* which produce the input stream(s). These nodes, called ***method input nodes***, are not necessarily equal to the child nodes in the query tree. They will be different if the method under consideration implements more than one operator. These pointers correspond to the data streams in the access plan.

Example 4.11: Consider the following implementation rule.

project (hash_join (1, 2)) by hash_join_project (1, 2)

This rule indicates that a special form of hash join, called ***hash_join_project***, can implement a join plus a subsequent projection operator. The function ***cost_hash_join_project*** is called with pointers to the project node and to the nodes corresponding to the inputs numbered 1 and 2 in the rule.

□

Argument transfer functions are used to assist the optimizer in maintaining and manipulating the argument fields. With each operator or method, the argument field specifies how the operator or method should be applied to the input streams. An argument field appears once in every node in the original query (for the operator) and in the access plan (for the method), and twice in every node in *MESH* (for the operator and for the method). Its type is defined by the DBI in a code section of the declaration part in the model description file. Typically, this field will have a ***union***-type.

When an argument transfer function is included in a transformation rule, this function is called to set all the argument fields in the new query tree. When this function is called, duplicates have not yet been removed from the tree. Recall that duplicates were defined as nodes with equal operators, operator arguments, and input streams. To compare arguments, the DBI must provide a Boolean function called *DIFFERENT_ARGUMENTS*.

When a duplicate is removed, the function *UNDO_NODE* is called to allow the DBI to do some housekeeping if necessary, e.g. deallocate space that a pointer in an argument field points to. In the case of an argument transfer function with an implementation rule, the function is called after the method has been determined.

By default, argument fields are copied from the initial query tree into *MESH*, between corresponding nodes in *MESH*, and from *MESH* to the final access plan. If this is inappropriate for some reason, these defaults can be overwritten. To do so, the DBI defines function names *COPY_IN* for transfers from the initial query tree into *MESH*, *COPY_OUT* for transfers from *MESH* to the final access plan, *COPY_OPERATOR_ARGUMENT* for use in transformation rules, and *COPY_METHOD_ARGUMENT* for use in implementation rules in a code section of the declaration part.

In each node in *MESH*, there are two fields called *OPERATOR_PROPERTY* and *METHOD_PROPERTY* which allow the DBI to store data dictionary information about intermediate results, for example cardinality and sort order. The types of these fields are defined by the DBI in a code section of the model description file. Information which can be derived from the operator, the operator argument, and the properties of the input stream(s) belongs in the operator properties, whereas information that depends on the chosen implementation method belongs in the method properties. These two are distinguished, as this allows the cost functions to use operator properties before method properties can be derived.

To derive the properties of *MESH* nodes, the DBI associates a property function with each operator and each method in the system. The names of these functions are the word ***property*** concatenated with the name of the operator or method respectively. The generator inserts calls to these functions into the generated code at the appropriate places in the procedure *REPLACE*.

As mentioned earlier, the DBI may wish to assist the optimizer in estimating the benefit of a transformation before the transformation is performed. A function name given in a transformation rule with the keyword ***estimate*** will be called by the optimizer when the rule has been matched successfully to anticipate the cost of a query tree after the transformation. Arguments to the benefit estimation functions are the query tree, the arguments, and the properties.

At first sight, there seems to be a lot of code for the DBI to write. Not all of the functions outlined above, however, are required. Only the cost functions are absolutely necessary. If no type for operator property fields is specified, operator property functions are not necessary. Similarly, calls to method property functions are only inserted into the generated code if a type for method property fields has been specified. Argument transfer functions and estimation functions are only used if they are indicated in a rule.

Some or all of the functionality provided by these procedures is required in any optimizer. Since they are data model dependent, they cannot be provided for the DBI. What we have done, though, and what we consider important, is to provide a framework that breaks the data model dependent code in a query optimizer into small but meaningful pieces.

We expect that our system will allow the incremental development of database systems. Hopefully, even small subsets of operators, methods and rules can be designed, implemented, and tested independently.

4.2. Optimizer Operation

In this section, we describe how the optimizer finds an access plan for a query. We only give as much detail as is necessary to understand the optimization process, and refer the reader to Appendix B for further technical details.

The generated optimizer transforms the initial query step by step, maintaining information about all the alternatives explored so far in a data structure called *MESH*. For

each query tree it contains, *MESH* also contains the cheapest associated access plan found. At any given time, there might be a large set of transformations which can be applied to query trees in *MESH*. They are collected in the data structure *OPEN*, as mentioned earlier.

The general optimization algorithm is:

```
transfer the initial query tree into MESH
while (OPEN is not empty)
        Select a transformation from OPEN
        Apply it to the correct node(s) in MESH
        Do method selection and cost analysis for the new nodes
        Add newly enabled transformations to OPEN
extract the final access plan from MESH
```

Before we describe the optimization process and the procedures involved in more detail, we will outline the principal data structures that are employed.

4.2.1. Data Structures

There are four data structures that are important during optimization. Two of them, *OPEN* and *MESH*, have been mentioned before. *EXP* is an array of **expected cost factors**, one for each transformation rule in each direction, which is used to anticipate the benefit of applying rules. *PHASE* is an array of records which contains parameters of the search for each optimization phase. If there is only one phase, this array contains only one element.

MESH is a network of nodes that represents both alternative query trees and access plans. It is anchored with a special pointer called *TOP* which initially points to the root node of the query tree. Since each node can be as large as several hundred bytes, and since there can be many query trees to consider, it was important that *MESH* be designed to avoid any unnecessary redundancy. Also, since we wish to avoid redundant processing, it seems natural to share as many nodes as possible between query trees. To achieve this, the optimizer allocates nodes only when necessary during a transformation, sharing copies whenever feasible. With this implementation, typically as few as 1 to 3 new nodes are required

for each transformation, independent of the size of the query tree. More precisely, a node is created for each operator that appears in the transformation rule on the "new" side. This process is described in the sections below on how a transformation is actually performed.

Example 4.12: Consider Figure 4.2. The bold arrows denote transformations, solid lines show the data streams (which flow upward), and dotted lines point to subtrees that are being reused. The first transformation pushes the selection down the query tree. The second transformation applies join associativity.

□

Each node in *MESH* contains a number of pointers to other nodes and to entries in *OPEN*. These will be discussed shortly when it will be more obvious why they are desirable.

OPEN is used to keep track of transformations that can be applied to query trees in *MESH* but which have been delayed in favor of other transformations. *OPEN* is organized as a priority queue and is implemented as a binary tree. Each element in *OPEN* is a record that describes a transformation by specifying the rule number and all operator nodes and input nodes involved. In addition, it contains a cost field which reflects the total cost of the query tree before the transformation, and a benefit field where the anticipated benefit is

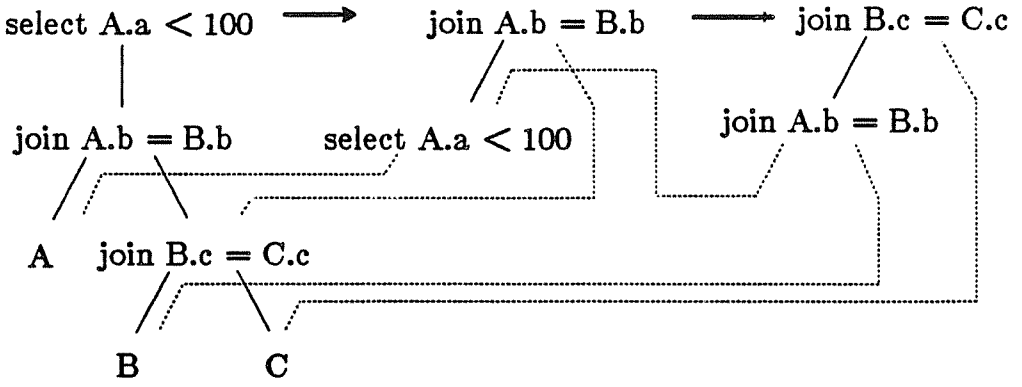


Figure 4.2.
Example Tree Transformation.

stored, i.e. the cost reduction anticipated for applying the transformation to the query tree. The elements in *OPEN* are ordered by the value of the benefit fields such that the transformation with the biggest benefit is selected first. The formula used to calculate the benefit field is described and justified in the section on the search strategy.

EXP is an array which is used in the promise evaluation. It has one entry, called the **expected cost factor**, for each transformation rule and each direction. The expected cost factors are a crucial element of the benefit calculation. Since they are hard to estimate, they are **learned** automatically by the optimizer; the approach used for this learning is also described in the section on the search strategy.

PHASE is an array of records, one for each phase in the optimization process. Each record contains values that control search and pruning in one optimization phase. The parameters and their value ranges are described in the sections on pruning and on stopping criteria.

4.2.2. Tracing a Transformation

The general algorithm of the optimizer was outlined at the beginning of this section. As long as there are transformations in *OPEN* that can be applied, one is selected, removed from *OPEN*, and applied to *MESH*. In this section, we describe in a step by step manner what happens when a transformation is applied. The management of *OPEN*, i.e. selecting and adding entries, will be treated in its own section.

After a transformation has been selected, it is passed to the procedure *APPLY* which was generated from the transformation rules. As explained above in the section on code generation, *APPLY* creates a new query tree, links the operator nodes and the input nodes together, and transfers operator arguments into the new nodes.

The optimizer then traverses the new nodes bottom-up and calls the procedure *REPLACE* for each new node. *REPLACE* tries to replace the node by an existing equal

node. Two nodes are equal if they have the same operator, the same operator argument, and the same input(s). To make the search for equal nodes fast, all nodes in *MESH* are inserted into a hash table, hashed on the operator and (the main memory address of) the child nodes. This scheme to detect equal nodes is already used when the initial query tree is transferred into *MESH*, so that common subexpressions in the query are recognized as early as possible.

If a new node can be replaced by an existing duplicate node, *REPLACE* deallocates the new node and returns the address of the equal node to *APPLY*. All pointers in the query tree that point to the replaced node are changed to point to its duplicate node.

If there is no equal node, the new node must be integrated into *MESH*. This happens in several steps. First, the operator properties are determined by calling the procedure *OPERATOR_PROPERTY*, which in turn calls the operator property function supplied by the DBI. As mentioned earlier, operator property functions and method property functions are optional. If one or both of these types are not declared, these functions calls are skipped. Next, the generated procedure *ANALYZE* determines the cheapest implementation method for the new node and determines the method arguments as discussed in the section on code generation. If the node under consideration is the root node of the new query tree and the cheapest implementation is more expensive than a predefined threshold, the entire new query tree is abandoned and removed from *MESH*. Otherwise, the node is put into the network of pointers that ties *MESH* together. Next, the method properties are determined by calling the generated procedure *METHOD_PROPERTY*. Finally, the new query tree is matched against the transformation rules by the procedure *MATCH*, and any applicable transformations are added to *OPEN*.

After the procedure *REPLACE* has been called for all new nodes, the procedure *APPLY* updates the statistics about rule applications by calling the procedure

UPDATE_RULE_STATISTICS and links the root of the newly created query tree into the transformed query tree's equivalence class by calling the procedure *NEW_EQUIVALENT*.

When a new query tree is inserted into an equivalence class, it is possible that a parent of a node in this class could now be implemented less expensively or transformed further if the new subquery were substituted for an existing member of this class. Thus, one function of the procedure *NEW_EQUIVALENT* is to match all parent nodes of the transformed subquery (those that point to the transformed subquery or an equivalent subquery as one of their input streams) against the implementation rules to propagate the cost improvement obtained by the transformation performed. We call this **reanalyzing**. In addition, the parent nodes are matched against the transformation rules, as there may now be some (new) possibilities for further transformations. This is termed **rematching**. For reanalyzing and rematching, *NEW_EQUIVALENT* calls the procedure *REANALYZE*.

Example 4.13: Consider Figure 4.3. The transformations push the selection down the query tree, reusing nodes where possible. To apply join associativity, the node labeled I must be rematched with the node labeled II as its right input, resulting in an entry in *OPEN* that will eventually lead to the transformation shown in Figure 4.4.

□

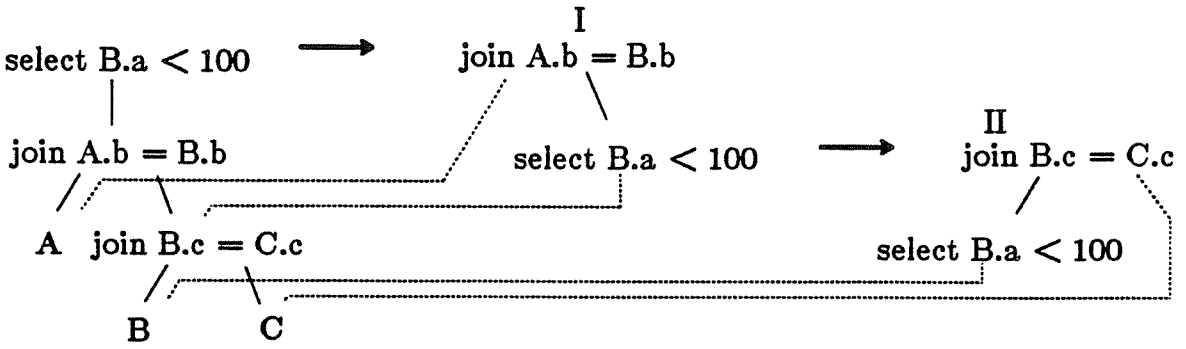


Figure 4.3.
Situation needing Rematching.

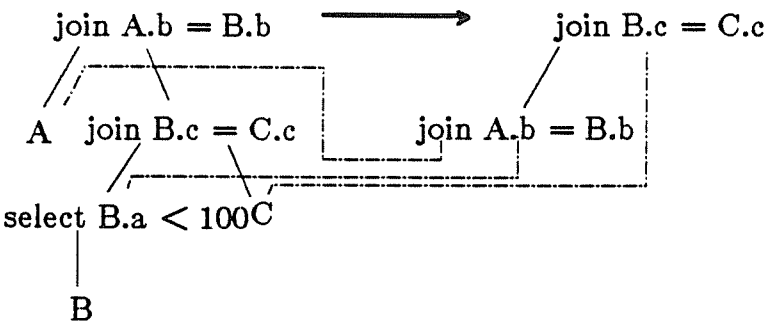


Figure 4.4.
Transformation by Rematching.

In each equivalence class, there is one designated node called the **class node** which contains pointers not found in the other nodes. First, it points to the cheapest alternative subquery in the class. Second, it serves as an anchor for a linked list of parent nodes of members of this class. To find the class node for a node quickly, each node in *MESH* contains a pointer to its class node.

For each possible parent node of the new query tree, the procedure *REANALYZE* creates a copy of the parent node with the same operator, operator argument, and input stream(s) except that the new subquery is used instead of an old equivalent subquery. The procedure *REPLACE* is called with this node, and will call all other necessary procedures like *OPERATOR_PROPERTY*, *ANALYZE*, *METHOD_PROPERTY*, *MATCH*, and *NEW_EQUIVALENT*. Finally, *REANALYZE* calls the procedure *NEW_EQUIVALENT* because the copy of the parent node belongs to the same equivalence class as the parent node itself. Notice that the procedures *NEW_EQUIVALENT* and *REANALYZE* invoke each other, i.e. there can be several levels of recursion, as many as there are levels in the query tree. In this way, cost advantages are propagated upwards in *MESH* until the *TOP* node is reached. While it is required that parent nodes be reanalyzed on all levels, rematching is only required depending on the level of recursion and the depth of nesting in the rule expression that is being matched. The level of recursion is tested for each transformation

rule before the pattern is matched, as was shown in Example 4.8.

Reanalyzing is not always worth the effort. For example, if a query has become much more expensive by replacing a subquery with a new alternative, it probably is not worth the effort. Notice that this is a heuristic, because replacing the old subquery with the new subquery might allow a very beneficial transformation to be applied to the parent node. Therefore, we introduce a factor, called the **reanalyzing factor**, that limits when reanalyzing should be done. If it is set to 1.5, for example, parents of a subquery are reanalyzed only if a new subquery is not more than 50% more expensive than the cheapest equivalent subquery.

4.2.3. Search Strategy and Learning

Since the number of possible transformations in *OPEN* can be very large for a complex query, it is critical that the optimizer avoid applying most of these transformations if such queries are to be optimized in a reasonable amount of time. To find the optimal access plan quickly, the search must be directed (Barr, 1981). To do this, the "right" transformation must be selected from *OPEN* at each step of the optimization process. The ideal situation would be to select only those transformations that are necessary to transform the initial query into the query tree corresponding to the optimal access plan. Unfortunately, this is not feasible since the optimal access plan and the shortest sequence of transformations are not known. Instead, the optimizer selects the transformation which promises the largest cost improvement. **Promise** is calculated using the current cost (before the transformation) and information about the transformation rule involved. To measure the promise of a transformation rule, an **expected cost factor** is associated with each transformation rule. Bidirectional transformation rules have two expected cost factors, one for each direction. The interpretation of this factor is as follows: if the cost before the transformation is **c** and the expected cost factor of the transformation rule is **f**, then the cost after the transforma-

tion is c*f. If a rule is a good heuristic, such as pushing selections down in the tree, the expected cost factor for the rule should be less than 1. If, however, a rule is neutral on the average, (e.g. join commutativity), its value should be 1.

The concept of expected cost factors raises two important issues. First, are such factors valid? That is, is it really possible to associate such a value with a rule independent of the database and the queries to be optimized? Second, how can these factors be determined? We will address the second question first.

We decided that it would be too difficult (and too error prone) if the DBI were asked to set the expected cost factors. On the other hand, since we do not know the data model and the rules that a future DBI might implement, we cannot provide these cost factors either. Thus, our belief is that they should be determined automatically by the optimizer by **learning** from past experience. An adequate method is to use the average of the observed cost quotients for a particular rule. Recall that the expected cost factor is an estimate for the quotient of the costs before and after applying the transformation rule. Thus, it is reasonable to approximate the expected cost factor with the observed quotients for the rule.

The simplest averaging method is to take the arithmetic average of all applications of the rule since the optimizer was generated. However, if the query pattern or the database changes, using the average of **all** observed quotients might be too rigid. One alternative would be the average of the last N applications (for some suitable N). This is fairly cumbersome to implement, however, as the last N values must be stored for each rule. A second alternative is to calculate a sliding average for each rule. The sliding average is the weighted average of the current value of the expected cost factor and the newly observed quotient, and is quite easy to implement efficiently. Finally, since we average over quotients, a geometric average might be more appropriate than an arithmetic average.

We evaluated the following four averaging formulas:

sliding geometric average $f \leftarrow (f^{K} * q)^{\frac{1}{K+1}}$	geometric mean $f \leftarrow (f^{c} * q)^{\frac{1}{c+1}}$
sliding arithmetic average $f \leftarrow \frac{f*K+q}{K+1}$	arithmetic mean $f \leftarrow \frac{f*c+q}{c+1}$

Table 4.1.
Learning Formulas.

In these formulas, f is the expected cost factor for the rule under consideration, q is the current observed quotient of new cost over old cost, c is the count of how many times this rule has been applied so far, and K is the sliding average constant.

In many cases, we will find that a beneficial rule is possible only after another (perhaps even negatively beneficial) rule has been applied. To reflect this in the search strategy, the optimizer actually adjusts the expected cost factor of *two* rules after an advantageous transformation. First, it recalculates the factor for the rule that was just applied using one of the techniques described above. This is called **direct adjustment**. Second, it also adjusts the factor of the rule that was was used to created the query tree just transformed, using the same formula but with only half the weight. We call this **indirect adjustment**. It ensures that a rule which frequently enables subsequent beneficial transformations will have an expected cost factor lower than 1 (the neutral value), and will be preferred over other neutral rules which do not provide this indirect benefit. Finally, if a cost advantage is realized while reanalyzing the parent nodes after a transformation, the transformation rule's expected cost factor is also adjusted with half the normal weight. We call this **propagation adjustment**. The weight for direct, indirect, and propagation adjustment can be set by the DBI if desired. In our experiments (which are reported in the next two chapters) we obtained satisfactory results using half the weight used for direct adjustment for indirect

and propagation adjustment.

Ordering the transformations in *OPEN* by the expected cost decrease has a negative effect in some situations. If *OPEN* contains two equivalent subqueries with different costs, each of which can be transformed by the same rule with an expected cost factor less than 1, the transformation of the more expensive query tree will be selected first. This is, of course, counterintuitive, and not a good search strategy. To offset this effect, the optimizer subtracts a constant from the expected cost factor when estimating the cost after a transformation of a part of the currently best access plan. The lowered expected cost factor increases the expected cost improvement, such that the currently best subquery is transformed before the other equivalent subquery.

The expected cost factors are used to direct the search, so the optimizer finds the optimal access plan quickly. Once the optimal access plan has been found, the optimizer could simply ignore all the remaining transformations in *OPEN* and output the plan. Unfortunately, it is impossible to know when the currently best plan is indeed the optimal one. Our solution is to let the optimizer keep searching, but to limit the set of new transformations that may be applied. To do this, the cost improvement expected by applying a transformation is compared with the cost of the best equivalent subquery found so far. If this improvement is within a certain multiple of the current best cost, the transformation is applied; otherwise, it is ignored and removed from *OPEN*. Using the analogy of finding the lowest point in a terrain, but sometimes having to go uphill to reach an even lower valley, we termed this technique **hill climbing**. The multiple mentioned above is the **hill climbing factor**. Typical values are 1.01 to 5, depending on the data model, the rule set, and the cost functions. If it is less than 1, neutral rules will never be applied, even though they might be necessary to explore the complete search space. On the other hand, the experiments described later show that, at least for the relational model, hill climbing factors

close to 1 work well.

Unfortunately, the appropriate values for the hill climbing and reanalyzing factors seem likely to depend on the data model. Thus, like the expected cost factors, they too should be learned by the optimizer. We have not yet implemented this feature, however.

4.2.4. Management of *OPEN*

OPEN is a priority queue implemented as a binary tree. Each entry in *OPEN* contains the number of a transformation rule and the query tree to be transformed. There are three basic operations performed on *OPEN*, namely insertion, selection, and adjustment of entries. We will discuss them in that order.

The procedures that insert an entry into *OPEN* and select an entry from it also prune parts of the search space by ignoring transformations. As mentioned before, pruning is necessary to avoid exhaustive search. The decision about whether a transformation is pruned or not is based on the **hill climbing limit**. The hill climbing limit is calculated as the product of the cost of the best equivalent subquery and the hill climbing factor. The hill climbing limit is defined for each equivalence class and decreases as better access plans are found.

Entries are added to *OPEN* by calls to the function *ADD_TO_OPEN* from the procedure *MATCH*. First, the cost after the transformation is anticipated using the cost before the transformation and the expected cost factor of the transformation rule. If a part of the currently best access plan is transformed, the expected cost factor is lowered as described above in the section on the search strategy. If the resulting value is within the range defined by the current hill climbing limit, the expected benefit is calculated and the entry is inserted into the binary tree. Otherwise, the entry is ignored, i.e. this transformation is pruned from the search tree.

The next transformation to be applied is selected by the procedure *SELECT_FROM_OPEN*. Before an entry is returned by this procedure, it is checked again whether it is still within the hill climbing limit, as the hill climbing limit might have decreased while this transformation was being delayed and stored in *OPEN*. In this case, the transformation is pruned and the next transformation in *OPEN* is considered.

Recall that the benefit calculation takes into account whether or not a node is part of the best access plan. When an access plan is found that is better than the best one known so far, different nodes in *MESH* should get priority in *OPEN*. Hence, when the best access plan changes, certain transformations must be found in *OPEN*, their benefit recalculated, and their positions in *OPEN* properly adjusted. To find these entries in *OPEN* quickly, all entries are also inserted into a list of transformations which is accessible from the root node of the "old" query tree in *MESH*.

4.2.5. Stopping Criteria

Our experiments with the prototype implementation indicate that a significant portion of the search effort is spent after the final access plan has been found. This is due to the fact that the optimizer has no way of determining whether the currently best access plan is indeed the final plan. Hence, it keeps transforming query trees until *OPEN* is exhausted.

We have implemented two other stopping criteria. First, it is possible to limit the number of transformations applied without improving the best access plan. Second, the size of *MESH* can be restricted to a maximum.

In the early part of the optimization process, it will be fairly easy for the optimizer to find better access plans. As more query trees are explored, it will take longer to find a better access plan than the currently best one. A graph representing the cost of the currently best access plan on the vertical and the size of *MESH* on the horizontal axis will show steep decrease on the left, corresponding to the fast succession of improvement of the

overall plan, and a flat line on the right, corresponding to the search for better plans after the optimal plan has been found. If the optimizer could determine where the flat part of the curve begins, i.e. whether the currently best plan is indeed the optimal plan, it would be possible to save a significant portion of the optimization effort. The generated optimizers approximate this by stopping when the curve has been flat for some period of time, assuming that the flat part after the final access plan has been reached. The DBI can set the length of this time period by setting a global variable. An alternative to using a number of nodes would be to use a factor. For example, if 20% of the nodes in *MESH* have been created after the best plan was found, the search is aborted.

The second way to limit is to restrict the size of *MESH* or, almost equivalently, the number of transformations. We implemented this by comparing the number of nodes in *MESH* with a limit each time before a new transformation is applied. This limit is also set in a global variable.

The global variables pertaining to the stopping criteria can be set once or they can be set for each query individually before the query tree is handed to the optimizer. More details are given in the section on tuning in Appendix B.

Query optimization in the commercial version of INGRES is halted when the time spent on optimization reaches a certain fraction of the anticipated query execution time (Kooi, 1982). Even though we believe that this is a valid heuristic, we have not implemented this criterion because it assumes that there is a way to compare the query execution cost with the optimization cost. The former, however, is defined by the DBI, while the latter is inherent in the optimizer. While it certainly would be possible to provide the DBI with a handle to limit the search by comparing the optimization effort with the anticipated execution effort, this would violate the principles that have guided us in separating DBI code and generated code. If the DBI insists on limiting the optimization by this criterion,

though, she or he can do so by setting the limit for the size of *MESH*.

4.3. Extending an Existing Optimizer

In this section, we describe briefly what is required to extend an existing optimizer implemented using the EXODUS optimizer generator. We describe what needs to be done if a new operator, a new method, a new access method, or a new abstract data type is added to the optimizer.

To add a new operator, the DBI has to extend the model description file, which was decribed in Section 4.1.1. First, the new operator must be declared in the declaration part; second, rules must be added in the rule part; and third, support functions must be written or updated.

The additions to the rule set must ensure that the extended rule set is sound and complete for the extended optimizer. The extended set of transformation rules must allow the optimizer to create all equivalent query trees from a query tree. In order to allow the optimizer to find access plans for queries involving the new operator, one or more implementation rules must be added to the model description file, associating the new operator with implementation methods. Quite likely, the new operator requires the addition of new methods to the system.

If the existing optimizer uses operator properties, a new operator property functions must be provided for the added operator. Possibly, the argument structure must be modified to be suitable for the extended operator set, and existing argument transfer functions must be updated to accomodate these changes. If the new rules refer to argument transfer functions or to benefit estimation functions, these functions also must be added to the optimizer.

Adding a new method requires that the new method be declared in the declaration part of the model description file, and that the new method be associated with operators using

implementation rules. A cost function must be provided for the new method. Update of the argument data structure and existing argument transfer functions might be required, depending on whether or not the new method can be supported with the existing argument data structure. If the existing optimizer uses method properties, a new method property function is necessary. Finally, new argument transfer functions are required if they are refered to in the new implementation rules.

Adding a new access method can only be modeled by adding a suitable set of new implementation methods to the model description. We have given some consideration to extending our model of optimization from two concepts to three concepts. Currently, there are two concepts, those of operator and of method. In order to make it more straightforward to introduce new access methods, our idea is to introduce a third concept, called ***method class***, which models common characteristics of a set of methods, e.g. ***exact match index look-up*** or ***range index look-up***. Implementation rules could either refer to a method (as in the existing optimizer generator implementation) or to a method class. A new kind of rule would associate a method with a method class, e.g. ***B-tree look-up*** with ***range index look-up***. In this three concept model, adding a new access method would require to associate a new access method with a method class, which would presumably require only a single rule. However, we have not yet designed an implementation of this idea.

Finally, to add a new abstract data type, the declaration part and the rule part of the model description file do not change at all. Recall (from Section 3.2.3.1) that abstract data types are hidden inside the argument structure. Accomodating a new abstract data type requires appropriate updates to all support functions that manipulate or inspect operator or method arguments, namely argument transfer functions, cost functions, property functions, and benefit estimation functions. If a new database management system is to support the definition of new abstract data types without recompilation and relinking of the system, the

support functions must be designed in a way flexible enough to incorporate new abstract data types instantaneously.

CHAPTER 5

Experiences with the EXODUS Query Optimizer Generator Prototype

In this chapter, we review some query optimizers that have been implemented or are being implemented using the optimizer generator. All these prototypes are based on the relational data model. This does not mean that the optimizer generator works only for relational systems. While this research is being conducted, a consensus seemed to form that new data models should include the relational model as a base. The result is that most data model proposed recently are extended relational models (Stonebraker, 1986,Manola, 1986). The kind of extensions proposed, however, are diverse, and will probably become more diverse in the future. To accomodate proposed and future extensions, we believe that a tool with the generality of the optimizer generator is the best alternative for database researchers who wish to implement their ideas.

First, we report on two optimizers that we implemented within the EXODUS project to show the feasibility of the approach. Then, we report briefly on two in-progress efforts to implement new query optimizers using the optimizer generator.

5.1. Prototypes Designed Within EXODUS

The optimizer generator was used to produce query optimizers for several, quite different, relational systems. Our first optimizer, called OPT1, uses a small rule set to optimize select-join queries. Projection was not included in the model because it seems not to be a major source of difficulty in relational query optimization. OPT1 uses 8 transformation rules and 7 implementation rules.

The input to OPT1 is a tree of select and join nodes. The operator in the leaf nodes is called ***get***. This operator was introduced for two reasons. First, it permits the definition of rules concerning the select and join operators to be defined independently of their positions in the tree. Second, its property functions perform the actual look-up in the relation and attribute catalogs. Since the leaf nodes are never transformed, they are shared among all query trees in *MESH*, which ensures that the catalog is consulted only once when the query tree is copied into *MESH* but not during the actual optimization. An example query tree optimized by OPT1 is shown in Figure 5.1.

The output of OPT1 is an access plan with the methods file scan, index scan, filter, sort-merge join, nested loops join, hash join, and index join. ***Index join*** is a nested loops join that uses an index. ***Filter*** is used for selections which cannot be included in a scan. An example access plan for the query tree in Figure 5.1 is shown in Figure 5.2. Notice that the merge-join requires sorting which is assumed to be part of the method, and that the index look-up is implicit in the index join, hence it is not modelled as an input stream in the access plan.

The selection predicates are of the form ***attribute comparison-operator constant***. As comparison operators, the six standard operators plus pattern-matching for strings are

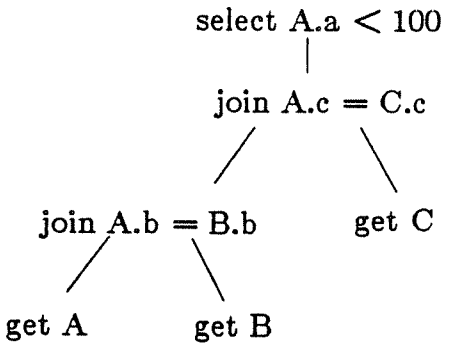


Figure 5.1.
Example Input to OPT1.

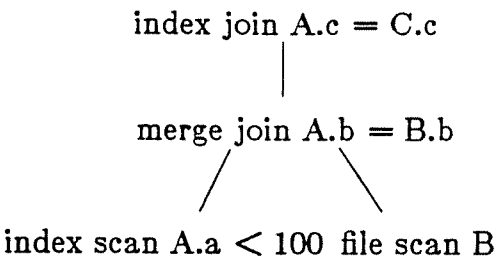


Figure 5.2.
Example Output of OPT1.

considered. The selectivity factors are estimated using the formulas used in System R (Selinger, 1979). The join predicates are of the form ***attribute*** = ***attribute***. The join selectivity can be calculated either as a fraction of the product of the input cardinalities or as a multiple of the smaller of the input cardinalities, depending on a run-time switch. In the experiments reported later in this chapter, we calculated the join selectivity as a fraction of the product of the input cardinalities.

The second optimizer, called OPT2, is used to support a user interface similar to QUEL (Stonebraker, 1976). It uses 18 transformation rules and 15 implementation rules. The user interface produces a tree consisting of a projection, a selection, and several Cartesian product nodes. The selection predicate consists of the complete ***where*** clause, hence it can be quite large. If there is an aggregate function in the qualification, appropriate aggregation operators are inserted into the query tree (Klug, 1982). The initial query tree for the query

retrieve (A.a, B.b, C,c) where
A.b = B.b and A.c = C.c and C.d = min (B.d by B.b)

is shown in Figure 5.3. The operator aggregate join performs the aggregation, the subsequent join on the by-list, and tests the condition on the calculated aggregate.

This optimizer operates in three phases. The first phase tries to transform the product nodes into equi-join nodes by breaking the selection predicate into smaller predicates. The second phase rearranges the tree such that projections will be done as early as possible. A

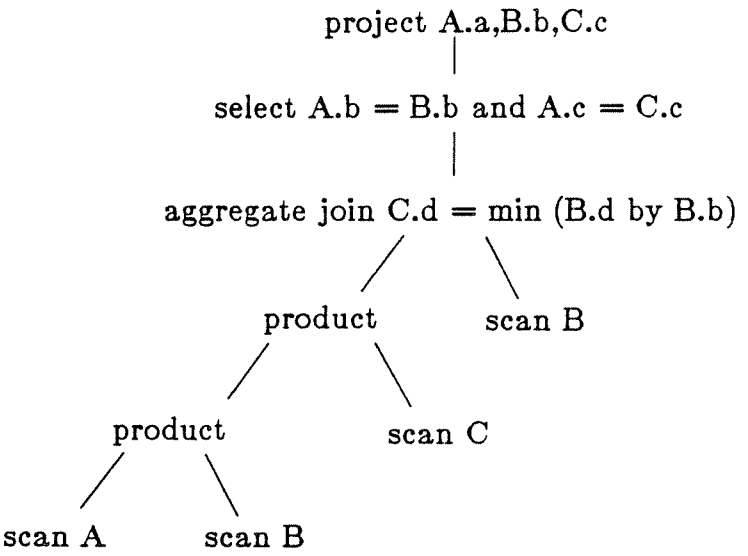


Figure 5.3.
Example Input to OPT2.

projection operator is assumed to include duplicate elimination. The third phase is the actual optimization phase. Its task is a superset of the work of our first optimizer, OPT1.

The predicates are stored and manipulated in the form of parse trees. The argument fields in the query trees, in *MESH*, and in the access plans are pointers to the root of these trees. This design makes full use of the flexibility provided by argument transfer functions and the macros *COPY_ARGUMENTS*, *COPY_IN*, and *COPY_OUT*.

5.1.1. Implementation

Implementing the relational optimizers was not particularly hard. Even though the generator itself and the library programs for the search strategy were still under development when OPT1 was built, it took only a few days to get the first version of the optimizer running. The modularization concept proved to be an important asset in the implementation effort.

For the transformation rules in OPT1, we used a subset of the rules given by Ullman (Ullman, 1982). We selected select and join commutativity, four join associativity rules and

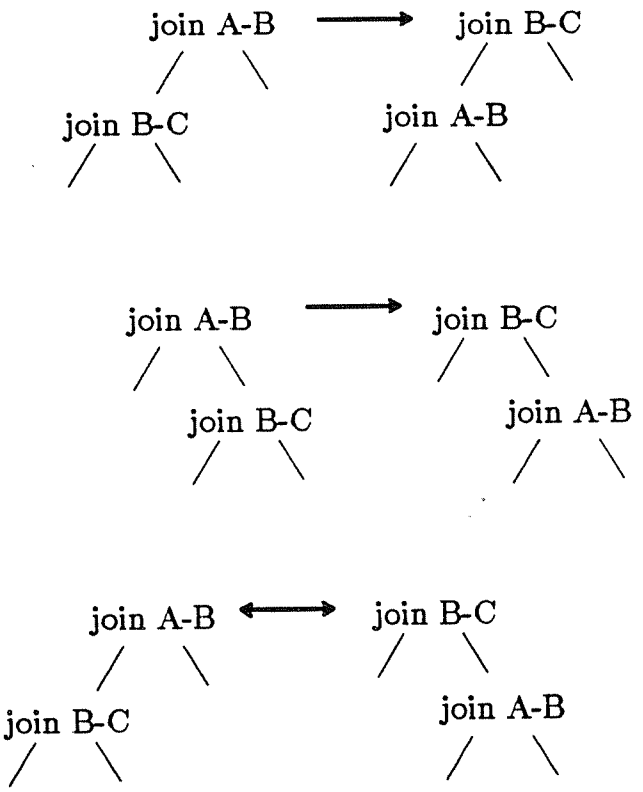


Figure 5.4.
Join Associativity Rules.

the select-join rule. The join rules are shown in Figure 5.4, and are called the left-shift rule, the right-shift rule, the left-to-right-shift rule, and the right-to-left-shift rule. The select-join rule allows the optimizer to push a select operator beneath a join and vice versa, and is shown in Figure 5.5.

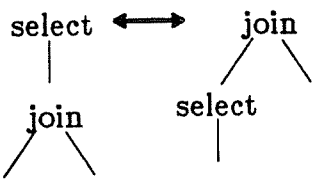


Figure 5.5.
Select-Join Rule.

For the implementation rules, we drew from the experience from WiSS (Chou, 1985) and from GAMMA (DeWitt, 1986). We considered file scan, index scan, sort-merge join, nested loops join, hash join, and index join. Index join is a nested loops join that finds matching tuples using an index on the inner relation. It requires only one input stream, and the other input of the corresponding join operator must be a *get* operator. For the other join methods, the inner relation is modeled as the right input. A method called *filter* was introduced which implements the operator *select* at any position in a query tree by applying the predicate to each tuple of the input stream. This operator typically does not occur in the final access plan for a query. It is necessary to include it in the optimization model in order to allow the optimizer to find access plans with realistic cost estimates for all correct query trees. If the optimizer cannot find an access plan for a query tree, the cost is assumed to be infinite, which prevents the optimizer from considering this query further.

The cost model for OPT1 includes I/O cost and CPU cost. I/O cost is split into sequential I/O and random I/O with different seek times. CPU cost is divided into key comparison and copying costs. The cost function for hash join further distinguishes inserting a tuple into the hash table and probing the hash table with a tuple. Passing data from one operator to another is considered to have negligible cost, as would be the case if data transfer between operators were implemented by passing pointers into the buffer pool. The cost estimates anticipate elapsed seconds for query execution on a 1 MIPS computer with a disk drive, and no overlap of CPU and I/O activity. Each cost function calculates these costs separately and then computes a weighted average. The weights were estimated using our experience and measurements from GAMMA (DeWitt, 1986, Gerber, 1986).

The argument data structure is a union type of a file name for the leaf operator *get* and simple predicates which consist of one or two variables and a comparison operator. Relation attributes are represented by pointers into the database catalog which is kept as

an in-memory data structure. The only argument transfer function in the system reverses the join predicate in the join commutativity rule. The default used for the other rules is to copy the argument records between corresponding nodes.

The operator properties include relation cardinality, arity, tuple width in bytes, and the list of attributes. The only method property considered is sort order which is used to determine whether or not sort-merge join must sort an input prior to the merge phase. The property function for ***get*** is the only function to search the catalogs using relation and attribute names. The property functions for join and select use the pointers which were found by ***properties_get*** when deriving the schema for internal nodes of the query trees. Since ***properties_get*** is called only while the initial query is copied into *MESH*, the time spent searching the catalogs is minimized. The other operator property functions consist only of a few calculations for the relation descriptor and pointer copying for the attributes. Deriving the sort order of the output stream from the sort orders of the input stream(s) and the method chosen is trivial for the implementation methods under consideration. In summary, the property functions are very short and fast.

From the implementation standpoint, the second optimizer, OPT2, was more challenging. The main reason is that the arguments to the operators and methods are significantly more complex. This optimizer allows AND's and OR's in its predicates, which are stored as a parse tree. Naturally, the argument transfer functions for transformation rules in this model are much more involved.

We realize that some will consider it a shortcoming of the EXODUS optimizer generator that it does not provide sufficient support for predicates. However, support for predicates can only be provided at the cost of generality of the argument data structure. Predicates are a special form of operator and method arguments. Yet, the argument data structure must be defined by the DBI, since it depends on the set of operators and methods in the

data model. Furthermore, it is not clear how general the logic governing the predicates in the data model should be. For statistical or intelligent databases, it might be required to provide predicates with probabilities and fuzzy logic (Gelenbe, 1986, Ghosh, 1986). It is not clear on which logic predicate support should be based in order to be sufficiently general yet efficiently implementable. Similarly, it is not clear whether or not (and how) to include NULL values. We believe that, in spite of the limited support for predicates, the optimizer generator makes it much easier to implement an optimizer, because the search engine is separated from the data model, and because a framework for a modular design of the rules and the argument transfer functions is provided.

Some of the transformation rules in OPT2 are specifically designed to break up complex *select* predicates, and to convert them into equi-join predicates. Since the initial tree contains only one predicate in the selection close to the top of the tree, such transformations make up the main part of the phase one rules.

Example 5.1: Consider the following transformation rule.

```
select 9 (product (1, 2)) -> join (select (1), select (2)) to_join
{{
   if (nojoin_predicate (OPERATOR_9.oper_argument))
      REJECT;
}}
```

This rule eliminates a Cartesian product by breaking the selection predicate into a join predicate and two selection predicates. This rule is applicable only if the select argument includes an equi-join clause, which is verified by the function ***nojoin_predicate***. Since the predicate transfer is non-trivial, a special argument transfer function (called ***to_join*** in this example) must be given with this transformation rule.

□

The argument transfer functions for these rules not only move the comparison clauses of the predicates into one of the new predicates, but they also try to optimize the resulting predi-

cates by removing redundant or constant terms.

Example 5.2: Figure 5.6 shows an original predicate tree as could have been found in the last example. Figure 5.7 shows one of the selection predicates resulting from the split. This predicate is then optimized by removing redundant terms, as shown in Figure 5.8. The other select predicate and the join predicate are built and optimized similarly.

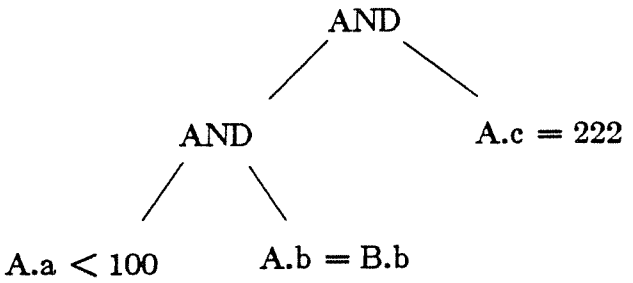


Figure 5.6.
Original Predicate Tree.

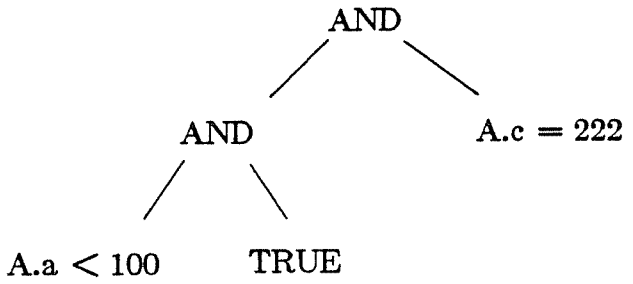


Figure 5.7.
Predicate After Split.

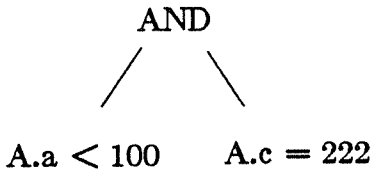


Figure 5.8.
Predicate After Optimization.

□

Some of the selection predicates generated with these rules are trivial, ie. the predicate consists of a single constant *TRUE*. Such selection operators can be removed from the query tree. This is done by the rule in the following example.

Example 5.3: The following rule eliminates redundant select operators.

```
select 9 (1) -> 1
{{
   if (nontrivial_predicate (OPERATOR_9.oper_argument))
      REJECT;
}}
```

When this rule is activated, no new node is created. The effect of this rule is that the parent nodes of redundant ***select*** nodes are reanalyzed and rematched with the ***select*** node eliminated.

□

Besides transforming selections and Cartesian products into joins, phase one of the optimization process in OPT2 also moves selections as close as possible to the leaves of the query tree. Phase two moves the projects as far down in the tree as possible. For each of the operators ***join*** and ***select*** there is a rule that moves the projection information beneath the operator, as shown in the next example.

Example 5.4:

```
project (select (1)) ->! project (select (project (1))) move_project;
```

The argument transfer function ***move_project*** proceeds in two steps. First, it copies the projection list to both of the new project operators and the selection predicate to the new select operator without change. Afterwards, those attributes that appear in the selection predicate but not in the projection list are added to the projection list of the project operator closest to input 1. This rule was coded as once-only rule because the final tree matches the rule, but a second application of the rule would provide no additional benefit. If this

rule were not a once-only rule, it would be applied two times to each eligible query tree, but the second time the resulting nodes would be detected to be duplicates and be eliminated immediately after creation.

□

Some of the ***project*** operators created by these rules can be redundant. These operators are removed from the query tree with a rule similar to the one for redundant selections.

After the projections have been moved as far down in the query tree as possible, phase three does the main part of the optimization by reordering ***select*** and ***join*** operators. This is done with the same rules used in the optimizer OPT1 described above.

The catalogs in OPT2 are maintained similarly to OPT1. To accomodate the use of QUEL range variables, there are two sets of catalogs. The first set is exactly equal to the one in OPT1. The second set stores only entries for active range variables, with the relation names replaced by the range variable names. The attribute entries in intermediate schemas and in arguments point to records in the second set of catalogs. This existence of two sets of catalogs does not have an effect on the optimizer. Inserting and replacing entries in the second set of catalogs is done by the user interface when range variables are declared.

The operator properties and the method properties in this optimizer are the same as the ones used in OPT1. The only difference in the operator property functions is the more complex argument structure, which includes more intricate formulas for selectivity factors. We modeled selectivity estimation following the design of System R (Selinger, 1979).

In summary, the implementation of these optimizers was fairly straightforward using the optimizer generator. The design of the rule sets was easy, and the support functions (cost functions, argument transfer functions, property functions) are all small and well-defined in their purpose. Writing these functions was tricky at times because they operate

directly on *MESH*. The nodes in *MESH* include both fields used by the search engine and fields defined and operated upon by the DBI functions. This requires some discipline in coding the support functions. On the other hand, dividing the nodes into two data structures and keeping each pair consistent would most likely have introduced too much processing overhead into the system.

We believe that most of the problems encountered stem from the fact that we did not perform a complete database implementation. If the data model and the run-time system are well-defined, many of the design decisions for the optimizer are fixed (e.g. the cost functions and the form and representation of predicates).

5.1.2. Experimental Databases and Queries

We experimented with the optimizer using two databases. The data for these databases did not really exist, as only their catalogs were required to run the optimizer. The databases were identical except for the relation cardinalities. There were 26 relations with 4 attributes each. The first 3 of these attributes were integers, and the last one was a string of 88 bytes, giving a total record length 100 bytes. There was a B-tree index on the first integer attribute.

The cardinalities of all relations in the first database was randomly chosen from a normal distribution with mean 1,000 and standard deviation 100. The cardinalities ranged from 802 to 1,219. With very similar cardinalities, many of the join orders resulted in equal cost, and it was difficult for the search algorithm to distinguish access plans by their cost. In the second database, the logarithms (base 10) of the cardinalities were randomly chosen from a normal distribution with mean 3 and standard deviation 1. The relation cardinalities in this database ranged from 60 to 57,549. Varying cardinalities made it more difficult for the optimizer to reliably estimate expected cost factors for the transformation rules.

The queries used in our experiments were generated by a recursive procedure. To generate a query tree, the operator at the root node was selected at random. The probabilities for both *join* and *select* were 0.3, while for *get* it was 0.4. In the case of join or select, the procedure called itself recursively to generate their child subtree(s). Finally, the argument was determined as follows. For the get operator, a relation name was selected from the relations not yet participating in the query. Using a relation twice in a single query would have required that the condition code for rules could distinguish which usage a certain attribute in a predicate came from. To avoid this problem, range variables were introduced in QUEL (Stonebraker, 1976). The concept of range variables was used in our optimizer OPT2.

The argument for a select operator was a comparison of a randomly selected attribute with a constant using a randomly selected comparison operator. A join predicate was an equality clause between two attributes randomly selected from the schemas of the two input streams. Random in this paragraph means equal probabilities for all possible values.

It was practical to build the input trees before the arguments because in the recursive generation procedure, we used the operator property functions coded for the optimizer to derive the intermediate schemas; this made it very easy to select attributes for the predicates randomly from the set of feasible attributes.

5.1.3. Validation of Expected Cost Factors

At the core of the search strategy provided with the optimizer generator are the expected cost factors. They are used to direct the search. An expected cost factor is associated with each transformation rule in the system. The benefit of a transformation can be assessed before the transformation is actually performed by anticipating the execution cost of the transformed query using optimal method selection.

The expected cost factors are learned by the optimizer. Before the first query is optimized, all expected cost factors are set to 1, the neutral value. With each application of a transformation, the repective cost factor is updated using an averaging formula. We tried the four formulas described in Section 4.2.3 and repeated in this table.

sliding geometric average	geometric mean
$f \leftarrow (f^K * q)^{\frac{1}{K+1}}$	$f \leftarrow (f^c * q)^{\frac{1}{c+1}}$
sliding arithmetic average	arithmetic mean
$f \leftarrow \frac{f*K+q}{K+1}$	$f \leftarrow \frac{f*c+q}{c+1}$

Table 5.1.
Learning Formulas.

In these formulae, f is the expected cost factor for the rule under consideration, q is the current observed quotient of new cost over old cost, c is the count of how many times this rule has been applied so far, and K is the sliding average constant.

We generated 25 sets of 400 queries each using the query generation procedure described above with different random seeds for each sequence, and optimized them four times with the four averaging formulas. Some of these queries were trivial, but many of them contained several join and select operators. All queries together included 9,226 join operators and 15,004 select operators. The database was the database described earlier with relation cardinalities varying from 60 to 57,549.

If there really is an expected cost factor independent of the queries, the 25 values collected for each of the 6 transformation rules and 4 averaging formulas should be independent from the query set. The left shift and right shift rules of Section 5.1.1 were omitted because the equivalent query tree transformations could be performed using the join commutativity and associativity rules. The rule set contained some redundancy, though, in that only one form of the join associativity rule is required if the join commutativity rule is

provided.

Due to the randomized process and arithmetic precision, they expected cost factors will not be exactly the same, but they will fall in a normal distribution around the true value. To test this hypothesis, we used a χ^2 test, comparing the observed distributions with the normal distribution with the same mean and variance. The range -∞ to +∞ was cut into 12 slices with equal expected frequencies. The breakpoints between slices were determined as quantiles of the distribution density function, i.e. the width of slices around the mean was much smaller than at the tails of the density function. The number of slices was chosen to be as large as possible without making the expected frequencies in each slice too small. From the observed (N_i for the i^{th} interval) and expected frequencies (np_i) in the 12 slices, a χ^2 value was calculated as

$$\sum_{i=1}^{12} \frac{(N_i - np_i)^2}{np_i}$$

and compared with the χ^2 table for 12—1 degrees of freedom (one less than the number of slices). This procedure is recommended in (DeGroot, 1975) and was performed using the S statistical software package (Becker, 1984).

Table 5.2 shows the levels of confidence (ranging from 0 to 1, where higher values represent higher confidence) with which we can reject the hypothesis that the values have a

Formula	SC	SJ-D	SJ-U	JC	JA-LR	JA-RL
Slid. Geom.	0.589	0.992	0.885	0.901	0.039	0.970
Geom.	0.181	0.851	0.385	0.866	0.297	0.385
Slid. Arithm.	0.810	0.473	0.751	0.027	0.828	1.000
Arithm.	0.385	0.487	0.911	0.113	0.022	0.436

Table 5.2.
Confidence of Rejection after 25 Times 400 Queries.

normal distribution. The abbreviations for the rules used in the top row and the abbreviations for the averaging formulas used in the left column are explained in Tables 5.3 and 5.4.

Using a 0.95 level of significance, we reject the hypothesis that the values follow a normal distribution for two of the rules for the sliding geometric and one for the sliding arithmetic average formulas, namely those where the value indicated is larger than 0.95. We do not reject the hypothesis for the other averaging formulas. Notice that rejecting does not mean that these averaging formulas are "wrong", it only means that the experiment does not support our hypothesis for the weights used for direct, indirect, and propagation adjustment, and for the length of the query sequence tested.

In a second experiment, we used again 25 times 400 queries, but ensured that each query included at least one select or join operator. These queries contained 15,235 join and 21,473 select operators. The result of the experiment was virtually the same; for almost all of the expected cost factors and rules we cannot reject the hypothesis of normal distribution.

SC	Select Commutativity
SJ-D	Select Join Rule, push select down
SJ-U	Select Join Rule, push join down
JC	Join Commutativity
JA-LR	Join Associativity, left-deep to right-deep
JA-RL	Join Associativity, right-deep to left-deep

Table 5.3.
Abbreviations for Rule Names.

Slid. Geom.	Sliding Geometric Average, K = 2,000
Geom.	Geometric Average
Slid. Arithm.	Sliding Arithmetic Average, K = 2,000
Arithm.	Arithmetic Average

Table 5.4.
Abbreviations for Averaging Methods.

We tested this hypothesis a third time with more, shorter sequences of queries. Notice that the shorter a sequence of queries is, the harder it is for the optimizer to approximate the expected cost factor close to the hypothetical "correct" value. We used 50 sequences of 200 queries each. All queries together contain 9,955 joins and 15,598 selects. We tested a sequence of 50 values by cutting the density curve into 25 slices. The table for this experiment is given below.

Formula	SC	SJ-D	SJ-U	JC	JA-LR	JA-RL
Slid. Geom.	0.647	0.760	0.990	0.915	0.538	0.976
Geom.	0.131	0.780	0.480	0.672	0.538	0.647
Slid. Arithm.	0.965	0.567	1.000	0.831	0.991	1.000
Arithm.	0.197	0.015	0.621	0.538	0.798	0.043

Table 5.5.
Confidence of Rejection after 50 Times 200 Queries.

It turns out that more values are to be rejected, but the majority is still on an acceptable level. In particular, only values for sliding averages are above the significance level. The reason could be that in this experiment, the parameter K of the sliding averaging process was set to 1,000 as compared to 2,000 in the first experiment, thus improving adaptability of the learning process while making the resulting value more susceptible to errors. The added adaptability was needed because the query sequences were shorter.

For all of the following experiments, we used geometric averages as the adjustment formula because this formula seemed to be the most reliable in all of the validation experiments.

5.1.4. Performance

In this section, we will present some performance figures to show that the optimizers generated with the optimizer generator find the optimal plans reasonably fast. We will not claim that the optimizers are extremly fast; we believe that optimizer speed is only of limited importance compared to the quality of the access plans produced.

We used only OPT1 for these performance measurements. The main reason is that the first two phases of OPT2 are very fast and we found it easier to generate test queries automatically for OPT1.

The access plan execution costs reported in this section are given as estimated by the cost functions. As mentioned earlier, we did not implement a complete database system. The cost measures used have been defined using our experiences with the database machine GAMMA (DeWitt, 1986, Gerber, 1986). We believe that comparing our cost measure exactly with an implementation would not verify the optimizer or the validity of the optimizer generator approach. Rather, it would only verify the formulas and the constants used in the cost functions, which are not the focus of our work.

5.1.4.1. Comparison with Exhaustive Search

Since we felt it was necessary to assess how good the access plans generated by our optimizers really are and since we had decided not to compare the chosen access plans and their alternatives with an existing database system, we needed another measure. We decided to explore how much additional effort spent on optimization affected the quality of the resulting access plan. The higher we set the hill climbing and the reanalyzing factors, the closer the chosen search strategy comes to being an exhaustive search over the entire range of possible access plans provided by the transformation and implementation rules. There are several reason why we believe that comparing our search strategy with exhaustive search is a more appropriate procedure for determining the quality of our search algorithm than a comparison with an implemented query evaluation system would be. First, we can compare the two approaches on an equal basis. The optimizer generator and the generated optimizer can use only the rules specified in the model description file. We are interested how effectively and efficiently the provided search algorithm chooses an access plan in the search space defined by the rules. Hence, we should use the same rules and

search space for the purpose of comparison. Second, we are not interested in verifying our cost formulas and constants as they are specific for our use of the generator.

Unfortunately, for some queries exhaustive search requires more resources than we were able to provide. In particular, for very large queries, *MESH* and *OPEN* can grow larger than the main memory of the machines we used for our experiments. Relying on virtual memory is highly impractical, because the search for duplicate nodes in *MESH* is based on hashing, thus providing no locality of reference. Therefore, we had to abort some exhaustive searches prematurely. However, since the optimizer applies the most promising transformations first, the transformation not applied due to preemption is not expected to have a very serious effect on the quality of the access plan.

The database used in this experiment is the contains 26 relations with cardinalities varying from 60 to 57,549. The query set consists of 1,000 queries with up to 5 joins per query. There are 2,130 joins and 2,811 select operators in all queries.

In the following tables, we compare the optimization effort as measured in the average

	1.01	1.02	1.05	1.1	1.2	1.5	2	5	10	1000
1.01	7.87	8.01	8.24	8.67	10.48	13.96	16.96	19.23	19.47	20.56
1.02	7.87	8.14	8.27	8.52	10.64	14.39	17.74	19.94	20.87	21.02
1.05	7.87	8.14	8.16	8.64	10.56	14.98	18.43	20.48	21.60	22.68
1.1	7.87	8.14	8.16	8.97	13.75	17.07	20.47	21.54	24.35	25.11
1.2	7.87	8.14	8.16	8.97	15.20	22.30	24.84	22.49	26.97	26.29
1.5	7.87	8.14	8.16	8.97	15.20	26.61	30.17	29.03	27.01	28.73
2	7.87	8.14	8.16	8.97	15.20	26.61	35.21	34.53	31.43	36.40
5	7.87	8.14	8.16	8.97	15.20	26.61	35.21	33.41	36.45	38.00
10	7.87	8.14	8.16	8.97	15.20	26.61	35.21	33.41	37.40	35.02
1000	7.87	8.14	8.16	8.97	15.20	26.61	35.21	33.41	37.40	34.98

Table 5.6.
Average *MESH* Size.

	1.01	1.02	1.05	1.1	1.2	1.5	2	5	10	1000
1.01	8.17	9.29	11.08	10.64	11.75	13.34	14.68	16.07	16.84	17.09
1.02	8.58	9.29	11.38	10.25	11.84	13.48	15.69	17.19	17.09	17.46
1.05	7.98	9.98	11.41	11.48	11.05	14.41	16.35	16.62	17.90	17.71
1.1	7.77	9.95	11.78	12.30	13.15	15.37	17.44	17.11	18.09	17.82
1.2	8.21	9.90	11.24	12.49	14.08	17.00	19.14	17.46	20.07	18.26
1.5	8.39	9.98	11.29	12.19	14.52	18.56	20.14	19.63	19.21	20.04
2	8.05	10.06	11.40	12.58	14.30	18.86	23.65	23.97	20.37	24.77
5	8.05	10.12	11.65	12.60	14.25	18.83	23.29	21.71	21.81	24.02
10	8.65	9.96	11.67	12.09	14.44	18.48	23.42	20.80	22.55	22.52
1000	8.50	9.79	11.23	12.71	13.35	18.42	24.71	21.59	23.13	21.07

Table 5.7.
Average CPU Time.

	1.01	1.02	1.05	1.1	1.2	1.5	2	5	10	1000
1.01	10.39	10.39	5.63	5.66	5.67	10.43	5.66	5.60	5.60	5.60
1.02	10.39	10.39	5.63	5.66	5.66	5.66	5.66	5.60	5.60	5.64
1.05	10.39	10.39	10.39	5.66	5.67	10.43	5.66	5.60	5.60	5.61
1.1	10.39	10.39	10.39	10.38	10.43	5.67	5.67	5.60	5.65	5.65
1.2	10.39	10.39	10.39	10.38	5.66	5.60	5.66	5.60	5.60	5.60
1.5	10.39	10.39	10.39	10.38	5.66	5.60	5.66	5.60	5.60	5.61
2	10.39	10.39	10.39	10.38	5.66	5.60	5.66	5.60	5.65	5.65
5	10.39	10.39	10.39	10.38	5.66	5.60	5.66	5.65	5.61	5.61
10	10.39	10.39	10.39	10.38	5.66	5.60	5.66	5.65	5.61	5.65
1000	10.39	10.39	10.39	10.38	5.66	5.60	5.66	5.65	5.61	5.61

Table 5.8.
Average Plan Execution Cost.

size of *MESH* (Table 5.6) and the CPU time[1] spent on query optimization (Table 5.7) with the cost of the final access plans as anticipated by the cost functions (Table 5.8). We varied the optimization effort by varying the hill climbing and reanalyzing factors. The hill climbing factor is indicated in the top row, and the reanalyzing factor in the left column.

It does not make sense to set the hill climbing and reanalyzing factors to a value less than 1, as this would inhibit cost-neutral rules such as join commutativity, which we know

[1] The optimizations were performed on a VAX 8600 from Digital Equipment Corporation, which is rated at about 6 MIPS. The times were measured using the UNIX "getrusage" system call, which allows users to inquire about the CPU time used in user mode as measured by the operating system.

are essential to finding good access plans. A hill climbing factor and reanalyzing factor of 1,000, on the other hand, indicates exhaustive search of all query trees and access plans that can be generated with the given rule set. We tried the values 1.01, 1.02, 1.03, 1.05, 1.10, 1.20, 1.50, 2.0, 5.0, 10, and 1,000 for each of the two search factors.

Tables 5.6 and 5.7 show that the search effort grows with increased search parameters. Comparsion of the smallest and largest values in these tables indicate that the effort spent on searching query trees and access plans can vary by as much as a factor of 5. If more memory would have been used, the differences would probably been even more significant. 345 of the 1,000 queries were preempted due to memory restraints[2] when the search factors were both set to 1,000, whereas only 5 when the search factors were set to 1.01. The plan execution costs, shown in Table 5.8, decrease with increased search factors. Interestingly, the plan execution costs do not decrease smoothly as the search factors decrease; rather, there are distinct thresholds above which the optimal plans are found. The thresholds for hill climbing and reanalyzing factors are dependent on one another and can substitute for one another to some extent. In general, however, it is required that both search factors are set sufficiently high in order to find the optimal access plan.

In Tables 5.6 and 5.7, it is obvious that a larger reanalyzing factor has only a small effect if the hill climbing factor is low, but has a significant effect for large hill climbing factors. The reason is that reanalyzing is only done after a transformation; if most of the possible transformation are prevented by the hill climbing limit, there is no need or opportunity for reanalyzing.

The best way to look at Table 5.8 is in terms of regions rather than of single values. In the top left region where both search factors are very low, the optimal access plans are

---

[2] We preempted optimization when the data space of the optimizer had grown to 3.5 Megabytes.

missed and the anticipated plan execution cost is high. In the bottom right region, less of the search space is pruned, and the optimal access plans are always found. There are, however, some exceptions, indicated by values over 10 in places in the table surrounded by values under 6. We hypothesize that in these cases, query trees with moderate cost were found early in the optimization process, preventing (through the hill climbing and reanalyzing limits) some transformations that would subsequently have lead to the optimal access plan.

For our optimizer, we see that the generated query optimizer is fast enough to be used in production systems. Furthermore, we have learned that it is not a good idea to set the search factors too low as the optimal access plan will frequently be missed. On the other hand, setting the search factors too high, i.e. higher than 5, does not improve the resulting plans.

5.1.4.2. Handling of Large Queries

The experiments in the last section were affected by the fact that query optimization must be aborted when *MESH* and *OPEN* grow very large. In the experiments reported above, this happened only in the exhaustive search case. If the initial query is very complex, however, it can also happen in a fairly restricted search. In this section, we report some results with complex queries.

When considering premature abortion of query optimization, it is important to realize that is not as disastrous as it seems at first. Typically, it takes a large number of transformations before main memory becomes scarce. Up to that time, the optimizer chooses the most promising transformations. There is thus a fairly good chance that the best plan possible for a query has already been found when optimization is aborted, and that aborting the optimization does not have a significant effect.

In this section, we report on how the optimizer performs for large queries. We generated 14 sets of 200 queries each. These queries had no select operators. We decided on this design because join tree reordering is the major difficulty in relational optimization. In the i^{th} set, there were i join operators in each query, i.e. i+1 relations participated in the query. The database chosen for this experiment was described earlier; the relation cardinalities ranged from 802 to 1,219. Note that using similar cardinalities makes it harder for the optimizer to improve the access plans significantly, thus making it harder for it to prune portions of the search space.

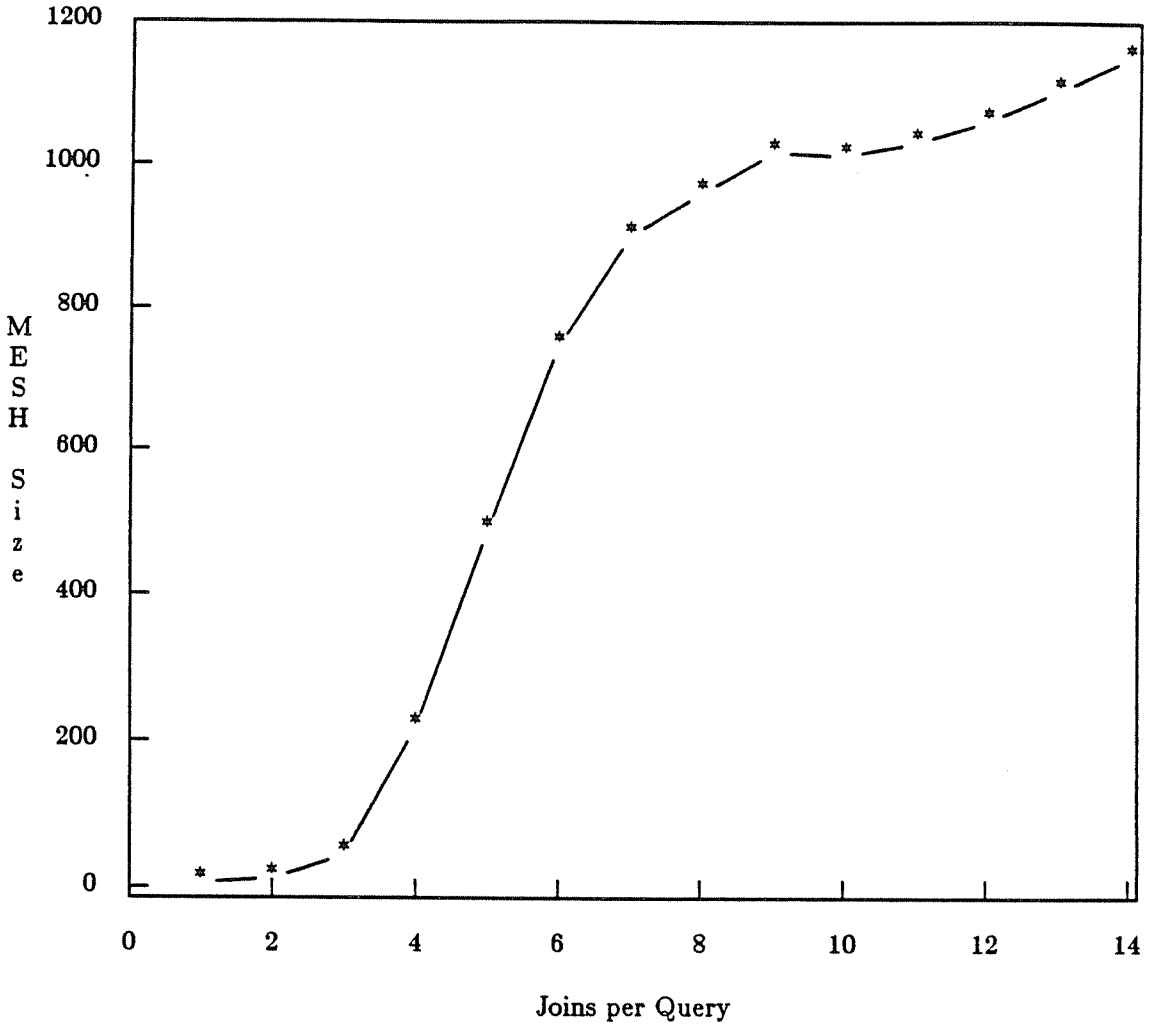


Figure 5.9.

Average *MESH* Size.

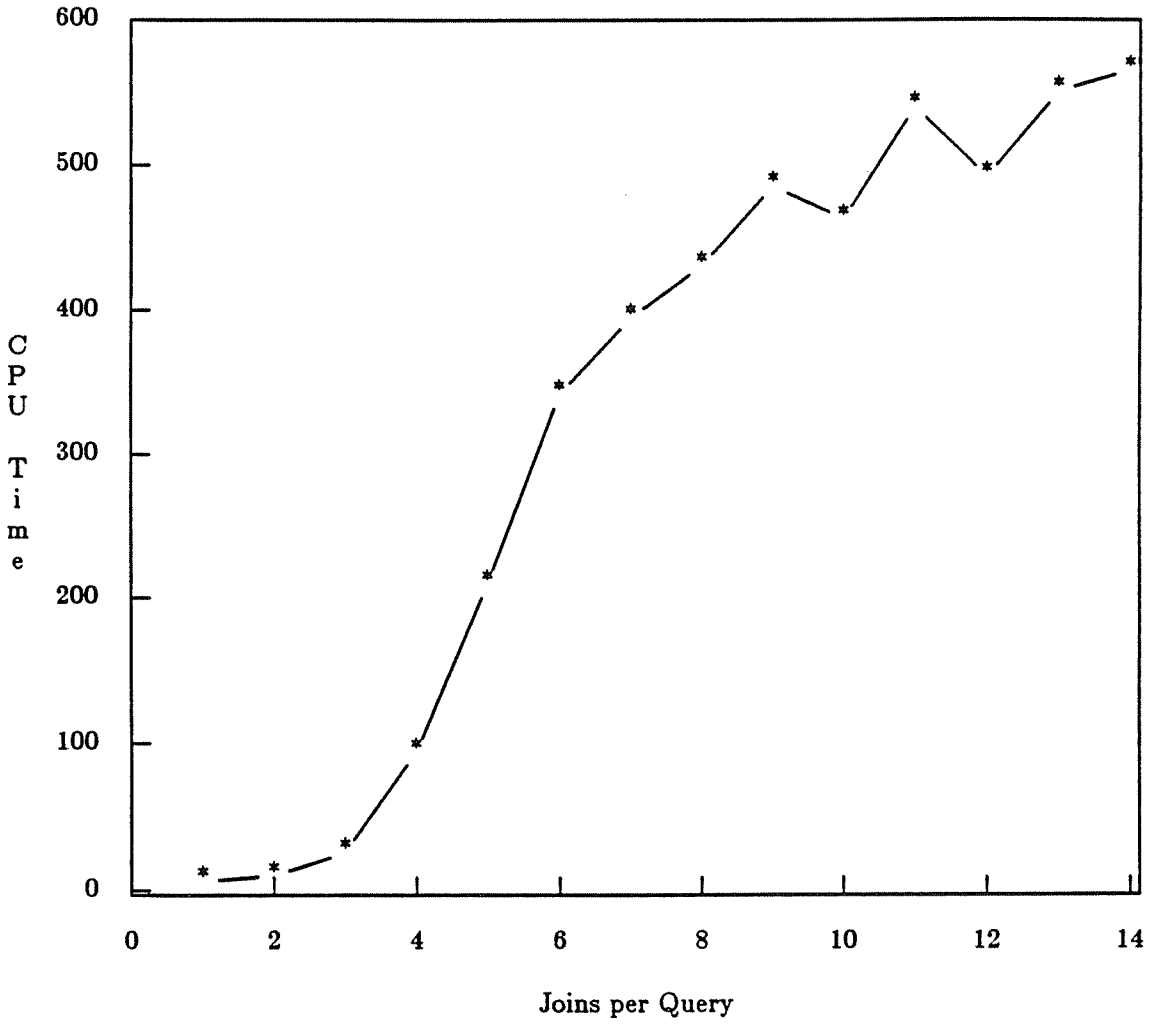


Figure 5.10.

Average CPU Time.

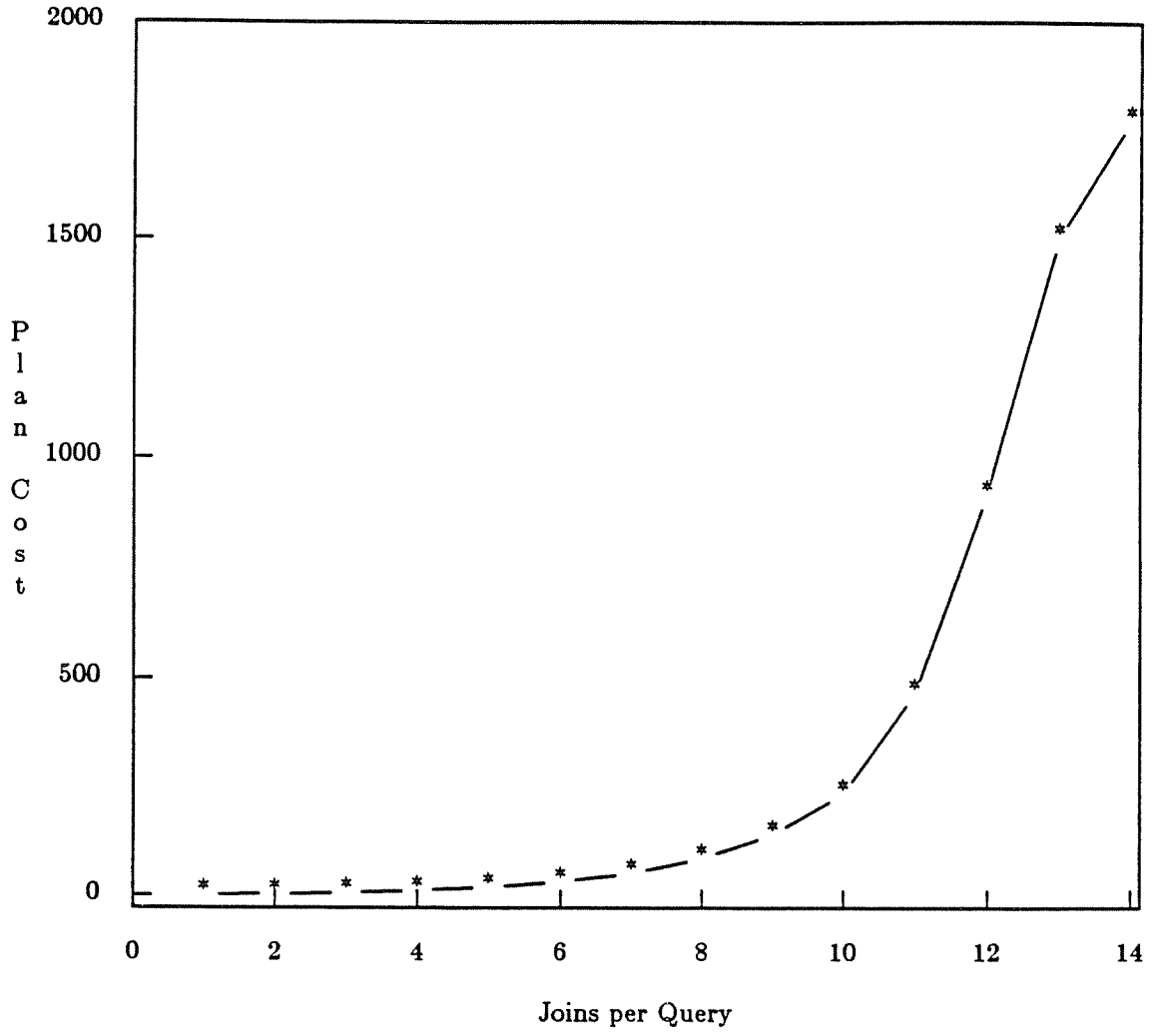


Figure 5.11.

Average Plan Execution Cost.

(in seconds processing time)

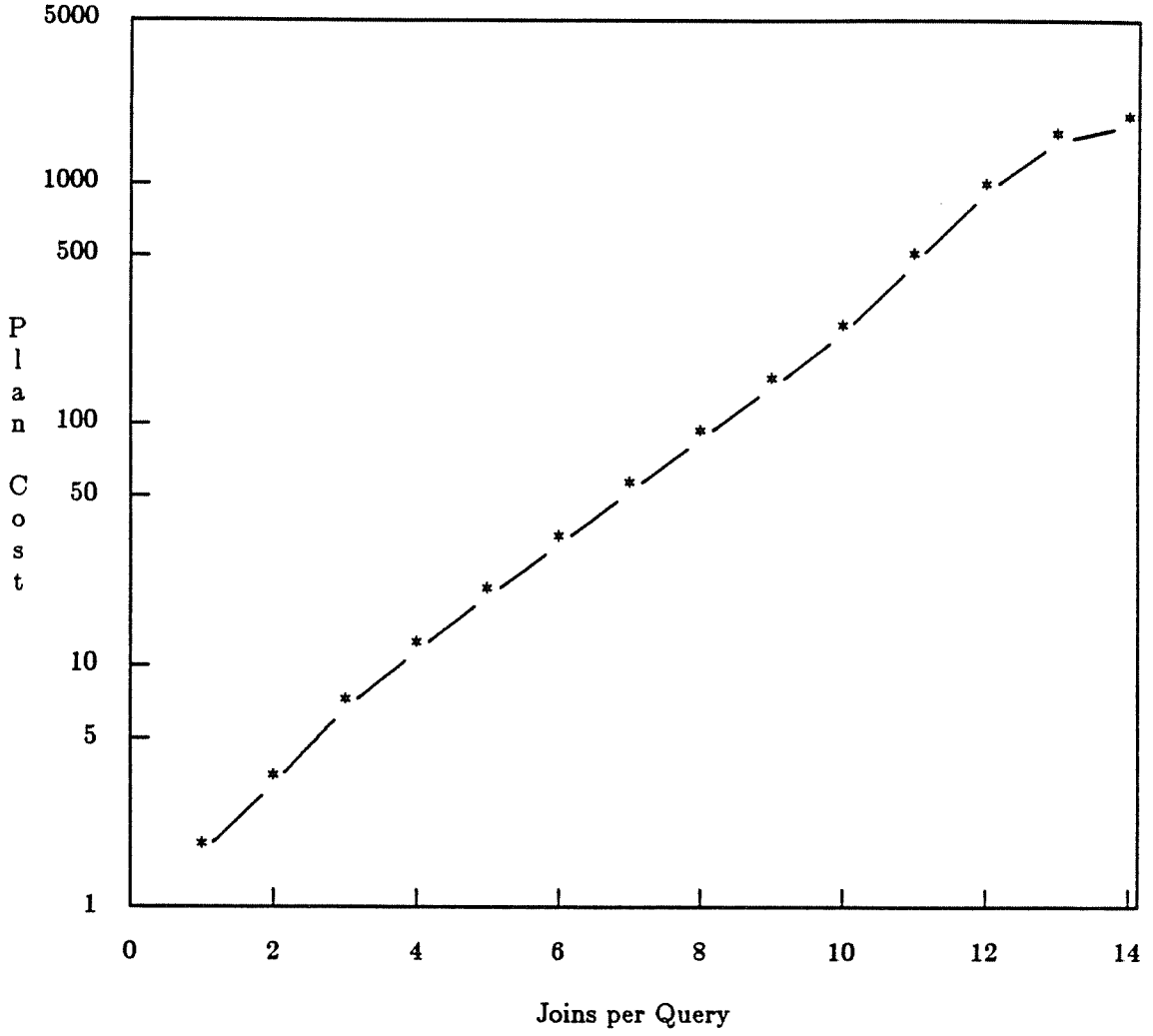


Figure 5.12.

Average Plan Execution Cost.

(in seconds processing time)

Figure 5.9 shows that the average *MESH* size grows with the number of joins in a query, as does the CPU time spent on optimization, shown in Figure 5.10[3]. This experiment was performed with the hill climbing and reanalyzing factors set to 1.5. The graphs illustrate that the generated optimizer can work on fairly complex queries.

Figures 5.11 and 5.12 show that the plan execution time grows exponentially, which can be seen best in Figure 5.12 which has a logarithmic scale. The reason is that, with each additional join, the number of tuples being created in evaluating the query grows. The cardinalities of each of the base relations is about 1,000, and the join selectivity for this experiment was set such that a join result contains 0.15% of the tuples of the Cartesian product.

5.2. Optimizer Generator Usage Outside of EXODUS

5.2.1. Relational Query Evaluation Using Horwitz's Method

A third optimizer is currently being implemented for a new query evaluation method proposed by Horwitz (Horwitz, 1985,Horwitz, 1986). Even though the model that the optimizer generator is based on does not exactly match this new method, the users of the generator started to use their optimizer fairly quickly.

The key idea of the new query evaluation method is to provide three forms for each of the operators in a relational system. They are called **relation-producing**, **selective-retrieval**, and **membership-test**.

A query tree with a relation-producing operator at the root will return the complete intermediate relation, as specified by the query tree. All of the operators considered in the optimizers described previously are relation-producing.

A selective-retrieval operator also returns a set of tuples, but only those tuples that satisfy a certain predicate. However, selective-retrieval operators need information from

---

[3] These experiments were also run on a VAX 8600.

operators further up in the tree, typically constants for the predicate. For instance, a relational equi-join using an index would be a method that uses a selective-retrieval operator for the inner relation. The new aspect of Horwitz's query evaluation method is that it does not require that the inner relation be a base relation indexed on the join attribute. Since all relational operators are available as selective-retrieval operators, this join method can be used at any position in a query tree.

A membership-testing operator does not return tuples; instead, it determines whether a given tuple would be produced or not. For instance, to determine the intersection of two relations, it is convenient to use a relation-producing method for one of the two relations, and for each of the tuples returned to use a membership-test method for the other relation.

Example 5.5: Consider a join node in a query tree. Methods like hash join or sort-merge join require relation-producing methods for both inputs. An index join uses a selective-retrieval method on the inner relation. If the query predicate and the target list are such that no values from the inner relation are needed, i.e. only the existence of a matching tuple is required, a membership-test is sufficient.

□

Initially, it was not immediately clear how these query evaluation methods would fit with the optimization model used by the generator. In particular, the existence of more than one form of each operator was not anticipated in the design of the generator. For the optimizer generator, each form of each relational operator is declared as a separate operator. Unfortunately, this requires a large number of rules, because each rule has to be specified for each combination of operator forms. In order to reduce the number of rules, artificial operators for relation-producing, selective-retrieval, and membership-test were introduced into the query trees. Using these artificial operators, the height of all trees is doubled, with levels of artificial and relational operators alternating.

Ideally, the relational operator beneath an artificial selective-retrieval operator is of the selective-retrieval type. Similarly, it is desirable that the relational operator beneath an artificial membership-test operator is of the membership-test type. However, this is not always possible, due to the fact that the artificial operators for selective-retrieval and membership-test can be above a relation-producing relational operator. This is required for two reasons. First, this is an intermediate step in propagating selective-retrieval and membership-test down in the query tree. Second, in some situations, particularly when membership-test is called very frequently, such a query tree represents the best way to evaluate the query. In such cases, the complete intermediate relation is collected using a relation-producing operator and stored in a temporary file or index; subsequent membership-tests are performed without refering to operators in lower levels of the tree.

Example 5.6: Consider Figure 5.13. This query tree produces all tuples that belong to the intersection of A and B. Consequently, there is a relation-producing artificial operator at the top. The child nodes of the intersection operator node are a relation-producing operator

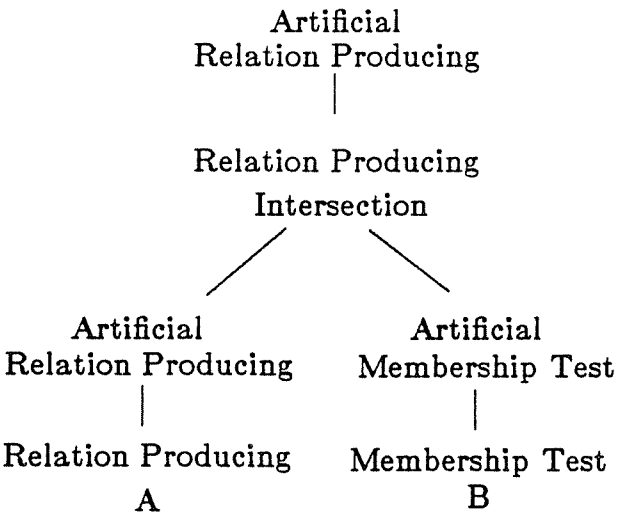


Figure 5.13.
Artificial Operators.

and a membership-test operator. All tuples from A are tested to see if they are also members of B, and qualifying tuples are returned to the top operator of the query.

□

The trees presented to the optimizer do not contain either selective-retrieval or membership-test operators. Rather, these are introduced step by step during the optimization process. For example, the query tree shown in Figure 5.13 is the result of an application of a rule that says that an intersection can be performed using a membership-test on the second relation instead of a relation-producing form.

Some of the problems encountered using the optimizer generator for this model resulted from the fact that, initially, zero costs were assigned to the artificial operators because these operators had no equivalent in the corresponding access plans. However, in some cases the best alternative might be to materialize an intermediate result in a temporary file and then do repetitive selective-retrievals or membership-tests on this file. To find these cases correctly, the cost model was changed and real costs were assigned to the artificial operators. (In some sense, these operators are not artificial anymore.) This change also eliminated some numeric problems that the search mechanism had with zero costs. For instance, special cases arise due to zero costs when updating rule statistics or in benefit estimation.

Other experiences here included the need for higher hill climbing and, in particular, reanalyzing factors to propagate changes in lower levels of the tree into higher levels, allowing for more efficient query evaluation plans.

5.2.2. The Alpha-Extended Relational Algebra

Finally, the optimizer generator is being used to implement a relational database system extended by the "alpha" operator by Agrawal (Agrawal, 1987). The alpha operator is most easily explained using a graph model of the data in a relation. Imagine a relation that

stores a graph by storing the start and end points of each edge in a tuple. In its simplest version, the alpha operator computes the transitive closure, i.e. the reachability graph, for this relation. Furthermore, it can perform aggregate computations when concatenating two edges. If there is a cost stored with each edge, the alpha operator can calculate a cost for all edges in the reachability graph. It also can decided which path to include in the result if there is more than one path between two vertices. This capability allows the the alpha-extended relation algebra to solve minimum cost path problems. Finally, tuples in the transitive closure can be selected depending on their derivation history, e.g. based on which (or how many) intermediate points a path includes.

Agrawal plans on using results from (Jagadish, 1987) and (Agrawal, 1987) to define a rule set for the optimizer generator. The implementation rules will reflect results presented in (Agrawal, 1987). The query tree as provided to the optimizer will contain transitive closure operators without aggregation or arbitration. First, the optimizer will replace a transitive closure operator with an instance of the alpha operator. Next, it will try to move as many operations as possible into the alpha operator. In a second optimization phase, standard relational equivalence rules will be used for operator reordering of projections, selections, joins, etc.

CHAPTER 6

A Comparison of Alternative Optimization Strategies

In this chapter, we compare alternative classes of optimization and query execution strategies for relational database systems. Most relational optimizers prune certain access plans without evaluation, namely those which require either intermediate files or sophisticated scheduling. The common rationale is that the set of pruned plans never contains the optimal access plan, and that the optimal plan or one very close to optimal can be found in the plans that are considered. In the first section of this chapter, we demonstrate that there are cases in which the optimal plan is among those pruned by most relational optimizers.

In the second section, we report on a comparison of a two-phase optimizer and an equivalent one-phase optimizer. In effect, we concatenate the two optimizers compared in the first section to capture the advantages of both. While this works well in general, the desired advantages cannot be realized in all cases.

Using the optimizer generator, it was possible to explore the tradeoffs between alternative optimizers with minimal effort. Using exactly the same the rules, cost functions, and the same search strategy provides significant benefits for the experiments. The search strategies and access plans are then directly comparable, both in terms of optimization effort and query execution time.

6.1. Left-Deep vs. Bushy Execution Trees

When designing the rule sets for our optimizers, we had to consider how the run-time system would evaluate queries. In particular, we had to decide whether the run-time system would freely use temporary files for intermediate results. While temporary files are

considered necessary for successful decomposition for storing intermediate results while the query optimizer decides on the next processing step (Wong, 1976), System R tries to avoid the use of temporary files by structuring the query evaluation plan appropriatly (Astrahan, 1976).

In System R, it is required that in any join operation the inner relation be a base table (i.e. a stored, permanent relation). The rationale is that this allows the use of an existing index to speed up the join. In our optimizer, the inner relation of a join was modeled as the right input. System R style access plans for queries involving multiple joins thus have one long left branch in our model, so we call them **left-deep trees**.

In contrast to System R, the generated optimizer presented in Chapter 5 considers trees of all shapes. We term the set of all sets the set of **bushy trees**. Figure 6.1 shows a left-deep tree and a bushy tree representing the same join query. Both the University version and the commercial version of INGRES allow access plans in the form of bushy trees (Wong, 1976,Kooi, 1982). In fact, in the Decomposition strategy, the sizes of intermediate results are used as basis of subsequent optimization decisions (Wong, 1976).

There are advantages and disadvantages to using either deep or bushy tree access plans. The differences have ramifications both in the optimization and execution phases of

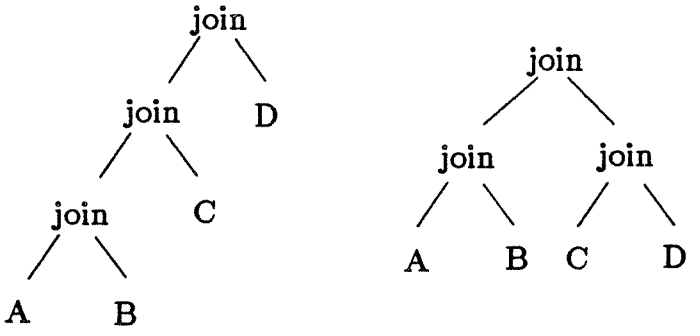


Figure 6.1.
Left-Deep vs. Bushy Trees.

a query. Selecting the optimal access plan (or one close to optimal) from the set of possible access plans becomes more difficult as this set grows. The set of bushy trees is a superset of the set of left-deep trees, and is possibly larger by several orders of magnitude. The number of left-deep access plans grows by the order of 2^N for N join operations in the query, whereas the number of bushy trees grows by 8^N. Hence, there is a strong reason to restrict the optimizer's search to left-deep trees.

During query execution, left-deep trees are also easier to manage. Consider a bushy tree as shown in Figure 6.1. The two lower join operations can be performed either sequentially or concurrently. In the former case, it is necessary to store the results of the first subtree executed until the second tree has been evaluated. This can be done either in main memory buffers, which increases buffer contention dramatically, or in a temporary file, which requires additional I/O for writing and reading. In the latter case, it is necessary to schedule the execution of the two trees so that their results are available exactly when they are needed for the join operator at the root of the tree. Clearly, this is very difficult to the point of being practically infeasible. For this reason, temporary files seem the most general of the three alternative implementation techniques outlined for bushy trees.

In order to compare these two query optimization and execution strategies, we restricted the optimizer implemented with the optimizer generator to producing left-deep trees only. This could be done quite easily with a simple change of the rule set, i.e. by disabling three of the join associativity rules shown in Figure 5.4 of Chapter 5, leaving only the left-shift rule. To incorporate the I/O cost for temporary files, we altered all cost functions for join methods to determine whether the right input is the result of a scan or an intermediate result. If it is an intermediate result, the cost for writing and sequential reading is included in the calculated cost of the query tree.

6.1.1. Experiments and Results

As mentioned earlier, bushy trees are a superset of left-deep trees. Consequently, the search space of the bushy tree optimizer is larger. This does not necessarily imply, however, that the bushy tree optimizer must be slower. This depends on the effectiveness of the search strategy and on the amount of sharing of nodes in *MESH* among query trees and access plans. On the other hand, we certainly expect that the access plans found by the bushy tree optimizer will be at least as good as those found by the left-deep tree optimizer.

The cost model in these experiments includes both I/O cost and CPU cost. The cost measure used is the total processing time, i.e. elapsed seconds for query evaluation without overlap of I/O and CPU activity. The buffer pool is assumed to contain 100 pages of 4,096 bytes. The constants and formulas used in the cost calculations are given in Table 6.1 and 6.2. The #pages parameter stands for the number of pages that an intermediate result occupies, #tuples stands for the number of tuples in an intermediate relation, and #buffers stands for the number of pages in the buffer pool. We assign relatively low costs to sequential read and write operations as we assume that file space allocation is done in a way so as to minimize disk seeks. For hash join, the spooling cost for the outer relation (in first line of the cost formula for hash join) is used only if #runs is greater than 1, i.e. if the outer relation does not fit into the hash table which is kept in the buffer.

activity	abbreviation	unit	ms
sequential read	S_RD	page	15
random read	R_RD	page	30
write	WR	page	20
memory to memory copy	COPY	page	2
key comparison	COMP	tuple	0.05
build hash table	BUILD	tuple	0.2
probe hash table	PROBE	tuple	0.5

Table 6.1.
Constants used in the Cost Calculations.

Method	Formula
file scan	S_RD * #pages
index scan	R_RD * #tuples
filter	COMP * #tuples
sort-merge join	2 * (#tuples$_{left}$ + #tuples$_{right}$) * COMP + #pages$_{result}$ * COPY
sorting	#pages * $\log_{\#buffers}$ (#pages) * (WR + S_RD) + #pages * $\log_{\#buffers}$ (#pages) * COPY + 2 * #tuples * $\log_{e}$ (#tuples) * COMP
nested loops join	#pages$_{right}$ * (WR + #runs * S_RD) + #tuples$_{left}$ * #tuples$_{right}$ * COMP + #pages$_{result}$ * COPY #runs = ceiling (#pages$_{left}$ / #buffers)
hash join	#pages$_{left}$ * (WR + S_Rd) + #pages$_{right}$ * (WR + #runs * S_RD) + #tuples$_{left}$ * BUILD + #tuples$_{right}$ * PROBE + #pages$_{result}$ * COPY #runs = ceiling (#pages$_{left}$ / #buffers)
index join	2 * #tuples$_{left}$ * R_RD + 10 * #tuples$_{left}$ * COMP + #pages$_{result}$ * COPY
spooling	#pages * (WR + S_RD)

Table 6.2.
Formulas used in the Cost Calculations.

The database used for these experiments is that described in Section 5.1.2 of Chapter 5. It contains 26 relations with 4 attributes each. The relation cardinalities are randomly chosen from a normal distribution with a mean of 1,000 and a standard deviation 100.

We were interested in seeing how bushy and left-deep tree optimization compare for various query sizes. To explore this issue, we used 200 queries with one join each, 200 with two joins each, etc., up to 9 joins per query. The queries were presented to the optimizer as

left-deep trees. We explored various selectivities for the join predicate. In the following, we give the results for one particular join selectivity setting. The result relation of a join contains 0.15% of the tuples of the Cartesian product of the two relations. Considering the cardinalities of the relations in our database, this means that the result size of a query grows which each additional join.

We provide four graphs comparing left-deep trees and bushy trees. In each graph, we give the number of joins per query on the horizontal axis, and show one curve labelled "*" for left-deep execution trees and one curve labelled "+" for bushy execution trees. First, we compare the average number of nodes in *MESH* at the end of the optimization of one query. This is one way in which we can measure the effort that was spent on optimization. Second, we compare the average CPU time spent optimizing one query[1]. Third, we compare the average plan execution costs as anticipated by the cost functions.

[1] These experiments were also run on a VAX 8600.

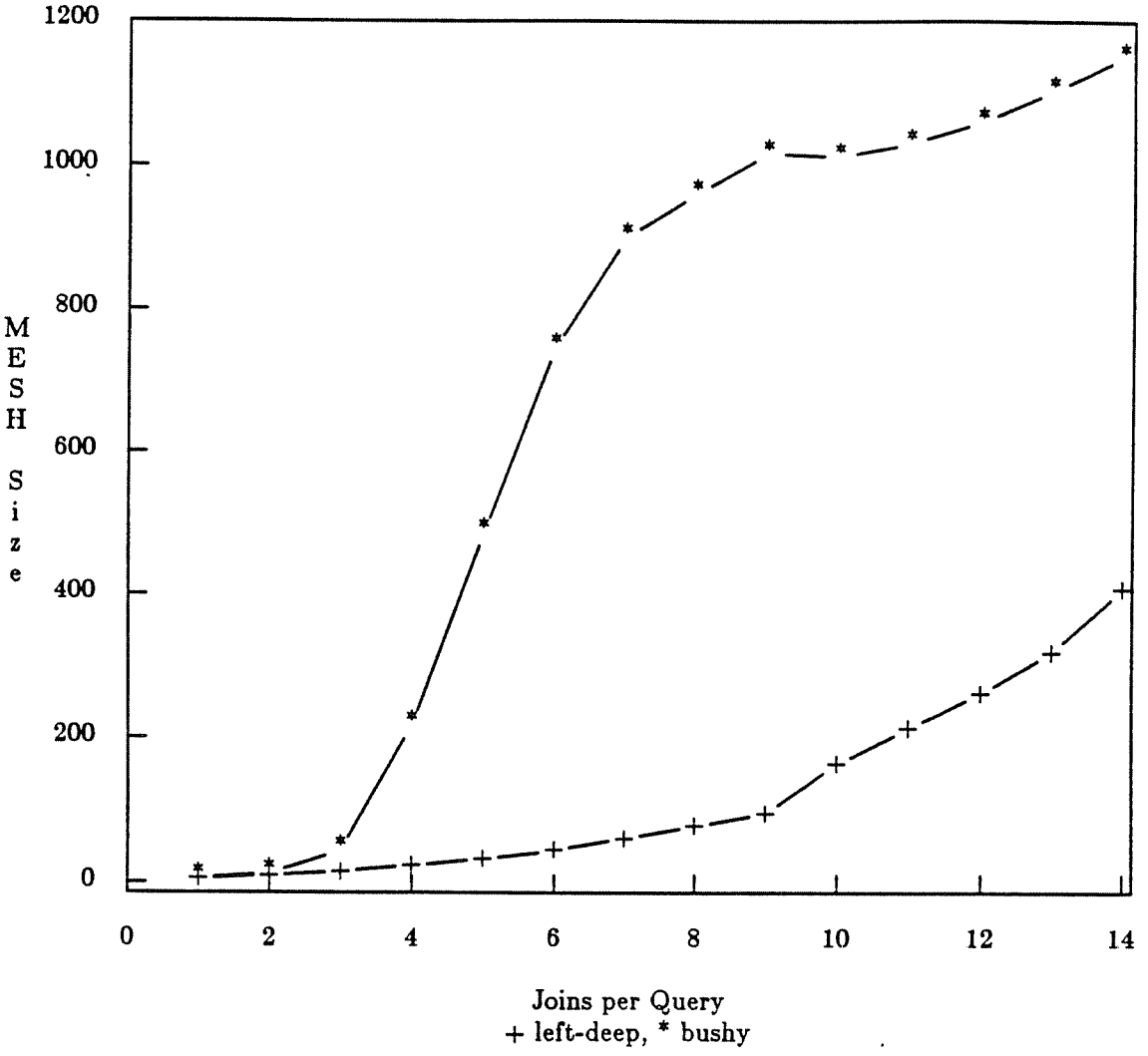


Figure 6.2.

Average of *MESH* Size.

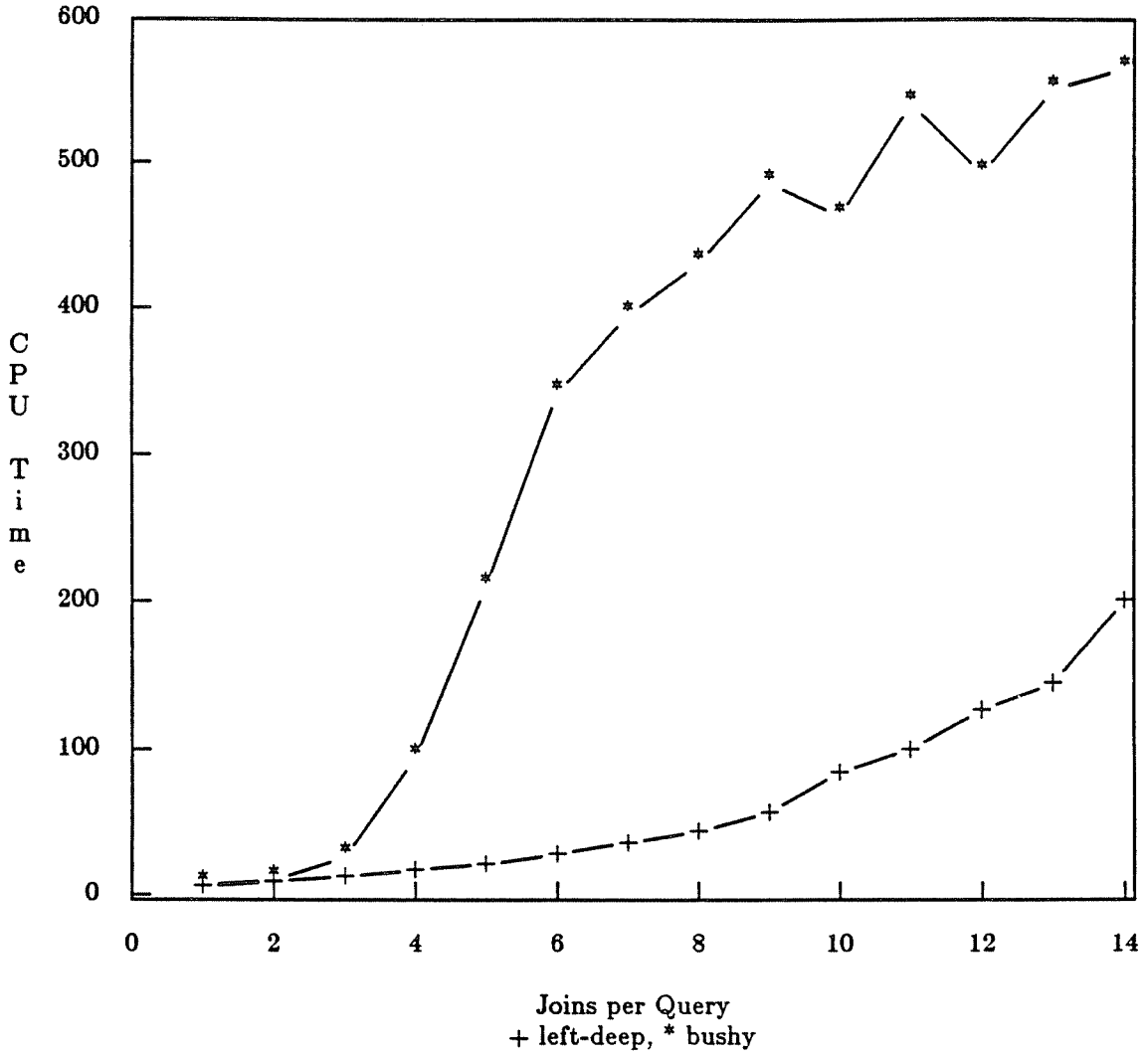


Figure 6.3.

Average of CPU Time.

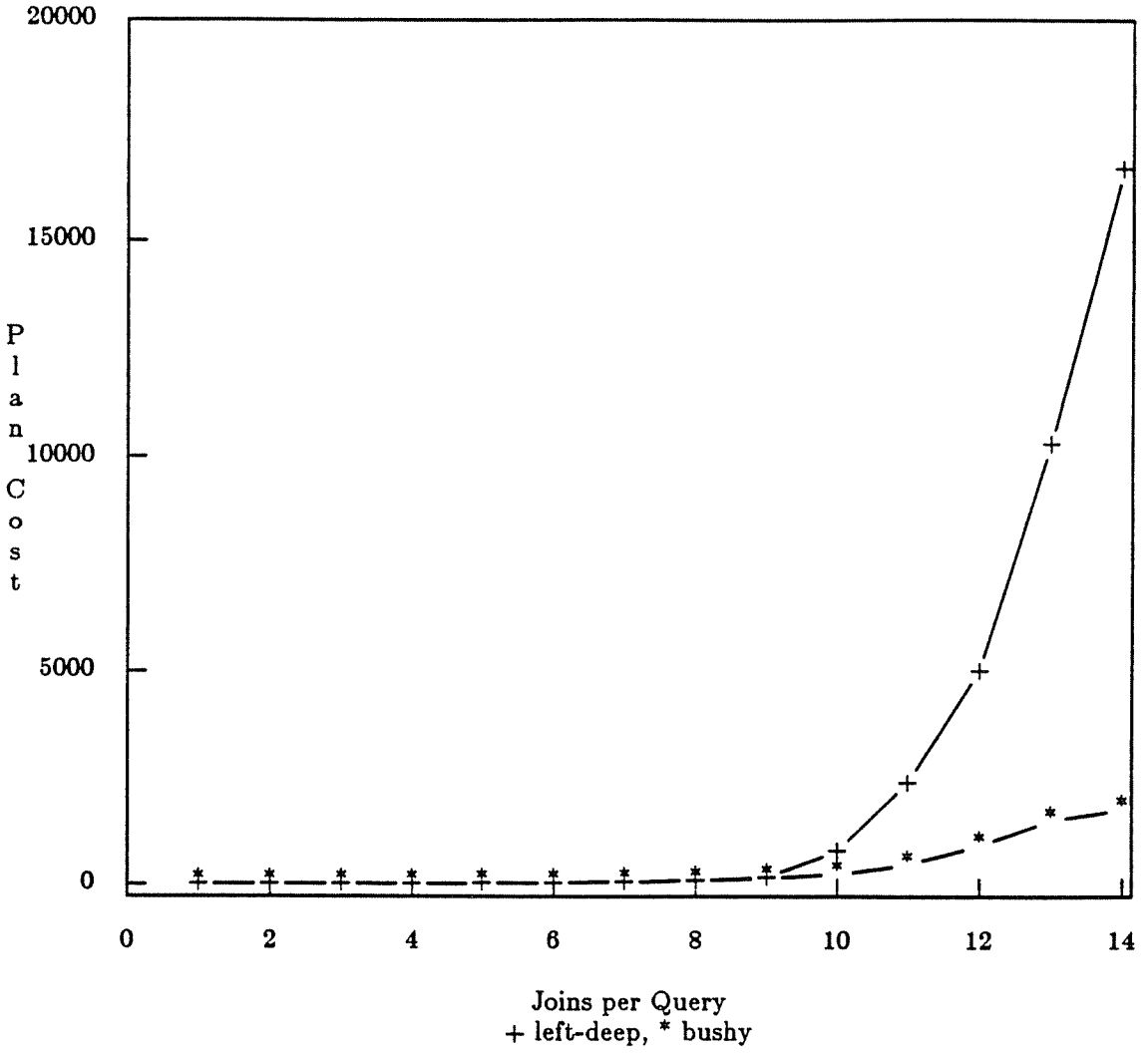


Figure 6.4.

Average of Plan Execution Cost.

(in seconds processing time)

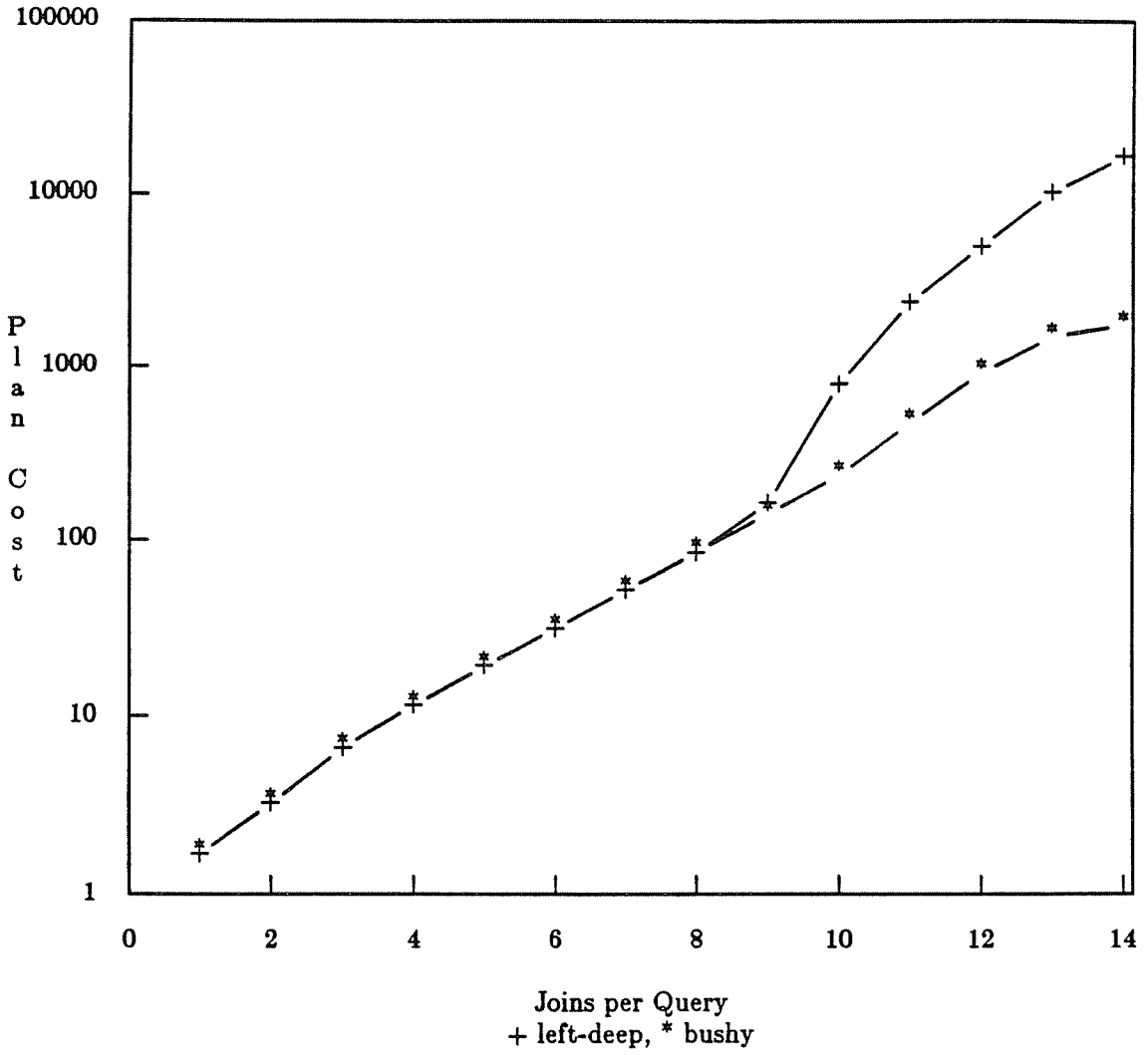


Figure 6.5.

Average of Plan Execution Cost.

(in seconds processing time)

The search effort, as measured by the final *MESH* size (Figure 6.2) and by the CPU time spent on optimization (Figure 6.3), grows with the number of joins in the query, i.e. the complexity of the query. Since there are many more bushy trees than left-deep trees, the optimization of bushy trees is more complex, and thus requires more resources.

The plan execution cost seems to be equal for queries of limited complexity, as can be seen in Figures 6.4 and 6.5. The design decision of the System R and GAMMA projects to use left-deep trees thus only seems to be justified. The plan execution cost grows exponentially with the number of joins in the query, i.e. with the number of tuples in all intermediate relations. However, for very complex queries, left-deep execution trees are much more expensive.

In fact, the results shown here were to be expected. The join selectivity factor and the cardinalities of the base relations cause the size of the intermediate relations to grow steadily with each additional relation joined. Hence, the number of tuples in each intermediate relation grow exponentially with the distance from the left-most leaf when using a left-deep query execution tree. A bushy tree, on the other hand, allows the optimizer to plan more join operations with moderate cardinalities, putting only one really large join operation at the top of the tree. The more join operations there are in the query, the more impact such "balancing" has, and the greater the execution cost advantage of bushy trees. If a join method requires multiple runs (because the relations are larger than the available buffer) or sorting of an intermediate result, the cost functions assume that the intermediate result must be stored on disk anyway for this purpose, and the spooling introduced by the bushy shape of the tree does not introduce additional cost in this case.

In many practical applications, the size of join results can be estimated more accurately with a different formula than the one used in the experiements reported above. To ensure that a relational database is in normal form (Kent, 1983), it is frequently required

that complex objects be stored in tuples from several relations (Lorie, 1985). If relational joins are used to rebuild complex objects, a more appropriate formula to estimate join result sizes is the product of the smaller of the two input cardinalities and a constant factor. The smaller of the two cardinalities is assumed to represent the number of objects to be rebuild. The constant factor is an estimate of how many components are to be retrieved in a join operation for each complex object. Experiments indicate the left-deep execution trees are satisfactory if all complex objects can be stored in main memory. If the main memory is not sufficiently large, it is more advantageous to assemble non-normalized components in a right branch of a bushy execution tree.

These experiments cannot be immediately generalized to other cost functions, databases, and queries. However, the assumptions made in our experiments are realistic enough that they could occur in a real database. If so, the conceptual ease of the execution model of System R and GAMMA, namely the restriction to left-deep execution plans, must be paid for in form of higher query execution costs for some queries and some databases. An assessment of this tradeoff should be part of any database implementation effort.

6.2. Two-Phase Optimization

Left-deep tree optimization is faster, whereas bushy tree optimization generates the better query plans. In this section, we explore whether the advantages of the two methods can be combined.

The more general formulation of this idea is to perform the optimization in an arbitrary number of phases, as we proposed in (Graefe, 1986). In Chapter 4, we reported how the necessary framework was implemented in the EXODUS optimizer generator. Using this framework, it was very easy to modify the model description file used for the experiments reported above such that there are two phases. The first phase does left-deep tree optimization, and then the second phase takes the query tree corresponding to the best access plan

found by the left-deep tree optimizer as input into the bushy tree optimizer and tries to find a better access plan. Again, the implementation rules, the cost functions, and the property functions did not need to be changed, making the results directly comparable.

A similar idea was outlined by Rosenthal et al. (Rosenthal, 1986), who called it the Pilot Pass Approach. The pilot pass is a search pass with a more restricted search that is done before the main optimization pass. The purpose is to allow the main optimization pass to find the optimal access plan faster. The hope is that more optimization effort is saved in the main optimization than is spent on the pilot pass. No implementation was reported by Rosenthal et al.

We compared the two-phase optimizer with the left-deep and bushy tree optimizers described in the last section. We used the same queries and database to measure the optimization effort and anticipated execution cost of the access plans. In the following graphs, the curve labelled "x" represents two-phase optimization.

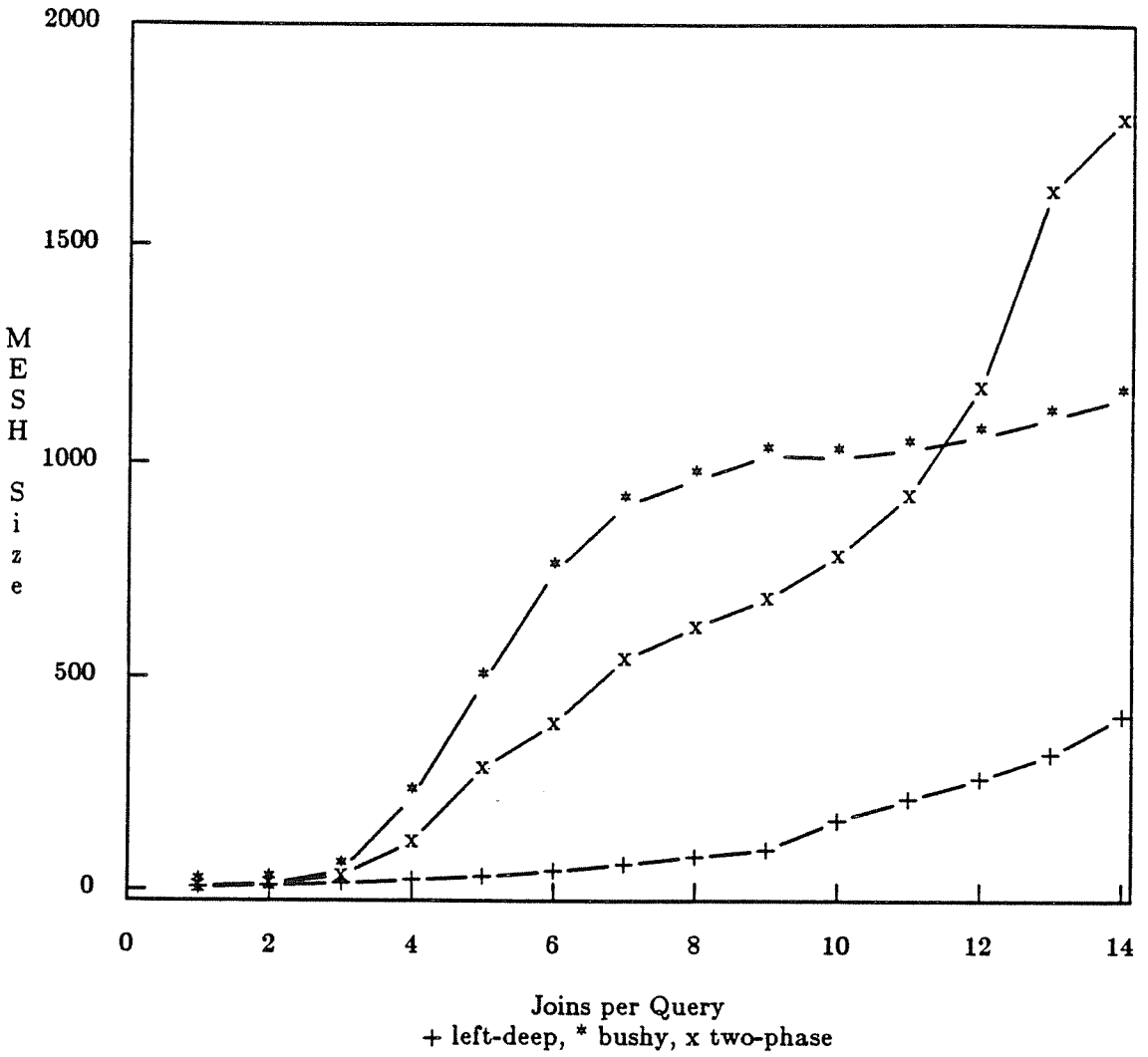


Figure 6.6.

Average of *MESH* Size.

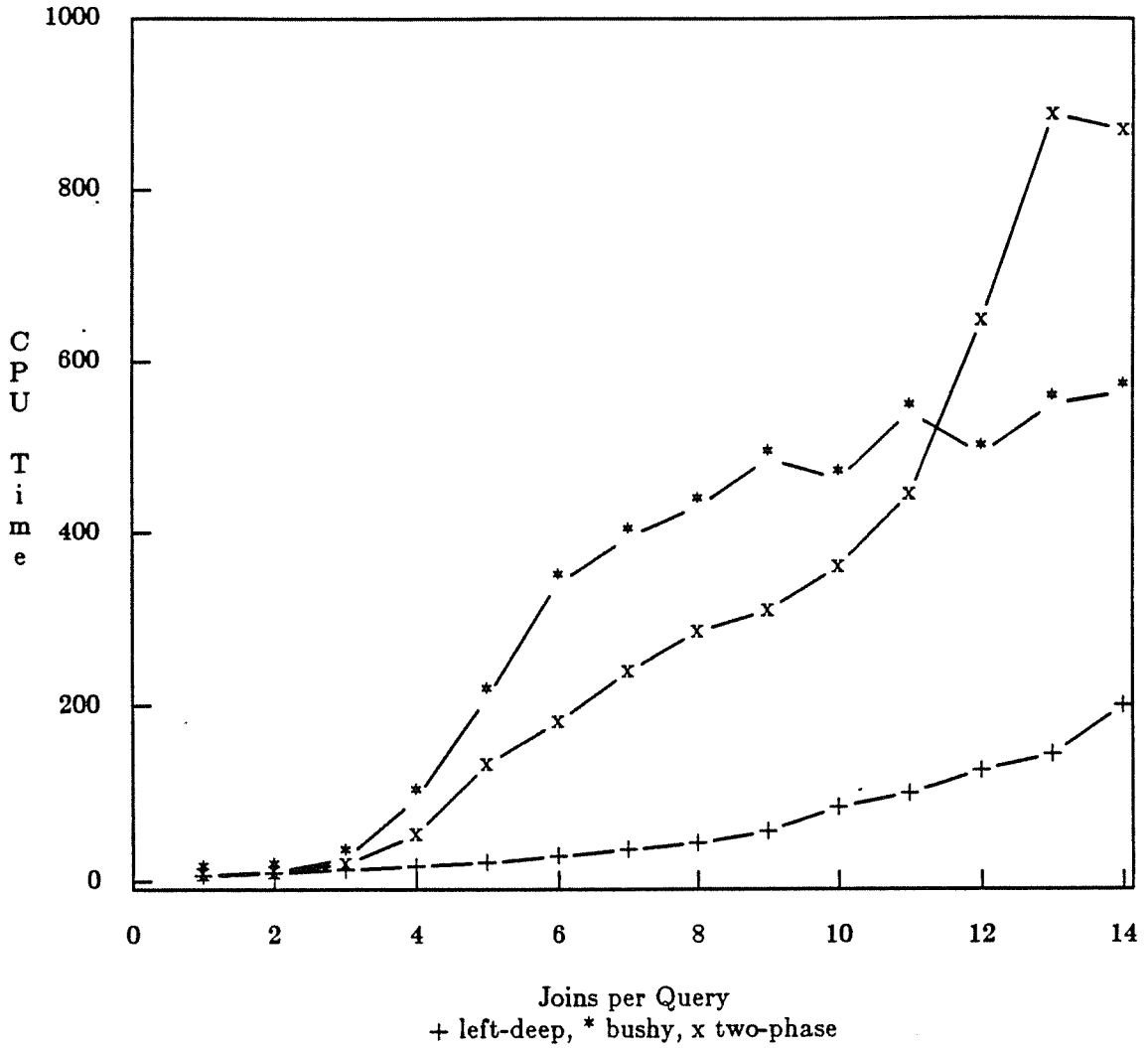


Figure 6.7.

Average of CPU Time.

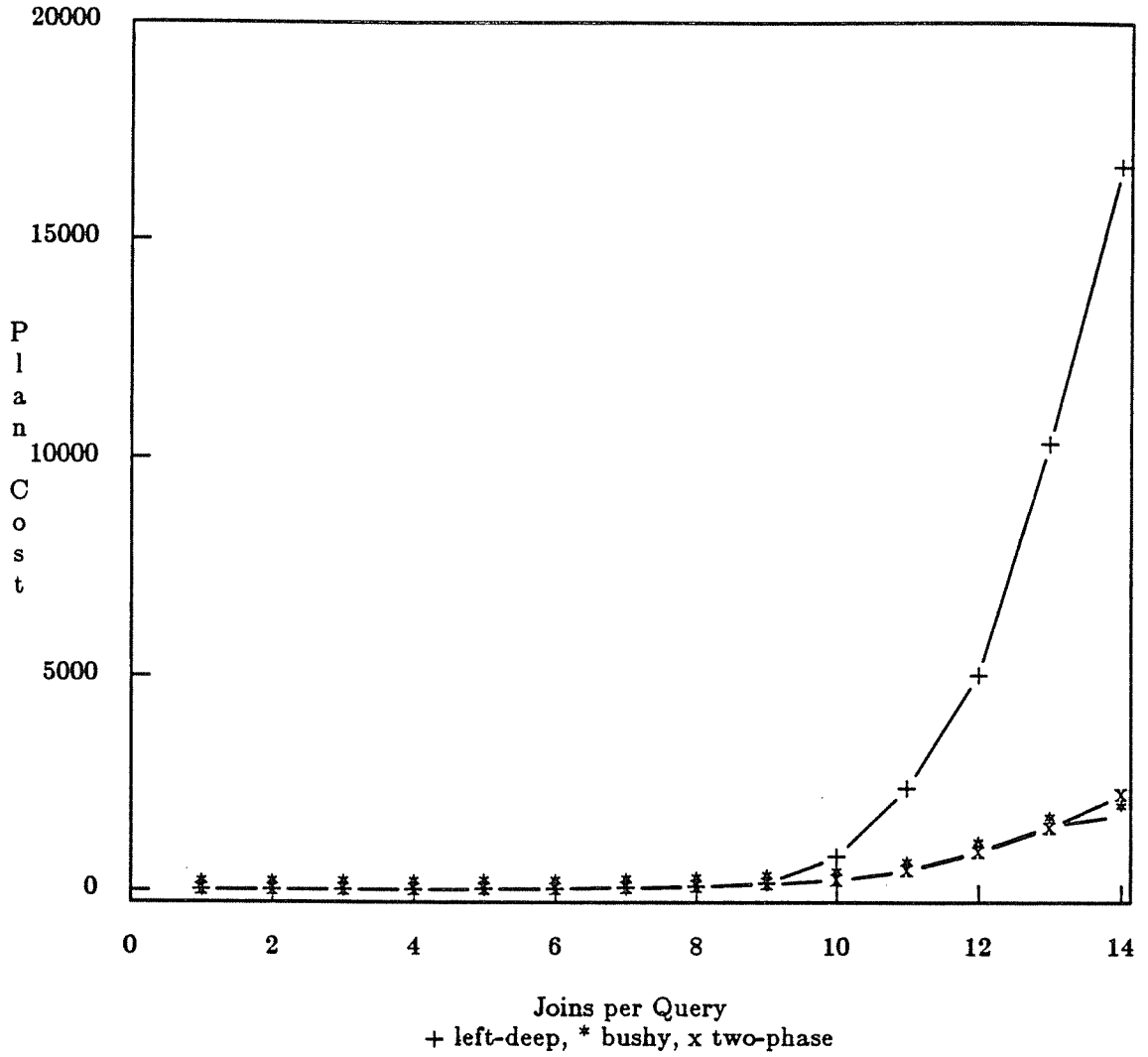


Figure 6.8.

Average of Plan Execution Cost.

(in seconds processing time)

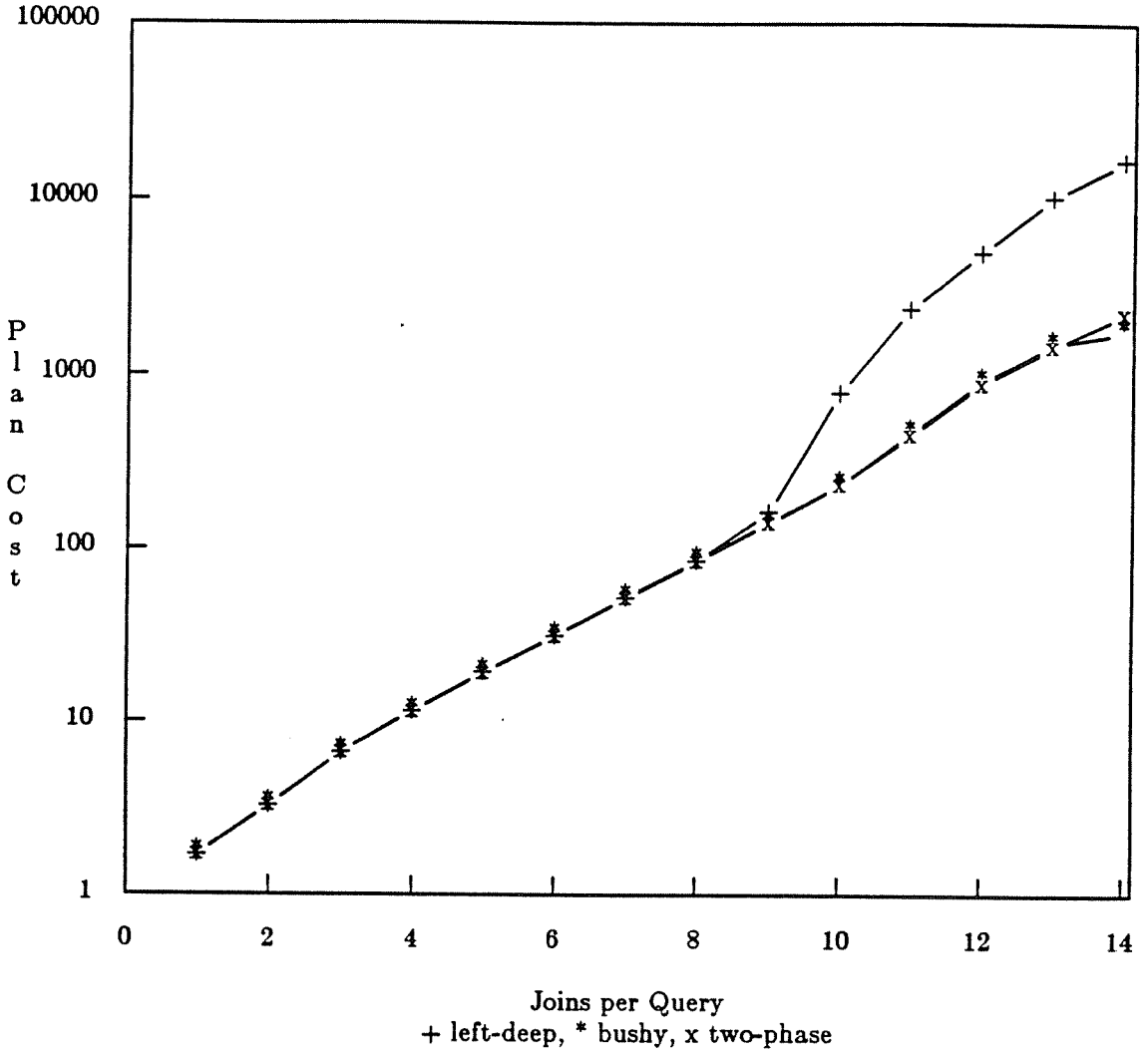


Figure 6.9.

Average of Plan Execution Cost.

(in seconds processing time)

The measures for optimization effort, *MESH* size (Figure 6.6) and CPU time (Figure 6.7), are again very similar to each other. For queries of moderate complexity (4 to 10 joins), we observe that two-phase optimization requires less resources than one-phase bushy tree optimization, i.e. the left-deep first phase saves more effort in the bushy second phase than it costs. For very large queries, however, this is not the case.

With respect to the anticipated plan execution cost, shown in Figures 6.8 and 6.9, we find that the plans resulting from two-phase optimization are just as good as those produced by bushy tree optimization. Since the second phase is exactly the bushy tree optimizer, this result was to be expected. For very large queries, the left-deep tree optimizer cannot find plans as good as the bushy tree optimizer or the two-phase optimizer. Interestingly, the range for which the two-phase optimizer is slower than the bushy tree optimizer (more than 12 joins) overlaps with the range for which left-deep trees are not competitive. Further research might show whether or not there is a connection between these two ranges.

The optimization model of EXODUS provides for any number of phases that the DBI wishes. Furthermore, what the phases do or what their search parameters (hill climbing factor, reanalyzing factor, etc.) are is not predetermined. This allows the DBI maximal freedom to explore alternative ideas in query optimization and execution. The generator architecture and its modularization framework allow for rapid implementation of these ideas. Our idea to concatenate the fast left-deep tree optimization and more thorough bushy tree optimization is but one idea for a multi-phase optimizer; other combinations might prove to be more effective. We have not yet explored this direction further, however, because the focus of our research has been to provide a flexible and powerful tool for database query optimization research.

CHAPTER 7

Summary, Future Work, and Conclusions

7.1. Summary

In this thesis, we have outlined problems and possible solutions for query optimization in extensible database systems. Our research suggests that a particular solution, namely a rule-based query optimizer generator, is a practical and powerful alternative. A prototype implementation for the EXODUS extensible database project was described, and some computational results were reported. Our results demonstrate that optimizers generated with our software can be valuable parts of new database systems.

To support query optimization in extensible database systems, the data model specific software components must be separated from the components that can be used for any data model. The reusable modules are part of the EXODUS effort, whereas the data model specific software is defined by the Database Implementor (DBI). An optimization model general enough to fit most modern data models allows the DBI to specify the data model for the optimization component. Our optimization model is based on the algebra of the data model, i.e. on operator trees and method trees. Optimization in this model consists of operator reordering and method selection.

The data model, for optimization purposes, is captured in a new rule language and in a set of support functions written in the DBI's implementation language. The rules are transformation rules for operator reordering and implementation rules for method selection. In general, the rules are non-procedural, but the rule language allows the DBI to express hints on how the rules are to be applied. The support functions can be associated with operators, methods, or rules. They include the cost functions for methods, property

functions for operators and methods, and argument transfer and promise evaluation functions for rules. Most of these support functions are optional, thus being a convenience rather than a burden for the DBI.

Using the rule set as a base for the optimizer provides a clear and concise framework for modularization of the DBI's optimizer code. The benefits include easy incremental development and testing of database query optimizers.

The rules are translated by the optimizer generator into executable source code, which is compiled and linked with the support functions and other database software. In the translation process, self-adapting search procedures are appended to the generated code. While the optimizer runs, the search engine observes the effects of transformations to guide further search; an ***expected cost factor*** is associated with each rule for this purpose. This self-adaptive search strategy avoids exhaustive search, drastically reducing the time required for query optimization.

The EXODUS prototype of the optimizer generator demonstrates the feasibility of the approach. Optimizers built for relational systems were shown to be fast enough to be used in production systems. The costs of the access plans produced are very close to those found by an exhaustive search of all possible query trees and access plans. Optimizers designed for other research efforts show the flexibility and the power of the approach.

Finally, we have shown results obtained with the relational optimizers generated with the EXODUS optimizer generator. While the comparison of left-deep vs. bushy query execution trees and single- vs. multi-phase query optimization were not immediate concerns of the EXODUS project, they are nevertheless issues that must be considered when implementing a new database management system. The flexibility provided by the EXODUS optimizer generator enabled us to implement easily several different optimizers that were directly comparable. The optimizer generator has thus proven to be a valuable tool for research and

experimentation in database query optimization.

7.2. Future Work

One interesting design issue that remains is to provide general support for the notion of predicates, as some form of predicates are likely to appear in most data models. Writing the DBI code for predicates was the hardest part of developing our optimizer prototypes. In the current design, the DBI must design his or her own data structures and provide all the operations on them for both rule conditions and argument transfer functions. It may be difficult to design a generally satisfying definition of predicates and a scheme for supporting predicates, but it would be a significant improvement for the optimizer generator. The fact that predicates are a special case of arguments poses an additional challenge, since the overall design of the argument data structure must still remain with the DBI. One way to approach this problem is to provide a library of subroutines which the DBI can choose to use if they fit the data model.

The hill climbing and reanalyzing factors were seen to have a significant effect on the amount of CPU time spent optimizing a query. These values are almost surely model and algebra dependent. Thus, they must either be set by the DBI or they must be determined automatically. We feel that the former alternative requires a level of sophistication or time for experimentation that cannot be expected from the DBI, hence we plan on implementing a scheme for the latter approach.

Furthermore, we realize that the stopping criteria for the generated query optimizers are not yet completely satisfactory. The inherent problem is that the optimizer cannot determine immediately when it has found the final access plan, and all search strategies thus spend some effort that does not influence the optimization result. More research is needed to identify reliable and precise stopping criteria.

We also plan on making several changes in the generated optimizers. The first is to recognize common subexpressions when the final access plan is extracted from *MESH*. Common subexpressions are detected in *MESH* and optimized only once, but the procedure which extracts the access plan from *MESH* does not exploit this feature. Furthermore, the cost of common subexpressions is not spread over their various occurences. Once common subexpressions are supported satisfactorily, optimization of multiple queries in a single optimizer run will be easy to implement. The other future change is to allow the definition of method classes, as discussed in Section 4.3. This would be useful when adding a new access method to an existing DBMS. In the current design, an implementation rule has to be added to the model description file once for each rule where the new access method can be used. Instead, by using a method class, the new access method would have to be added only once, to the definition of the class.

Finally, we realize that the optimizer generator works largely on the syntactic level of the query algebra. The semantics of the data model are thus left to the DBI's code. This has the advantage of allowing the DBI maximal freedom regarding the type of data model implemented, but it has the disadvantage of leaving a significant amount of coding work to the DBI. We would therefore like to incorporate some semantic knowledge of the data model into the description file. This, however, is a long term goal which we have not begun to address.

7.3. Conclusions

This thesis has described a first attempt to identify and solve the problems encountered in extensible query optimization. We believe that we have contributed to database systems technology both conceptually and practically. The optimization model outlined and used in this work, based on trees of operators or methods, can be used as a unifying framework for database query optimization. It is useful for describing the theoretical underpinnings of a

new data model, which can be captured in the transformation rules; for describing the set of possible execution strategies, captured in alternative sets of implementation rules; and for the comparison of optimizer search spaces, which can be defined by the set of trees considered during optimization. On the practical side, we have designed and implemented a tool that allows database researchers to experiment with query optimization. For the first time, alternative optimizers can be implemented with a reasonable amount of programming effort. We hope that this tool will assist EXODUS users and database researchers to build semantically richer and more user-friendly database systems.

APPENDIX A

An Example Model Description File

This section shows a model description file. Some of the cost functions, operator property functions, and method property functions are missing; this appendix is intended as a source of examples only. The types *ARG_TYPE*, *SCHEMA*, and *ORDER* are assumed to have been defined in the file ***defs.h***. To learn further details how to use the optimizer generator, please refer to the user's manual.

```
%{
# include "defs.h"

# define ARGUMENT ARG_TYPE
# define OPER_PROPERTY SCHEMA
# define METH_PROPERTY ORDER

extern ATTR_DESC * find_attr ();
%}

%operator 0 get
%operator 1 select
%operator 2 join

%method 0 file_scan index_scan
%method 1 filter
%method 2 merge_join
%method 2 loops_join
%method 2 hash_join
%method 1 index_join
%%

# select commutativity rule
# ==========================
select 1 (select 2 (1)) ->! select 2 (select 1 (1))
{{
   /* don't do it if the selections are on the same relations */
   if (strcmp (OPERATOR_1.oper_argument.select_arg.attr->rel,
         OPERATOR_2.oper_argument.select_arg.attr->rel) == 0)
      REJECT;
}}
```

```
# select join rule
# ================
select 1 (join (1, 2)) <-> join (select (1), 2)
{{
# ifdef FORWARD
   if (find_attr (OPERATOR_1.oper_argument.select_arg.attr,
         & INPUT_1.oper_property) == NIL)
      REJECT;
# endif
}}

# join commutativity rule
# =======================
join (1, 2) ->! join (2, 1) join_comm;

# left shift rule
# ===============
join 1 (join 2 (1, 2), 3) ->! join 2 (join 1 (1, 3), 2)
{{
   if (find_attr (OPERATOR_1.oper_argument.join_arg.left,
         & INPUT_1.oper_property) == NIL)
      REJECT;
}}

# right shift rule
# ================
join 1 (1, join 2 (2, 3)) ->! join 2 (2, join 1 (1, 3))
{{
   if (find_attr (OPERATOR_1.oper_argument.join_arg.right,
         & INPUT_2.oper_property) == NIL)
      REJECT;
}}

# join associativity rule
# =======================
join 1 (join 2 (1, 2), 3) <-> join 2 (1, join 1 (2, 3))
{{
# ifdef FORWARD
   if (find_attr (OPERATOR_1.oper_argument.join_arg.left,
         & INPUT_2.oper_property) == NIL)
      REJECT;
# endif
# ifdef BACKWARD
   if ((find_attr (OPERATOR_2.oper_argument.join_arg.right,
         & INPUT_2.oper_property) == NIL)
      REJECT;
# endif
}}

# implementation rules
# ====================
```

```
double cost_filter (node, meth_input)
   REG NODE            *node, *meth_input [];
{
   return (W_comp * meth_input [0]->oper_property.rel_desc.card);
} /* cost_filter */

void properties_select (node)
   REG NODE            * node;
{
   REG SCHEMA          * result, * input;
   REG int          index;
   double           selectivity;

   result = & node->oper_property;
   input = & node->oper_input [0]->oper_property;
   *result = *input;

   strcpy (result->rel_desc.rel, "tmp");

   switch (node->oper_argument.select_arg.comp)
   {
   case EQ: selectivity = dbh_select_sel; break;
   case NE: selectivity = one - dbh_select_sel; break;
   case LE: case GE: selectivity = (one + dbh_select_sel) * half; break;
   case LT: case GT: selectivity = (one - dbh_select_sel) * half; break;
   }

   result->rel_desc.card = input->rel_desc.card * selectivity;
} /* properties_select */

void properties_filter (node)
   REG NODE            * node;
{
   node->meth_property = node->oper_input [0]->meth_property;
}
```

References

Agrawal, 1987.
R. Agrawal, "Alpha: An Extension of Relational Algebra To Express a Class of Recursive Queries," *Proceedings of the IEEE International Conference on Data Engineering*, (February 1987).

Agrawal, 1987.
R. Agrawal and P. Devanbu, *Moving Selections into Linear Least Fixpoint Queries*, AT&T Bell Laboratories, Murray Hill, N.J. (1987).

Agrawal, 1987.
R. Agrawal and H.V. Jagadish, *Direct Algorithms for Computing the Transitive Closure of Database Relations*, ATT Bell Labs, Murray Hill, NJ. (1987).

Astrahan, 1976.
M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson, "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems* 1(2) pp. 97-137 (June 1976).

Barr, 1981.
A. Barr and E.A. Feigenbaum, *The Handbook of Artificial Intelligence*, William Kaufman, Inc., Los Altos, CA. (1981).

Bartels, 1986.
R. Bartels, "Implementierung einer regelbasierenden Planungskomponente fuer die Optimierung von Datenbankanfragen in einer SEQUEL-artigen Sprache," *Diplom Thesis*, Technische Hochschule, Fachbereich Informatik, (January 1986).

Becker, 1984.
R.E. Becker and J.M. Chambers, *S - An Interactive Environment for Data Analysis and Graphics*, Wadsworth, Belmont, CA. (1984).

Blasgen, 1977.
M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases," *IBM Systems Journal* 16(4)(1977).

Bobrow, 1983.
D.G. Bobrow and M. Stefik, "The LOOPS Manual," in *LOOPS Release Notes*, XEROX, Palo Alto, CA. (1983).

Carey, 1985.
M.J. Carey and D.J. DeWitt, "Extensible Database Systems," *Proceedings of the Islamorada Workshop*, (February 1985).

Carey, 1986.
M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E. Richardson, E.J. Shekita, and M. Muralikrishna, "The Architecture of the EXODUS Extensible DBMS: A Preliminary Report," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems*, pp. 52-65 (September 1986).

Carey, 1986.
M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita, "Object and File Management in the EXODUS Extensible Database System," *Proceedings of the Conference on Very Large Data Bases*, pp. 91-100 (August 1986).

Chamberlin, 1974.
D.D. Chamberlin and R.F. Boyce, "SEQUEL: A Structured English Query Language," *Proceedings of ACM SIGMOD Workshop*, pp. 249-264 (May 1974).

Chamberlin, 1975.
D.D. Chamberlin, J.N. Gray, and I.L. Traiger, "Views, Authorization and Locking in a Relational Data Base System," *Proceedings NCC*, pp. 425-430 (1975).

Chamberlin, 1981.
D.D. Chamberlin, M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnik, P. Griffiths Selinger, D.R. Slutz, B.W. Wade, and R.A. Yost, "Support for Repetitive Transactions and Ad Hoc Queries in System R," *ACM Transactions on Database Systems* **6**(1) pp. 70-94 (March 1981).

Chamberlin, 1981.
D.D. Chamberlin, M.M. Astrahan, M.W. Blasgen, J.N. Gray, W.F. King, B.G. Lindsay, R. Lorie, J.W. Mehl, T.G. Price, F. Putzolo, P. Griffiths Selinger, M. Schkolnik, D.R. Slutz, I.L. Traiger, B.W. Wade, and R.A. Yost, "A History and Evaluation of System R," *Communications of the ACM* **24**(10) pp. 632-646 (October 1981).

Chou, 1985.
H.T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug, "Design and Implementation of the Wisconsin Storage System," *Software - Practice and Experience* **15**(10) pp. 943-962 (October 1985).

Clocksin, 1981.
W. Clocksin and C. Mellish, *Programming in Prolog*, Springer, New York (1981).

Codd, 1970.
E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* **13**(6) pp. 377-387 (June 1970).

Codd, 1972.
E.F. Codd, "Relational Completeness of Database Sublanguages," pp. 65-98 in *Data Base Systems*, ed. R. Rustin,Prentice-Hall, New York (1972).

Cruz, 1987.
K.F. Cruz, A.O. Mendelzon, and P.T. Wood, "A Graphical Query Language Supporting Recursion," *Proceedings of the ACM SIGMOD Conference*, pp. 323-330 (May 1987).

Dayal, 1985.
U. Dayal and J.M. Smith, "PROBE: A Knowledge-Oriented Database Management System," *Proceedings of the Islamorada Workshop*, (February 1985).

DeGroot, 1975.
M.H. DeGroot, *Probability and Statistics*, Addison-Wesley, Reading, MA. (1975).

DeWitt, 1984.
D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD Conference*, pp. 1-8 (June 1984).

DeWitt, 1986.
D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of*

the Conference on Very Large Data Bases, pp. 228-237 (August 1986).

Forgy, 1981.
C.L. Forgy, "OPS5 Reference Manual," *Computer Science Technical Report 195*, Carnegie-Mellon University, (1981).

Frank, 1987.
D. Frank, "Functionality of the EXODUS Type Manager," *Internal Working Memo*, University of Wisconsin, (January 1987).

Freytag, 1985.
J.C. Freytag, "Translating Relational Queries into Iterative Programs," *Ph.D. Thesis*, Harvard University, (September 1985).

Freytag, 1986.
J.C. Freytag and N. Goodman, "Rule-Based Transformation of Relational Queries into Iterative Programs," *Proceedings of the ACM SIGMOD Conference*, pp. 206-214 (May 1986).

Freytag, 1987.
J.C. Freytag, "A Rule-Based View of Query Optimization," *Proceedings of the ACM SIGMOD Conference*, pp. 172-180 (May 1987).

Gelenbe, 1986.
E. Gelenbe and G. Hebrail, "A Probability Model of Uncertainty In Data Bases," *Proceedings of International Conference on Data Engineering*, (February 1986).

Gerber, 1986.
R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," *Computer Sciences Technical Report* **672**University of Wisconsin, (October 1986).

Ghosh, 1986.
S.P. Ghosh, "Statistical Relational Tables," *IEEE Transactions on Software Engineering*, (December 1986).

Graefe, 1986.
G. Graefe, "Rule-Based Query Optimization for the Extensible Database System," *Preliminary Proposal*, University of Wisconsin, (September 1986).

Gray, 1981.
J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolo, and I. Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys* **13**(2) pp. 223-242 (June 1981).

Gray, 1987.
J. Gray, *Personal Communication*, (May 1987).

Hanson, 1987.
E.N. Hanson, "A Performance Analysis of View Materialization Strategies," *Proceedings of the ACM SIGMOD Conference*, pp. 440-453 (May 1987).

Horwitz, 1985.
S.B. Horwitz, "Generating Language-Based Editors: A Relationally-Attributed Approach," *Computer Science Technical Report*, (696)Cornell University, (August 1985).

Horwitz, 1986.
S.B. Horwitz, "Adding Relational Databases to Existing Software Systems: Implicit Relations and a New Relational Query Evaluation Method," *Computer Sciences*

Jagadish, 1987.
H.V. Jagadish, R. Agrawal, and L. Ness, "A Study of Transitive Closure as a Recursion Mechanism," *Proceedings of the ACM SIGMOD Conference*, pp. 331-344 (May 1987).

Jarke, 1984.
M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys* **16**(2) pp. 111-152 (June 1984).

Johnson, 1975.
S.C. Johnson, "YACC: Yet Another Compiler Compiler," *Computing Science Technical Report*, (32)Bell Laboratories, (1975).

Kent, 1983.
W. Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," *Communications of the ACM* **26**(2) pp. 120-125 (February 1983).

King, 1981.
J.J. King, "QUIST: A System for Semantic Query Optimization in Relational Databases," *Proceeding of the Conference on Very Large Data Bases*, pp. 510-517 (September 1981).

Klug, 1982.
A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *Journal of the ACM* **29**(3) pp. 699-717 (July 1982).

Kooi, 1980.
R.P. Kooi, "The Optimization of Queries in Relational Databases," *Ph.D. Thesis*, Case Western Reserve University, (September 1980).

Kooi, 1982.
R.P. Kooi and D. Frankforth, "Query Optimization in Ingres," *Database Engineering* **5**(3) pp. 2-5 IEEE, (1982).

Lohman, 1985.
G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, "Query Processing in R*," pp. 31-47 in *Query Processing in Database Systems*, ed. W. Kim, D.S. Reiner, and D.S. Batory,Springer, Berlin (1985).

Lorie, 1983.
R. Lorie and W. Plouffs, "Complex Objects and Their Use in Design Transactions," *Proceedings ACM SIGMOD Database Week*, pp. 115-121 (May 1983).

Lorie, 1985.
R. Lorie, W. Kim, D. McNabb, W. Plouffe, and A. Meier, "Supporting Complex Objects in a Relational System for Engineering Databases," pp. 145-155 in *Query Processing in Database Systems*, ed. W. Kim, D.S. Reiner, and D.S. Batory,Springer, Berlin (1985).

Mackert, 1986.
L.F. Mackert and G.M. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proceedings of the ACM SIGMOD Conference*, pp. 84-95 (May 1986).

Mackert, 1986.
L.F. Mackert and G.M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proceedings of the Conference on Very Large Data Bases*, pp. 149-159 (August 1986).

Maier, 1986.
D. Maier, P. Nordquist, and M. Grossman, "Displaying Database Objects," *Technical Report*, (CSE-86-001)Oregon Graduate Center, (January 1986).

Manola, 1986.
F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems*, pp. 17-25 (September 1986).

Moeller, 1986.
J. Moeller, "Konzeption einer regelbasierenden Planungskomponente fuer die Optimierung von Datenbankanfragen in einer SEQUEL-artigen Sprache," *Diplom Thesis*, Technische Hochschule, Fachbereich Informatik, (January 1986).

Muralikrishna, 1986.
M. Muralikrishna, *Personal Communication*, (Dec. 1986).

Nguyen, 1982.
G.T. Nguyen, L. Ferrat, and H. Galy, "A High-Level User Interface for a Local Network Database System," *Proceedings IEEE Infocom*, pp. 96-105 (1982).

O'Neil, 1986.
P.E. O'Neil, "The Escrow Transaction Method," *ACM Transactions on Database Systems* **11**(4) pp. 405-430 (December 1986).

Pavlovic-Lazetic, 1986.
G. Pavlovic-Lazetic and E. Wong, "Managing Text as Data," *Proceeding of the Conference on Very Large Databases*, pp. 111-116 (August 1986).

Reps, 1984.
T. Reps and T. Teitelbaum, "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Composion on Practical Software Development Environments*, pp. 42-48 (April 1984).

Richardson, 1987.
J.E. Richardson and M.J. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the ACM SIGMOD Conference*, pp. 208-219 (May 1987).

Richardson, 1987.
J.E. Richardson, "E: A Language for Implementing Database Systems," *Preliminary Proposal*, University of Wisconsin, (May 1987).

Rosenthal, 1986.
A. Rosenthal, U. Dayal, and D. Reiner, "Fast Query Optimization Over a Large Strategy Space: The Pilot Pass Approach," *unpublished manuscript*, (1986).

Schwarz, 1986.
P. Schwarz, W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst Database System," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems*, pp. 85-92 (September 1986).

Selinger, 1979.
P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference*, pp. 23-34 (May-June 1979).

Selinger, 1980.
P. Griffiths Selinger and M. Adiba, "Access Path Selection in Distributed Database Management Systems," *Computer Science Research Report*, (RJ2883)IBM Research Laboratory, (August 1980).

Smith, 1975.
J.M. Smith and P.Y.T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Communications of the ACM* **18**(10) pp. 568-579 (October 1975).

Stonebraker, 1975.
M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," *Proceedings of the ACM SIGMOD Conference*, (June 1975).

Stonebraker, 1976.
M. Stonebraker, E. Wong, P. Kreps, and G.D. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems* 1(3) pp. 189-222 (September 1976).

Stonebraker, 1980.
M. Stonebraker, "Retrospection on a Database System," *ACM Transactions on Database Systems* 5(2) pp. 225-240 (June 1980).

Stonebraker, 1983.
M. Stonebraker, B. Rubinstein, and A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Data Bases," *Proceedings ACM SIGMOD Database Week*, pp. 107-113 (1983).

Stonebraker, 1986.
M. Stonebraker and L.A. Rowe, "The Design of POSTGRES," *Proceedings of the ACM SIGMOD Conference*, pp. 340-355 (May 1986).

Tagg, 1984.
R.M. Tagg, "End User Access to Very Large Databases in an Automated Office Workstation Environment," *Proceedings of the Conference on Very Large Data Bases*, pp. 272-279 (August 1984).

Ullman, 1982.
J.D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, MD. (1982).

Warren, 1977.
D.H.D. Warren, L.M. Pereira, and F. Pereira, "PROLOG - The Language and its Implementation Compared with Lisp," *Proceedings of ACM SIGART-SIGPLAN Symposion on AI and Programming Languages*, (1977).

Wong, 1976.
E. Wong and K. Youssefi, "Decomposition - A Strategy for Query Processing," *ACM Transactions on Database Systems* 1(3) pp. 223-241 (September 1976).

Wong, 1982.
H.K.T. Wong and I. Kuo, "GUIDE: Graphical User Interface for Database Exploration," *Proceeding of the Conference on Very Large Data Bases*, pp. 22-32 (September 1982).

Yang, 1985.
D. Yang, "Expections Associated with Compound Selection and Join Operations," *Computer Science Technical Report*, (RM-85-02)University of Virginia, (July 1985).

Zloof, 1977.
M.M. Zloof, "Query-By-Example: A Data Base Language," *IBM Systems Journal* 16(4) pp. 324-343 (1977).