OPTIMIZATION OF EXTENDED RELATIONAL DATABASE SYSTEMS

by

Timoleon K. Sellis

Memorandum No. UCB/ERL M86/58

23 July 1986

OPTIMIZATION OF EXTENDED RELATIONAL DATABASE SYSTEMS

by

Timoleon K. Sellis

Memorandum No. UCB/ERL M86/58

23 July 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

OPTIMIZATION OF EXTENDED RELATIONAL DATABASE SYSTEMS

by

Timoleon K. Sellis

Memorandum No. UCB/ERL M86/58

23 July 1986

ELECTRONICS RESEARCH LABORATORY

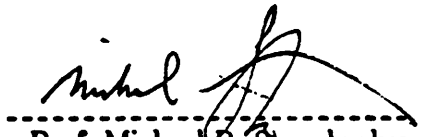
College of Engineering
University of California, Berkeley
94720

Optimization of Extended Relational Database Systems

Timoleon K. Sellis

Optimization of Extended Relational Database Systems

Ph.D. *Timoleon K. Sellis* Computer Science Division
Department of EECS



Prof. Michael R. Stonebraker
Committee Chairman

ABSTRACT

Current relational Database Management Systems (DBMS) must be extended to function well in Engineering and Artificial Intelligence applications. Various additional functionalities have been proposed and in this thesis we study the optimization of one extended environment. Specifically, we consider the optimization of a version of the QUEL query language extended with two new features:

- the repetitive execution of commands, and
- the execution of relation fields in which collections of QUEL commands are stored.

An extended query processing algorithm based on the original INGRES decomposition algorithm is first presented and then various modifications aiming to improve its performance are suggested. Caching of query results is also considered as another means to improve the performance of the processing engine. We analyze and suggest solutions to the various problems related to the design of a query result cache (replace-

ment policies, invalidation techniques, etc).

Based on the above extensions, a relation field may contain more than one QUEL commands. Accessing such a field triggers the execution of all these commands. We present a set of tactics that can be used to reduce the cost of processing multiple commands using some interquery analysis. Special cases amenable to different kind of processing are also identified and studied.

In the case where all commands stored in a field are retrievals from the database, sharing of accessed data is possible. We study the optimization of processing a set of queries in detail, by deriving efficient access plans which take advantage of common intermediate results. Experimental results are also given in support of the proposed algorithms. These results show that significant savings (up to 50%) can be achieved by sharing common data.

Acknowledgements

The effort put into producing this thesis was a result of continuous encouragement and support by my beloved wife, Marilena. Understanding, patience and strong belief in me were reflected in her life all these years. More than that, she has offered me the most valuable gift bridging career and life: our sweet daughter, Stefania. A new meaning for life and a hope for a better tomorrow. For all these, Marilena deserves my deepest love and admiration.

My parents gave me the initial directions in life and continued to remain close to me through joys and sorrows. I am deeply grateful for their continuous support and only hope that I have fulfilled their expectations and added joy to their life. This thesis is the result of my first major effort in life. In recognition to their support, it is dedicated to them.

My advisor Prof. Michael Stonebraker deserves my warmest thanks for his continuous encouragement. During the last three years he has been my major source of unlimited support. In addition to his ability of giving constructive criticisms on every aspect of my academic life, he was always there, ready to help as a real friend. I have been very fortunate to benefit from his experience and technical skills and for all these he deserves my deepest and true gratitude.

Yannis Ioannidis, my "twin brother", is not just a colleague. He is the person I worked with, days and nights, for our undergraduate thesis. The person with whom I shared, more than a house, all my first experiences with graduate studies at Harvard. Without his kind willingness to offer me sincere and rigorous criticisms on my research work, this thesis would not have existed. He is, and will always be, a real friend and I am truly sorry our parallel journeys will have to be temporarilly interrupted with our new careers.

In addition to my advisor and Yannis, the whole INGRES group deserves my wholeheartedly thanks. Professors Eugene Wong and Lawrence Rowe were always there when I needed their valuable opinion and criticism. Margaret Butler, Eric Hanson, Brad Rubenstein, Oliver Gunther, Toni Guttman, Margie Murphy and many others offered me their unlimited support through stimulating discussions on my research work. Joe Kalash and Jeff Anton were the enlighteners for all my problems with the INGRES software. My collaboration with Leonard Shapiro was very pleasant especially because it resulted to my first publication. I am really happy I had the opportunity to work with all of my colleagues and wish to thank them all for their support.

Finally I would like to express my thanks to Professors Eugene Lawler and Jack Silver for their kind interest to serve in my thesis committee and the U.S. Air Force Office of Scientific Research for supporting this research under the grant number 83-0254.

Timos Sellis
July 1986

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

Traditionally Database Management Systems (DBMS) have been used in business applications to efficiently store and organize large amounts of data. The main thrust of database research has focused on designing data structures and algorithms [WONG76,SELI79] so that operations, common in this environment, can be processed efficiently. Recently, there has been considerable interest in extending the use of database management systems into new application areas. In particular, relational DBMSs [CODD70] have been used in support of applications such as text processing [PAVL83,STON83], computer graphics [LORI79], Computer Aided Design (CAD) [LORI81,KATZ82,LORI83,GUTT84b], Artificial Intelligence and Expert Systems [KERS84,KERS86]. The main difference between the business applications and the ones mentioned above lies in the type of information that the two types of applications are using. Business applications are mainly concerned with large volumes of data , while Artificial Intelligence or Engineering Applications usually involve a sophisticated control mechanism that handles relatively smaller amounts of data. Therefore, a system of the second type should be able to support storing and handling control information in addition to data. Our interest is to investigate the possibility of extending current relational database management systems to support storing information of both kinds.

1.1. Why Database Management Systems?

Using a data manager with full capabilities offers the advantages of better data organization, simple user interface, integrity of data in multi-user environments [BERN79,CARE84] and recovering from hardware or software crashes [GRAY78]. Given these advantages, there have been various attempts to build systems that support non-traditional database applications over large volumes of data. In general, there are three different approaches that can be taken

- One can enhance a specific application system (e.g. VLSI design system) with a *specialized* data manager
- One can *interface* a specific application to a general purpose DBMS
- Finally, one can *extend* a general purpose data manager by enhancing it with more sophisticated capabilities (e.g. inference, triggers, etc).

The first approach suffers from two major disadvantages. First, considerable effort must be put into designing and building several modules that DBMSs already include (data definition and data manipulation facilities, query processing algorithms, etc). Second, such specialized data managers are very narrow, in the sense that they cannot be easily modified to support applications other than the ones, for which they were originally written.

In the second approach there is a clean interface between a specialized application program and a general purpose DBMS. The DBMS acts as a server to the application program by supplying on demand the data that the latter requires. However, the major disadvantage of this approach lies in the difficulty to define exactly where

the two systems must be interfaced. As an example, consider the problem of interfacing PROLOG [CLOC81] with a general purpose DBMS. Although that interface appears particularly natural, due to the common theoretical foundations of the two environments, attempts to build such an interface have not been very successful because of the differences in the way each system retrieves its data [WARR81,JARK84a,CERI86]. These attempts to interface PROLOG and general purpose DBMSs make significant changes to the PROLOG query processor trying to improve its performance in an environment where data resides in secondary storage. [BROD84,ZANI84,SCIO84] provide good criticisms of this approach.

Because of the above mentioned difficulties, data managers with extended capabilities have been proposed. In this third approach data manipulation and control functions are integrated into a single system in a *homogeneous* way. As a first example, consider previous work in supporting various semantic data models [MYLO80,SHIP81,ZANI83]. In all these proposals several new constructs were introduced (general objects, classes, unnormalized relations, set-valued attributes, etc). Another similar approach is to design systems based on the object oriented programming paradigm [COPE84,DERR86]. The data manager stores objects that a general program can then fetch and store. Both of these approaches however suffer from two major disadvantages. First, due to the incompatibility between the needs of the various application environments, it is very hard to incorporate all of the above mentioned constructs in a single data manager. Such a system would be extremely complicated and, most probably, inefficient. A second disadvantage is that a complete database management system must be written from scratch. For example, a query optimizer is

needed to support queries. A transaction management system is needed to support shared access and to maintain data integrity in case of software or hardware crashes. Clearly, these modules account for a large fraction of the code that already exists in current DBMSs.

Looking at a different direction, several researchers have proposed other ways of extending relational DBMSs. The basic idea is to come up with a simple system that gives to the user the capability to *build on top* of a basic set of functions whatever constructs are required by specific applications. Moreover, it has been assumed that minimal extensions to the relational model should be attempted. An example of such an effort has been *Deductive Databases* [GALL78]. The direction here is to provide basic support for expert systems applications. In a deductive database system both deductive aspects of the world (*rules*) and asserted information (*facts*) are stored in the same system. The framework represented by logic programming [KOWA74] and typified by the programming language PROLOG, is used as a common example. However, because of the problems mentioned above, various researchers have been engaged in designing extensions of DBMSs instead of trying to interface PROLOG or a general inference engine to a data manager. In [IOAN84,DAYA85,ULLM85,ZANI85] several designs for database systems enhanced with inference capabilities are proposed, each being a specific implementation of the above model of rules and facts. In particular, these systems are distinguished based on the representation they adopt for rules. This approach has been rather successful, the main reason being that relational database systems require minimal extensions to support inference.

Rules have been used in deductive database systems to allow users to incorporate control information in a form other than the simple operators that the relational model offers (e.g. selections, joins, etc). In general, control information can be represented procedurally and/or declaratively. A database can be clearly used for the latter. As a final approach to building extended data managers, the following section describes the idea of extending DBMSs based on the use of procedures.

1.2. Extending Database Systems Using Procedures

Stonebraker et. al. proposed in [STON84] the idea of storing database commands in the database as a means for increasing the functionality of the system. Commands are stored in relation fields and can be accessed as any other field using a slightly extended query language. Moreover, since these commands can be executed, a new operation is introduced allowing a user to execute the contents of relation fields. In that sense, it is suggested that *database procedures* are considered as full fledged database objects. Hence, using this extension of [STON84], the database can be made the single source of information, either procedural or declarative. This is the approach we will take in this investigation also.

To motivate the use of procedures for increasing the functionality of a relational data manager, we give some examples of possible applications.

- *Storing Programs in a Database*

In many applications that use data residing in a database there is a need for code written in the data manipulation language of the DBMS, i.e. database programs. These programs can be stored in the database and then be executed using the DBMS

query language. For example, in [KUNG84] it was shown how a problem like heuristic search can be addressed using such an extended database management system. There, a relation ALGORITHMS(alg_id,alg_type,code) was used, where *alg_id* is a unique identifier, *alg_type* indicates the general class that the given algorithm belongs to (e.g. Dynamic Programming, Branch and Bound, etc.) and *code* is a field used to store the database procedure that implements the algorithm. Therefore the form of the relation ALGORITHMS will be

alg_id	alg_type	code
10	Dynamic Progr.	code line 1 code line 2 .....
15	Dynamic Progr.	code line 1 .....
20	Branch and Bound	code line 1 .....
..	.....	.....

The syntax of the DBMS allows the user to select and execute an algorithm based on its *alg_id* and *alg_type*. Such a syntax may for example be

```
execute (ALGORITHMS.code) where ALGORITHMS.alg_id = 15
```

which will select the Dynamic Programming algorithm with identifier 15 and will process the commands that constitute the body (*code*).

- *Supporting Rules*

Suppose a relation EMP (name,salary,age), with the obvious meanings for its fields, and another relation CATEG_EMPS with the following contents

status	emps
wellpaid	retrieve (EMP.name) where EMP.salary > 80 retrieve (EMP.name) where EMP.salary > 60 and EMP.age < 30 retrieve (EMP.name) where EMP.salary > 65 and EMP.age < 40
underpaid	retrieve (EMP.name) where EMP.salary < 20
..	.....
..	.....

are given. This second relation gives a way to categorize employees according to their salaries or salaries and ages. In some sense it is a set of rules that define when an employee is wellpaid, underpaid, etc. A query asking for wellpaid employees would then be

```
retrieve (CATEG_EMPS.emps.name) where CATEG_EMPS.status = "wellpaid"
```

where the reference to CATEG_EMPS.emps.name will first evaluate the queries stored in the *emps* field of CATEG_EMPS and then project the result of this evaluation on the

name column. More complicated rules can be expressed using the full capabilities of the query language. In addition, general condition-action rules can be defined, since a procedure in a relation field may include update operations as well. Actions can be then implemented through updates to other relations in the database.

- *Supporting Complex Objects*

Complex objects can also be implemented using database procedures. A query expression in a relation field simply describes the way components of other relations (i.e. tuples) are combined to build an instance of a more complex object. As an example, suppose we have a relation POINTS(x,y) describing points on the plane. Another relation LINES(line_id,description) can then be defined, where *description* is a field containing expressions of the form

```
range of POINT,POINT1 is POINTS
retrieve (POINT.x,POINT.y,POINT1.x,POINT1.y) where Qualification
```

Qualification describes how the two points POINT and POINT1 that define a line segment are selected from the POINTS relation. A significant advantage of using procedures for the definition of complex objects is the ability to allow many objects to share the same subobjects. Hence, a hierarchy of objects can be built and inheritance is free since it can be naturally achieved through retrievals of data from the same relations [STON85].

It is clear from the above examples that supporting procedures in a DBMS is of significant importance. POSTGRES [STON86b], a new relational DBMS under development at the University of California, Berkeley, will support procedures as full

fledged database objects. Among other capabilities, the user of POSTGRES can manipulate data, define rules, specify triggers and alerters, etc., using *only* the extended query language that the system provides (POSTQUEL). However, preliminary results in [STON85] show that there is a serious degradation in performance for non-standard data retrieval operations. In addition, there is a need in modifying algorithms that work efficiently in a main memory based system, to algorithms that will work sufficiently well in a database environment [KUNG84,SELL85]. The purpose of this investigation is to study these problems and suggest techniques that improve the performance of extended database management systems.

Optimizing the execution of procedures will be a significant part of this work. Procedures are simply sequences of database commands. However, these commands do not have necessarily to be processed one at a time. Some *interquery optimization* is possible, leading to a more efficient execution. For example, in the special case of read-only procedures where only retrieval commands are used, savings can be achieved by means of common data that the queries may access. In the employee example mentioned above, determining which employees are wellpaid, requires the execution of all three queries stored under the *emps* field of CATEG_EMPS. When processing these queries the intermediate result built for answering the second request and containing the tuples of employees with salaries more than 60K can be used to answer the first query on employees with salary more than 80K. This way the second look-up of the EMP relation is avoided.

Some researchers have studied in the past the problem of multiple query (i.e. procedure) optimization or other related problems. In [GRAN80] and [GRAN81], Grant

and Minker describe the optimization of sets of queries in the context of deductive databases. Roussopoulos in [ROUS82a] and [ROUS82b] provides a framework for interquery analysis based on query graphs [WONG76], in an attempt to find fast view processing algorithms. More recently, Chakravarthy and Minker [CHAK82,CHAK85] have suggested an algorithm based on the construction of integrated query graphs. All of the above proposals assume procedures to be sets of retrieve-only commands. When updates are allowed, different techniques must be used. We propose such techniques in later chapters of this thesis.

1.3. Outline of Thesis

In the remainder of this report we investigate, analyze and solve problems associated with extended relational database management systems. Although the discussion is restricted to the INGRES [STON76] relational DBMS, the ideas are generally applicable to other systems as well.

Chapter 2 begins by describing QUEL+ [STON85], an extension to the query language QUEL used by INGRES. QUEL+ introduces two new features. First, a new operator that allows repetitive execution of database commands is introduced. This way, iterative constructs can be embedded in database procedures. The second feature introduced, is the ability of the system to support procedures by means of storing query language expressions in relation fields. Chapter 2 then continues with a detailed discussion on how query processing should be done in light of these extensions. A variation of the original INGRES decomposition algorithm [WONG76] is first presented. Then various improvements to this algorithm are discussed. These

improvements aim at producing more efficient access plans for some special classes of queries.

The query processing algorithm deals only with the problem of generating efficient access plans to process a given query. Other ideas that can improve the performance of a system that supports procedures are also discussed in Chapter 2. First, we examine the idea of storing results of previously processed procedures in secondary storage. That idea is called *caching* of procedure results [STON85]. Using a cache, the I/O and CPU cost of processing a query can be reduced by preventing multiple evaluations of the same procedure. Problems associated with cache organizations are examined in depth. Policies for replacing entries of the cache with newly produced procedure results along with algorithms that decide if a given result should be cached, are discussed. However, results of procedures may become invalid when relations used in the evaluation of a procedure are updated. The problem of checking the validity of cached entries is also examined. Finally, schemes for efficient searching of the cache are discussed.

Another means for reducing the execution cost of queries is indexing. Indexes are used in DBMSs to provide efficient access to relations. When procedures are evaluated, the fields of the resulting relations can also be indexed. However, at any given time, it is highly probable that not all procedures stored in a relation have been evaluated. Therefore, a conventional indexing scheme cannot be used, for it would assume that all values resulting from the execution of procedures are known. As a solution to that problem, a new indexing scheme, Partial Indexing, is proposed and analyzed. A partial index contains information only on results of procedures that have

been materialized in the past. Uses of partial indexes in conventional database systems are also described.

Chapter 2 deals with the problem of efficiently processing queries that reference results of procedures. These procedures are simply sequences of database commands. How to efficiently process the procedures themselves is also an interesting issue. It was mentioned in the previous section that some interquery optimization is possible. Chapters 3 and 4 investigate this problem and propose algorithms for processing multiple database commands. Chapter 3 examines general database procedures where update as well as retrieval operations are possible. Several transformations and optimization techniques are suggested. Some of them are drawn from the area of compiler design where similar problems have been examined in the context of general programming languages (e.g. moving loop invariants out of loops). Others are extensions to conventional query processing or physical database design techniques. Cases where special transformations are possible are also identified and studied. Although such transformations are not applicable to all kinds of procedures, they are very important to several engineering applications [KUNG84].

Chapter 4 studies a special case of procedure optimization, where only retrieval commands are used. In this case, savings can be achieved by means of common data that the queries may access. The model that will be assumed for queries is first described and then an analysis of several algorithms that perform some interquery analysis and suggest efficient access plans is given. These algorithms differ in the amount of time one is willing to spend to preprocess a given set of queries. There is a trade-off between the time required for interquery optimization and the actual cost for

executing the queries. Such issues are also discussed in depth. We then present some experimental results that show that multiple query optimization is useful and can significantly improve the performance of systems that support database procedures. Finally, in Chapter 5, a summary of our results is given along with some discussion on important problems for future research.

CHAPTER 2

QUEL+ : THE LANGUAGE AND HOW TO PROCESS QUERIES

2.1. Introduction

This chapter examines the approach of extending a database manager to handle not only data but control information as well. We will first present the structure of QUEL+ [STON85], which is an extension to QUEL, the query language designed for INGRES [STON76]. There are two major extensions made to QUEL:

a) repetitive execution of commands, and

b) storing query language commands in relation fields

The first extension allows the user to implement iteration using the query language itself instead of escaping to a general purpose programming language. In EQUEL/C [ALLM76] for example, the programmer can embed INGRES commands in C [KERN78] programs and therefore can implement iteration through the iterative constructs of C. The second feature follows the paradigm of LISP [WILE84] and allows the uniform treatment of data and control information, or procedures in [STON85], where the latter is implemented using database commands.

Physical and conceptual modeling, query processing, concurrency control and crash recovery are some of the well known DBMS problems [ULLM82]. The solutions to many of these problems can still be used in the QUEL+ environment. However, performance will deteriorate due to the complexity of the new operations. Our goal in

this chapter is to examine ways of improving the performance by providing more sophisticated optimization tactics. More specifically, we concentrate on the problem of query processing. Issues that deal with user interfaces, physical and conceptual modeling, consistency in a multiple user environment and robustness, are examined in more detail in [STON86b] in the context of the design of a new DBMS being developed at the University of California, Berkeley, called POSTGRES.

This chapter is organized as follows. Section 2.2 presents the language QUEL+ and motivates its use with a set of examples. Then, in section 2.3 we study the problem of query processing by presenting first a simple algorithm and then proposing a set of possible improvements. Sections 2.4 and 2.5 present ideas on supporting schemes that improve the performance of the system, like caching and indexing. Finally, we conclude in section 2.6 by summarizing the discussion of this chapter.

2.2. The Query Language QUEL+

As mentioned above, the major extensions that are introduced to QUEL+ are the repetitive execution of standard QUEL commands and storing QUEL commands in relations fields. [STON85] gives a detailed discussion of the language. We review here some of the extensions that will serve as the basis of our presentation.

2.2.1. Iterative Execution of QUEL Commands

Iterative execution of commands was first introduced to INGRES by Guttman in [GUTT84b]. Guttman mainly used the iterative version of the append command in order to express queries that produce the transitive closure of a binary relation, in his case, parts explosion in a VLSI design environment.

To motivate the use of iterative execution, we use the following example of a relation EMP (name,salary,mgr), with the obvious information about employees. The goal is to perform an update on the EMP relation, so that all employees that *eventually* work for Smith (through the manager hierarchy), change their *mgr* field to Smith. For example, given the following EMP relation

name	salary	mgr
Stones	20K	Smith
Jones	10K	Stones
Lam	15K	Riggs
Felps	10K	Jones

it is required that the *mgr* field values be modified, yielding the following relation

name	salary	mgr
Stones	20K	Smith
Jones	10K	Smith
Lam	15K	Riggs
Felps	10K	Smith

One way this can be achieved, is by repetitively executing the command

```
range of EMP,EMP1 is EMP
replace EMP (mgr = "Smith")
   where EMP.mgr = EMP1.name
   and   EMP.mgr = "Smith"
```

until it fails to modify EMP. In QUEL+ we add a * (asterisk) to a standard QUEL command and introduce repetition with the following semantics

To process command*, *process* command *repetitively until it has no further effect on the database*

The above semantics do not necessarily imply that the command will be processed by iterative execution. The work of [GUTT84b] and [IOAN86] shows that iterative execution of the same operation is not always the most efficient way process transitive closure commands in a database environment. Using the * extension, we can perform the above update with the single QUEL+ command

```
range of EMP,EMP1 is EMP
replace* EMP (mgr = "Smith")
   where EMP.mgr = EMP1.name
   and   EMP.mgr = "Smith"
```

This shorthand notation not only simplifies the user interface but also gives the flexibility to the query optimizer to optimize the loop as a unit instead of a single `replace` command.

2.2.2. QUEL as a Data Type

It was first proposed in [STON84] that QUEL commands be stored in relation fields in the same way data is stored in relations. For simplicity, these fields are thought as variable length strings. In INGRES, relation fields can be accessed individually through the dot (.) operator. For example, `EMP.mgr` in the above command accesses the manager names recorded in `EMP`. Extending these semantics, it will be assumed that accessing a relation field containing QUEL commands (*QUEL field*) implies the execution of the commands that are stored in the field. In addition to that accessing mechanism, a new QUEL+ command, called `execute`, is allowed. The

semantics of

```
execute (Relation.fd) where Qualification
```

where *fd* is a QUEL field of some relation Relation, is to process the commands stored in the *fd* field of those tuples in Relation that satisfy the Qualification. Through execute, the user can explicitly request the execution of specific commands. For example, in [KUNG84], an ALGORITHMS relation is defined where specific implementations of algorithms that solve the shortest path problem are stored in the form of sequences of QUEL commands (or database procedures). Using execute, a user can then select and process any of these procedures.

In light of these two extensions, we differentiate for processing reasons between two types of QUEL fields.

a) collections of retrieve-only commands (queries), or

b) collections of general QUEL commands (i.e. queries and updates)

In the first case the result of processing the queries is a set of relations that the user has requested while in the second case updates may be performed on the database and no specific result is returned. Processing QUEL fields amounts to evaluating the commands that are stored in these fields. As mentioned in the introduction, we study the problem of efficiently evaluating the contents of QUEL fields in Chapters 3 and 4. Here, we will concentrate on the problem of processing QUEL+ queries.

We motivate the discussion that will follow in the next section on the problem of processing QUEL+ queries by using an example. Consider, a relation EMP (name.salary.mgr.hobbies) where *name, salary* and *mgr* are conventional

fields while *hobbies* is a field of type QUEL. We use *hobbies* to retrieve data on the various hobbies of employees. Assume also that the following relations exist in the system

```
SOFTBALL (name,position,performance)
SOCCER   (name,position,goals,performance)
MUSIC    (name,instrument,performance)
```

Adding Jones as an employee can be done now as follows:

```
append to EMP (name = "Jones",salary = 40K,mgr = "Smith",
               hobbies = "retrieve (SOFTBALL.position,SOFTBALL.performance)
                            where SOFTBALL.name = "Jones"
                          retrieve (SOCCER.position,SOCCER.performance)
                            where SOCCER.name = "Jones""
             )
```

It is assumed that the corresponding entries for Jones have been already inserted in SOFTBALL and SOCCER. An instance of the EMP relation after the above insertion of the above tuple will be

name	salary	mgr	hobbies
Riggs	20	Smith	retrieve (SOFTBALL.position,SOFTBALL.performance) where SOFTBALL.name = "Riggs"
Jones	30	Smith	retrieve (SOFTBALL.position,SOFTBALL.performance) where SOFTBALL.name = "Jones" retrieve (SOCCER.position,SOCCER.performance) where SOCCER.name = "Jones"
Lam	80	Moore	retrieve (MUSIC.all) where MUSIC.name = "Lam"
..	....	....	.....

We discuss how fields of type QUEL are accessed and used in queries in the next

subsection.

2.2.3. Using QUEL Fields in Queries

The QUEL syntax is extended using the *multiple dot* notation borrowed from Zaniolo's GEM language [ZANI83,ZANI84]. For example, one can retrieve the performance of Jones in all his hobbies as follows:

```
retrieve (EMP.hobbies.performance)
    where EMP.name = "Jones"
```

The number of dots that can be used depends on the relation nesting level. With the use of the multiple dot notation, QUEL+ allows the user to actually "navigate" through relations using QUEL fields as links between the accessed tuples.

Clearly, the result of evaluating (*materializing*) a QUEL field is a set of relations, or in general a set of tuples. These sets are themselves database objects (relations). It is very natural for a user to be able to use these objects as parts of his/her queries. For example, one may wish to get all pairs of employees that play in the same positions and with the same performance in their hobbies. QUEL+ supports the most common set operators like set equality, set inequality, union, intersection and containment as well as database oriented operators like the outer and natural join. The above query can then be formulated as

```
range of EMP,EMP1 is EMP
retrieve (EMP.name,EMP1.name)
    where EMP.name ≠ EMP1.name
    and   EMP.hobbies == EMP1.hobbies
```

where == is the set equality operator. We briefly discuss here some issues on the implementation of such operators.

The relation level operators can be implemented in either of two ways. First, one can write specialized routines. These routines must of course be coded to work efficiently in a database environment where whole pages are read and written as a unit. This approach seems to be rather straightforward, with the only disadvantage that some (considerable) effort must be put in writting this code. The second way is based on the fact that one can use the expressive power of the query language to write programs that implement the set operators. This approach requires minimal effort and no substantial extension to the query optimization code, since the only thing that is needed is the capability to issue queries from within the system itself. It is also similar to the approach taken in [WONG85] for extending relational database systems with new types and operators. To give an example, let us assume that we want to find out if two QUEL fields evaluate to identical relations. After processing the left and right hand operands, two relations R1 and R2 respectively will be produced. Checking if R1 == R2 can be done using the following QUEL query

```
/* assume that it returns 1 if they are equal, null otherwise */

retrieve (true=1) where
count (R1.TID) = count (R1.TID where R1.fd_1=R2.fd_1 and
                                    R1.fd_2=R2.fd_2 and
                                    ..........
                                    R1.fd_n=R2.fd_n)
```

where it has been assumed that relations R1 and R2 have fields *fd_1, fd_2, ...,fd_n* and *TID* is a unique *T*uple *ID*entifier that is used to augment every tuple in the database. Similarly, one can derive QUEL queries for the rest of the relation level operators.

After reviewing the structure and semantics of QUEL+, we now examine the problem of query processing. As mentioned above the analysis is restricted to QUEL fields containing retrieve-only commands.

2.3. Processing QUEL+

This section presents a query processing algorithm that INGRES can use to evaluate QUEL+ queries. First, it discusses how the original decomposition algorithm of Wong and Youssefi [WONG76] was extended to handle queries in relation fields and the extended relation level operators. An example is also used to illustrate the flow of the algorithm. Then, some possible improvements are suggested and explained through examples.

2.3.1. Extended Decomposition

Figure 2.1 shows a diagram of the extended decomposition algorithm as suggested in [STON85]. The modifications done to the original Wong-Youssefi algorithm can be summarized as follows

a) All one-variable clauses except those that include a multiple dot reference or a relation level operator are processed first. The reason is that clauses involving extended operators cannot be processed efficiently. For example, none of the following two clauses

```
EMP.hobbies.position = "catcher"
```

or

```
EMP.hobbies == some_constant_relation
```

should be processed first because that would imply the materialization of the *hob-*

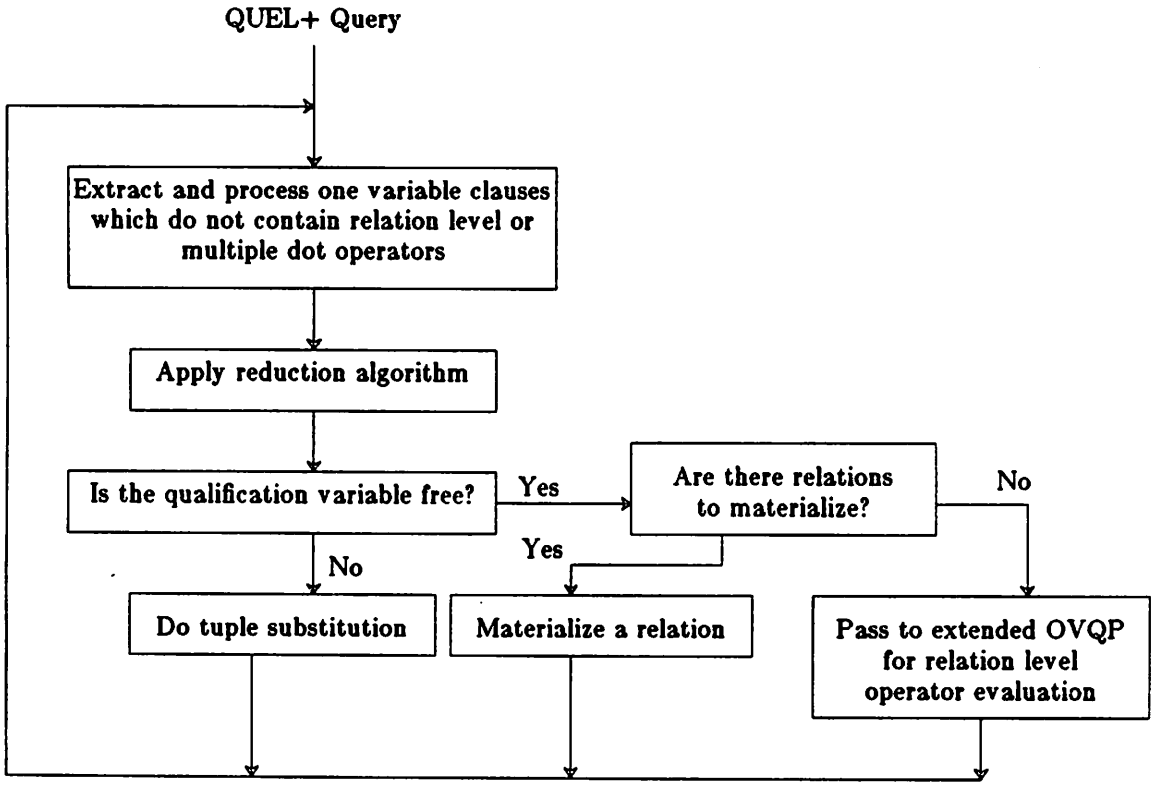


Figure 2.1: Extended Decomposition Strategy
(OVQP : *One Variable Query Processor*)

bies entries of all employees, which is very expensive. An exception to that is the case where an index exists on `EMP.hobbies.position`. This case is discussed in more detail in section 2.5.

b) An extra step is required to check if all QUEL field entries have been materialized. Materialization is done by passing the queries found in the QUEL field to a second INGRES process which in turn returns the result relation(s). The decomposition algorithm continues processing one-variable clauses and materializing

QUEL fields until no more such fields are left unevaluated.

c) In [WONG76] the criterion for selecting a relation to iterate over in the case of tuple substitution, is the size of the relations. The presence of QUEL fields makes this criterion ineffective. Not only the number of tuples but the cost for materializing the corresponding QUEL fields should be considered. The reason is that during tuple substitution, each tuple variable will be replaced with specific field values read from the relation. In case of QUEL fields these values are the materialization results. Therefore the criterion for selecting a relation to iterate over will generally be a function of the size of the relation and the characteristics of the materialized objects. One of these characteristics which is of major importance is the ability of the system to keep materialized objects in secondary storage, i.e. caching. This aspect is treated in more detail in section 2.4.

To illustrate the extended decomposition algorithm, a detailed example is now presented. Given the EMP relation of the previous section, we are looking for the names of employees that play as catchers, play in the same positions and with the same performance with their managers and these managers are well paid. In QUEL+ this is expressed as

```
range of EMP,EMP1 is EMP

retrieve (EMP.name)
    where EMP.hobbies == EMP1.hobbies
    and   EMP.hobbies.position = "catcher"
    and   EMP.mgr = EMP1.name
    and   EMP1.salary > 70
```

Following the flow chart of Figure 2.1, we first identify the one-variable clause on

salary and process it

```
retrieve into TEMP1 (EMP.name,EMP.hobbies) where EMP.salary > 70
```

The new query is now

```
retrieve (EMP.name)
    where EMP.hobbies == TEMP1.hobbies
    and   EMP.hobbies.position = "catcher"
    and   EMP.mgr = TEMP1.name
```

Notice that the other one-variable clause EMP.hobbies.position = "catcher" is not processed, since that would require materialization of all *hobbies* entries in EMP. Continuing, we find that no reduction is possible. Since there are still variables in the query, tuple substitution must be performed. Assume that iteration is done over TEMP1. Then the query becomes

```
retrieve (EMP.name)
    where EMP.hobbies == QUEL-constant-1
    and   EMP.hobbies.position = "catcher"
    and   EMP.mgr = constant-1
```

QUEL-constant-1 is now a collection of QUEL commands that were stored in the *hobbies* field of TEMP1. Since the above query now has a one-variable clause, we process that first

```
retrieve into TEMP (EMP.name,EMP.hobbies) where EMP.mgr = constant-1
```

changing the query to

```
retrieve (TEMP.name)
    where TEMP.hobbies == QUEL-constant-1
    and   TEMP.hobbies.position = "catcher"
```

Processing again returns to tuple substitution and variable TEMP is chosen. Substitut-

ing the fields for their values we get

```
retrieve (constant-2)
    where QUEL-constant-2 == QUEL-constant-1
    and   QUEL-constant-2.position = "catcher"
```

Now the query has no variables and is passed to the materialization module. If QUEL-constant-2 is chosen the resulting query will be:

```
retrieve (constant-2)
    where TEMP3 == QUEL-constant-1
    and   TEMP2.position = "catcher"
```

As pointed out in [STON85] QUEL-constant-1 is not changed to TEMP2 in both occurrences, the reason being that TEMP2 will be processed separately to check if the second qualification clause is satisfied. As a result, TEMP2 will be reduced to being only the tuples with position="catcher", which would make impossible to check the first condition (==) correctly. That is why two variables ranging over the same relation were introduced. Should we have liked to avoid that, the above original query could have been expressed with a clause that checked if catcher was contained in the list of positions an employee plays. That is, use

```
and EMP.hobbies.position >> ("catcher")
```

where >> is the containment operator. Then one tuple variable would be enough since the modified OVQP (One Variable Query Processor) would handle that clause by simply returning *true* or *false* and not altering TEMP2. Generally, more than one tuple variables need be introduced if the same QUEL-constant appears in both simple selection or join clauses that include relation level operators. However, the latter must have only one level of reference (i.e. one dot). For example,

```
and EMP.A1.A2.A3 = constant-value
and EMP.A4 > constant
```

will be changed to

```
and TEMP.A2.A3 = constant-value
```

where TEMP is the set of employees with values of A4 higher than constant. Then, in the next iteration, two different variables will be used to substitute for TEMP.A2.

Returning to our example, we see that the new query now has a one-variable clause which can be detached and processed. If TEMP2 does not contain "catcher", the query is false and will be terminated. Otherwise, we continue with the query

```
retrieve (constant-2)
    where TEMP3 == QUEL-constant-1
```

Now, there is just one more QUEL field (QUEL-constant-1) to materialize, yielding

```
retrieve (constant-2)
    where TEMP3 == TEMP4
```

This is a variable-free query that must be passed to the one-variable query processor. This module will process the operator == for the two relations involved and if it returns *true*, the value constant-1 can be returned to the user.

The above extended decomposition algorithm delays materializing a QUEL field until there is nothing else that the conventional query processor can do. Even tuple substitution must be done first, the reason being that checking a condition that involves multiple dot references implies a loop over all tuples in the relation. During that loop QUEL fields are materialized and checked through lower level fields. Generally, the absence of any information about the contents of relations in QUEL fields

makes optimization very hard, if not impossible. In the next section we discuss one possible improvement through saving the results of materializing QUEL fields (*caching*); in this case, the contents of QUEL fields are known and conventional cardinality estimation methods [SELI79] can be used to estimate the cost of the various processing strategies. However, before moving to caching we suggest some other possible improvements that apply directly on the algorithm itself.

2.3.2. Improvements to Extended Decomposition

In this subsection some possible improvements to the algorithm presented above are examined. First, we give some rules that can be applied in general; then, some other special case transformations that can be used are outlined.

The first general rule as, suggested above, is to process one-variable clauses and do reduction as the initial Wong-Youssefi algorithm proposes [WONG76]. This will certainly be the best thing to do independent of the number of relations or QUEL field materializations that will follow. The problem arises when tuple substitution is necessary. We motivate our proposal using an example.

Let us assume that in the EMP relation the *hobbies* field produces a relation, which itself has a field *performance* that also produces a relation as a result and the field we are interested in is the *location* field of that last relation. We also assume the existence of another relation DEPT (name,mgr,location). The query is

```
retrieve (EMP.name,DEPT.name)
    where EMP.hobbies.performance.location = DEPT.location
    and   EMP.mgr = DEPT.mgr
```

The question that arises here is over which relation to iterate doing tuple substitution.

The main idea behind tuple substitution is to introduce single variable selection clauses as early as possible. Using such clauses relation sizes are reduced and, consequently, the number of materializations that will be needed is also lower. For example, in the above query tuple substitution should be done over DEPT independently of the sizes of the two relations. The following analysis supports this decision. Let |EMP| and |DEPT| be the cost of scanning the relations EMP and DEPT respectively. For simplicity we will assume here that the cost of processing a one-variable clause is equal to the cost of scanning the relation, while the cost of processing a join between two relations is equal to the product of the costs of scanning each of these relations. The reason for making such assumptions is to simplify the analysis that follows. We discuss in the end of the paragraph how general cost functions can be used in the presence of indexes or other join algorithms (e.g. merge scan). Also let SEL_E be the percentage of EMP tuples that satisfy a constraint EMP.mgr=DEPT.mgr for the various departments and SEL_D be the percentage of DEPT tuples that satisfy a constraint DEPT.mgr=constant. Finally, it will be also assumed that the cost of producing EMP.hobbies.performance for the various employee tuples is M and S is the average size of the resulting relation (i.e. S=|EMP.hobbies.performance|). Based on the above, we now analyze the cost of processing the above query by tuple substituting either over EMP or DEPT.

a) *Tuple substitute over* EMP: For each EMP tuple, process the query

```
retrieve (constant,DEPT.name)
    where QUEL-constant.performance.location = DEPT.location
    and   constant-1 = DEPT.mgr
```

Since |DEPT| is the cost of processing the one-variable clause and assuming that materialization results are kept in secondary storage to avoid re-evaluation of QUEL fields, the cost of processing each employee tuple will be

```
|DEPT|          +   /* cost of doing the one-variable selection */
  M             +   /* cost of materializing the QUEL field */
S*|DEPT|*SEL_D =   /* cost of doing the join */

              = |DEPT|*(1+S*SEL_D) + M
```

for a total of

$$|EMP|*|DEPT|*(1+S*SEL\_D) + |EMP|*M \qquad (1)$$

b) *Tuple substitute over* DEPT: For each DEPT tuple, process the query

```
retrieve (EMP.name,constant)
    where EMP.hobbies.performance.location = constant-1
    and   EMP.mgr = constant-1
```

Again under the above assumptions, for each department tuple the cost will be

```
|EMP|          +   /* cost of doing the one-variable selection */
|EMP|*SEL_E*M  +   /* cost of materializing the QUEL fields */
|EMP|*SEL_E*S  =   /* cost of doing the final one-variable selection */

              = |EMP|*(1 + SEL_E*(M+S))
```

and assuming that re-materialization of the same field is never needed, the total cost will be

$$|EMP|*|DEPT|*(1+S*SEL\_E) + |EMP|*SEL\_E*M \qquad (2)$$

Subtracting (2) from (1) we get

$$DIFF = |EMP|*|DEPT|*S*(SEL\_D-SEL\_E) + |EMP|*M*(1-SEL\_E)$$

and considering the second factor to be much more significant because of the high

materialization cost, we may conclude that it is better to tuple substitute over the relation that will cause the least number of materializations, in our example DEPT since SEL_E<1. The reason for that is that tuple substitution will create some one-variable clauses which can then be used to restrict the number of tuples that need to be considered for materialization of their fields (in the above case that was EMP).

Returning now to the simplistic assumptions made for the cost of processing one-variable clauses and joins between two relations we can see that the above analysis still holds. However, the formulas are not that simple any more. In general, the cost of doing the one-variable selection on a relation R is a function F(|R|) and the cost of doing a join between two relations R1 and R2 will be J(|R1|,|R2|). Hence the two corresponding formulas for (1) and (2) will be

$$|EMP|*F(|DEPT|) + |EMP|*J(|S|,|DEPT|*SEL\_D) + |EMP|*M \qquad (1a)$$

and

$$|DEPT|*F(|EMP|) + |DEPT|*|EMP|*SEL\_E*|S| + |EMP|*SEL\_E*M \qquad (2a)$$

Evaluating these two formulas and checking their difference will indicate which plan is preferable. However, if we assume that still the materialization cost M is the primary factor in the above, DEPT will be the best candidate for tuple substitution.

In general, an algorithm that selects a relation to iterate over, attempts to minimize the total number of tuple substitutions required, assuming the most expensive processing lies in QUEL field materializations. Such an algorithm would go as follows. Let V be the set of all non one-variable clauses. Assume also the existence of at least one clause of the form $R_1.fd_1 = R_2.fd_2$. Such clauses are called *simple*. Let

$TS(C,R_1)$ be the number of tuple substitutions required over R_1 for the clause C to be evaluated. In other words, $TS(C,R_1)$ is the number of dots in the reference to relation R_1. For example, assume that we have the clauses

(C1) EMP.hobbies.performance.location = DEPT.location

and

(C2) EMP.mgr = DEPT.mgr

Clearly, three tuple substitution loops must be executed over EMP in order to make the first clause effective. Hence, TS(C1,EMP) = 3. DEPT can become effective with only one substitution, i.e. TS(C1,DEPT) = 1. Considering the second clause, both EMP and DEPT need only one tuple substitution; therefore, TS(C2,EMP) = 1 and TS(C2,DEPT) = 1.

Next we compute

$$diff(R) = \max_{C \in V} TS(C,R)$$

for each relation R involved in some clause. Intuitively, these numbers measure the *difficulty of processing* the query depending over which relation tuple substitution is performed. This difficulty is considered to be mainly due to the number of tuple substitutions required to reach ground relations, i.e. relations with no QUEL fields. Suppose that R_M is the relation with the minimum *diff* value, i.e. the relation such that $diff(R_M) \leq diff(R)$, for all R that are involved in simple clauses. We choose to tuple substitute over the relation R_M (in case of ties we favor the smaller relation). For example, in the example mentioned above, we will have $diff$(EMP)=3 and $diff$(DEPT)=1 and we choose to tuple substitute over DEPT due to clause C2. It is straightforward to

show with an analysis similar to the one presented for the example that this is the best tuple substitution strategy.

The above algorithm gives a rigorous way of selecting the relation over which tuple substitution will be done. In cases where every clause involves at least one relation accessing a QUEL field, i.e. there are no simple clauses, the above algorithm will not work. However, these are not of major interest since one way or the other the entries will all be materialized during tuple substitution. Notice also that in the above analysis, three basic assumptions have been made. First, computed results were kept in secondary storage to prevent multiple materializations of the same entries. Second, the materialization cost M was dominating any other cost in our formulas (1) and (2). Finally, the costs for processing one-variable and join clauses were very simplistic. In general, formulas (1) and (2) will have two factors. One is the estimated cost for doing the join between the two relations EMP and DEPT by tuple substituting over either of the relations. This factor is determined using conventional cost estimating techniques [SELI79]. In the general case, the cost M may not dominate all other cost factors. Then, in order to compare the costs of the two processing strategies, some estimate for the cost of materializing a given QUEL field is needed. This cost can be calculated using standard techniques, at the time tuples with QUEL fields are inserted. If for efficiency reasons preprocessing of queries at insertion time is not possible, some kind of off-line processing can compute the estimated costs and store them along with the QUEL fields. In any way, the query processor will have two specific estimated values for the costs of the two strategies derived from formulas similar to (1) and (2). Comparing these values and selecting the minimum one will suggest the most efficient

processing strategy.

Let us now describe a different technique that can be used to improve the performance of the query processor. The basic idea is that when an entry from a QUEL field is materialized, the query that has to be processed next is known. More specifically, the structure of the query is known and through that the optimizer can identify access structures that may be desirable in order to speed up processing. For example, in the query

```
retrieve (EMP.name,DEPT.name)
    where EMP.hobbies.performance.average = 10
    and   EMP.mgr = DEPT.mgr
    and   EMP.hobbies.leader = DEPT.mgr
```

the algorithm outlined above, will choose to tuple substitute over DEPT, the new query being

```
retrieve (EMP.name,constant-1)
    where EMP.hobbies.performance.average = 10
    and   EMP.mgr = constant-2
    and   EMP.hobbies.leader = constant-2
```

Finally, after the detachment of the one-variable clause the following query will be processed

```
retrieve (TEMP.name,constant-1)
    where TEMP.hobbies.performance.average = 10
    and   TEMP.hobbies.leader = constant-2
```

At this point the query processor will start materializing entries from the *hobbies* field of TEMP. Let TEMP1 be the result of materializing a specific entry of *hobbies*; then the type of queries that will have to be processed for each TEMP tuple will be

```
retrieve (constant-2,constant-1)
```

```
where TEMP1.performance.average = 10
and   TEMP1.leader = constant-2
```

From that last query one can observe that depending on the size of TEMP1 it may be beneficial to build a secondary index on *leader* so that the second qualification clause can be processed efficiently. This structure will be built in the process of producing TEMP1 (*on the fly*) and no extra time need be spent at the time the query will be evaluated. Dynamic creation of indexes or imposing other structures on relations (like sorting) has also been used in conventional query processing [YOUS78,KOOI82]. However, a difference is that in the QUEL+ environment no significant additional cost need be spent on creating the index. At the same time a result of a materialization is produced and stored in a temporary relation, some adequate organization is chosen or a secondary indexing structure is built.

In the same spirit we describe another optimization technique that can be used to reduce the cost of processing a query. Clearly, one wants to materialize QUEL fields and produce results that will be used subsequently in the course of processing a given query. However, in some cases, not all queries stored in QUEL fields will give relevant information. For example, consider the relation EMP (name,salary,mgr,hobbies) of the previous section, and the query

```
retrieve (EMP.name)
    where EMP.hobbies.instrument = "violin"
```

When the various entries in the *hobbies* field are materialized, only those queries that involve in their result a field *instrument* should be evaluated. In our example, the queries that retrieve data from the SOFTBALL and SOCCER relations should not be

evaluated. Checking which queries are useful is not hard. It amounts to simply checking the target list (projection fields) of each query. Moreover, even if the query in *hobbies* retrieves many fields from the MUSIC or any other relation that includes a field *instrument*, the contents of the materialized relations should be restricted to contain only the information that is absolutely necessary, in this case the *instrument* field. This way the size of the materialized objects is kept as small as possible which is especially crucial in the case where these objects are kept in secondary storage. We should also notice here that the same idea exists in conventional query processing as well. When intermediate relations are built as the result of processing a one-variable clause or a join, the fields that are projected in such a relation are the ones that are needed either to form the final result of the query or to continue processing the query [WONG76,SELI79].

The above technique tries to reduce the amount of space required for storing materialized objects. However, there are some cases where *no* space at all need be allocated for materialization. This is the case where a QUEL field contains a single `retrieve` or `define view` command. In this special (but very common case) there is no need to even produce the result of the command. What we propose to do is to simply transform the original query in the same way conventional query modification [STON75] does in view processing and integrity constraint enforcement. For example, consider the following query

```
retrieve (EMP.hobbies.position)  where  EMP.hobbies.average ≤ 5
```

and the *hobbies* field of the EMP relation contains one of the following QUEL expres-

sions

```
    retrieve (SOCCER.all) where SOCCER.name = constant
or
    retrieve (SOFTBALL.all) where SOFTBALL.name = constant
```

i.e. all employees have at most one hobby. Then the given query can be transformed to

```
retrieve (REL.position)
    where REL.average ≤ 5
    and   REL.name = constant
```

where REL is either SOCCER or SOFTBALL. This transformation not only prevents the query processor from materializing relations, but it also allows the optimizer to have more information on the structure of the query, and therefore to process it with a better access plan. It is possible to generalize this technique to handle multiple statements but only in the case where all queries in the QUEL field are returning data from exactly the same relation. Then the transformed query will be simply the disjunction (or in QUEL) of smaller subqueries like the one we used in the above example. Section 4 of Chapter 3 discusses this transformation in a different context.

This concludes our presentation of the extended decomposition algorithm for processing QUEL+ queries. In addition to the basic algorithm, we presented some less general tactics that can be used to improve the performance of the query processor. In the two sections that follow two other issues that are of significant importance to query processing are discussed, namely *caching* and *indexing* of the results of QUEL fields.

2.4. Caching Materialized QUEL Fields

As it was seen in the previous section, materializing an entry of type QUEL amounts to executing, possibly several, QUEL queries. Hence, it will be generally very slow to perform this operation every time a QUEL field is accessed. This section examines ways to make QUEL+ processing more efficient through the use of a cache.

2.4.1. What is Caching?

We mentioned at several points in the previous sections that one way to avoid evaluating the same QUEL field entries multiple times, is *caching*. By caching we mean computing the values of QUEL fields and storing them in some specifically assigned area of secondary storage. This computation can be done either at the time tuples are inserted in relations or the first time they are referenced. We will call the former *precomputation* of QUEL field entries since it occurs before even the content of the specific field is accessed. However, our focus here is on the latter case which is more natural. The basic idea is to keep in secondary storage materialized objects that are frequently used in queries. Under that formulation, the caching problem is conceptually the same with the well known caching problem in operating systems [MATT70]. Notice also, that the cache can be used not only for materialized QUEL fields but for generally holding the results of any query issued by the user. These can be saved because either the same query may be given by a user frequently or they can be used to answer other queries [FINK82,LARS85,SELL86].

The caching problem introduces several subproblems to be solved. The following list is the set of issues that will be discussed in this section.

a) *Which query results to cache?*

b) *What algorithm should be used for the replacement of cache entries?*

c) *How to check the validity of a cached object?*

d) *How to index the entries of the cache?*

We will assume that the general model of the cache is a limited area in secondary storage where entries of the form

```
(Qid,Query_expression,Result)
```

are stored. `Qid` is some unique identifier, `Query_expression` is some canonical representation for queries, e.g. query graphs [WONG76], and `Result` is the relation resulting after executing the query or set of queries that were found in some QUEL field and described by the second field (`Query_expression`). The following four subsections give answers to each of the above mentioned questions (a) through (d).

2.4.2. Which Query Results to Cache?

Depending on the information known about the queries, the system can decide whether a result is worth caching it or not. For a given materialization result R, this decision will be generally based on the frequency of references to R, the frequency of updating the relations used to build R and the costs for computing, storing and using R. Specifically, the following is the list of parameters to the caching problem

Caching Problem Parameters	
$\mathbb{C}$	Size allocated for the cache
r_i	Probability of referencing result R_i
u_i	Probability of updating R_i
M_i	Cost of producing R_i (materialization)
S_i	Cost of writting R_i in the cache
U_i	Cost of using R_i from the cache
$\mid R_i \mid$	Size of R_i
IN	Cost of invalidating a cache entry

Table 2.1: Caching Problem Parameters

$\mathbb{C}$ is the number of disk pages allocated for the cache. r_i and u_i are the probabilities of referencing and updating respectively a result R_i. M_i is the cost of materializing the QUEL field that gives the result R_i while S_i and U_i are the costs of writting to and reading from the cache R_i respectively. Finally, it will be assumed that invalidating an object in the cache incurs a cost IN. Given these parameters, we now describe various alternatives for the problem of selecting which results to cache. Depending on the amount of storage allocated for the cache, we differentiate between two cases: Unbounded and Bounded Space.

Unbounded Space

In this case $\mathbb{C}=\infty$ and therefore the decision to cache a result R_i, is local; that is, it depends only on the values of parameters associated with R_i. Since each object is examined individually, it will be $u_i+r_i=1$. The criterion is based on comparing the cost of processing R_i without using the cache with the corresponding cost assuming

that R_i will be cached. Let the two costs be denoted by NC_i and C_i respectively. In the case where no caching is used, the result must be produced at each reference by materializing the corresponding QUEL field. Hence the total cost will be

$$NC_i = r_i\ M_i$$

In the case where caching is used, a result is stored in the cache and is invalidated each time an update to the database has some effect on it. In order to compute the cost C_i we will differentiate between the following four cases for the types of two subsequent requests:

a) *Read-Update*: In this case the result is invalidated because of the update, the contribution to the total cost being

$$r_i\ u_i\ IN$$

b) *Read-Read*: In this case the result is simply read from the cache with total cost

$$r_i\ r_i\ U_i$$

c) *Update-Update*: The cost here is due to doing only the invalidation of the cached entry, that is

$$u_i\ u_i\ IN$$

d) *Update-Read*: This is the case where the object must be re-materialized and stored in the cache. The total cost will be

$$u_i\ r_i\ (M_i+S_i)$$

Hence for the case where the cache is used, the cost of processing will be

$$C_i = r_i\, u_i\, IN + r_i\, r_i\, U_i + u_i\, u_i\, IN + u_i\, r_i\, (M_i + S_i)$$

or, since $r_i + u_i = 1$,

$$C_i = u_i\, IN + r_i\, \left[r_i\, U_i + u_i\, (M_i + S_i)\right]$$

Comparing now C_i and NC_i we can identify the cases where it is worth caching result R_i. That happens when $NC_i > C_i$. Using the formulas extracted above, we can see that this is true if

$$\boxed{M_i > U_i + (\frac{1}{r_i^2} - 1)}$$

Checking the above condition will determine if the result of a given QUEL field materialization should be kept in the cache.

Bounded Space

This case is more realistic than the previous, in the sense that some limited space on secondary storage is allocated for caching. Hence, in this case $\mathbb{C}$ is some finite number of disk blocks. In contrast to the criterion used for Unbounded Space, *all* objects to be cached must be considered. Let N be the number of results to be cached. Each object R_i has reference and update probabilities, r_i and u_i respectively. Since many results can now be affected by the same update to a ground relation, it cannot any more be assumed that $r_i + u_i = 1$. We will however state the following property that holds in this case

$$\sum_i (r_i + u_i) = 1$$

The formulas derived above for the case of using the cache are still valid. There is an additional constraint that must be imposed here, and that has do with space limitations. This restriction indicates that the total space occupied by cached results cannot be more than $\mathbb{C}$. Given all these parameters we formulate now the problem of caching in the case of Bounded Space.

Let $A:\mathbb{N}\rightarrow\{0,1\}$ be an *allocation function*. A result R_i will be cached if $A(i)=1$; if $A(i)=0$, R_i will be discarded after it is used. Hence in the lifetime of the system, result R_i will contribute

$$BC_i = \begin{cases} C_i & \text{if } A(i)=1 \\ NC_i & \text{if } A(i)=0 \end{cases}$$

to the total processing cost. The optimal caching policy will be to cache some of the N objects so that the total cost is minimal and the space required is less than the allowed fragment on secondary storage. In other words, we seek a function A such that

$$\sum_{i=1}^{N} BC_i \qquad \text{is minimal} \tag{C1}$$

subject to the constraint

$$\sum_{i=1}^{N} A(i)\,|\,R_i\,| \;\leq\; \mathbb{C} \tag{C2}$$

This problem of optimal allocation has been shown to be NP-complete (see [CHAN77] for a similar problem). However, almost identical constraints have to be satisfied in the *view indexing* problem that Roussopoulos examined in the context of improving the performance of view based queries [ROUS82a,ROUS82b]. In [ROUS82a], he

defines a state model to formulate the above allocation problem and then gives an A* algorithm [RICH83] that finds a near-optimal allocation. We will not go here into the details of that algorithm; the reader is referred to [ROUS82a] for a rigorous and detailed presentation of the technique.

The output of the A* algorithm identifies which results are worth keeping in the cache. This allocation will be used throughout the lifetime of the system. Hence, this approach is meaningful only in the case where all QUEL fields are materialized in advance and a decision is made on which of them should be cached. However, that policy may not be the best to use. Periodically the system may re-run the same algorithm and use statistics acquired during the execution of various queries and updates. Even for objects not cached, the system may keep some statistics and recompute the allocation function A so that new results can get a chance to be stored in the cache.

In summary, the above two cases shared the fact that the reference and update probabilities for the various objects were known in advance. In the most general case, the values of the above parameters are not known and the system must be able to dynamically adapt its caching behaviour, so that the contents of the cache always reflect the most frequently used and/or costly results. We will not present here a special algorithm for the case where no statistics are available. The following subsection discusses that issue in the context of the replacement policies that can be used for the cache.

2.4.3. Replacement Algorithm

The problem of selecting a policy for replacing objects in the cache, is abstractly formulated as follows:

A *state* s of the cache is the set of objects that are stored in it $<R_1,R_2,\cdots,R_n>$ along with some statistical information associated with each R_i. We will assume here that this information is

t_i	The time since R_i was last referenced
u_i	Probability of updating R_i
M_i	Cost of producing R_i (materialization)
$\mid R_i \mid$	Size of R_i

and that the cost of writting and reading an object from the cache is equal to the size of that object. Let **S** and **R** be the set of all possible states and results to cache, respectively. Then, a *replacement policy* P, is a function P:**S**$\times$**R**$\rightarrow$**S** that, given a state s for the cache and a newly materialized result R_i, decides

a) if R_i should be cached, and
b) in case the answer to a) is positive but there is not enough free space in the cache to accommodate R_i, which other result(s) should be discarded to free the space needed.

In operating systems an optimal page buffer replacement policy is one that uses the whole (past and future) pattern of references to decide on which pages should be cached (see algorithm OPT in [MATT70]). This algorithm is not practical though, unless one can predict with high probability the future behaviour of the system. The closest approximation is the LRU (Least Recently Used) algorithm which selects to discard the object with maximum time since last reference. In the area of database

management systems, the same policy can be used in the design of buffer managers. DeWitt and Chou give in a recent article [CHOU85] an analysis of these algorithms in a database environment.

In our caching problem, an object R_i is cached independently of its parameters, as long as space can be allocated to store R_i in the cache. If this is not the case, then some result(s) must be discarded to free the space needed for storing R_i. There are generally two approaches one can take

a) We can first try to approximate the parameters of Table 2.1 using the statistics the system has acquired. The sizes $|R_i|$ and the materialization costs M_i are given since the objects have been computed already. The update probability u_i is also easy to derive, assuming that the probabilities of updating ground relations are given. For example, the probability of updating the result of a join between two ground relations is equal to the sum of the probabilities of updating each of the two relations. What remains to be provided is the probability of referencing a result as well as the probability of updating the result, in the case where the frequencies with which ground relations are updated are not known. For objects already in the cache, these probabilities can be estimated from the reference patterns already observed. For new results, one can predict the reference pattern if the query processing algorithm is known. For example, in the case of processing a join, if it is known that either nested loops or merge scan will be used, we can predict the way QUEL fields are accessed, and therefore have a rough estimate for the needed probabilities.

Once these values are known, the A* algorithm of the previous subsection can be run and give a new allocation for the cache. This will provide the system with a good cache allocation for a limited time interval. Clearly, because the A* algorithm is very expensive to run, one would not like to decide on a new allocation each time a new object is materialized. The solution we propose is to run the algorithm once some threshold is reached. Such a threshold may be a fixed number of materializations. Another threshold may be the difference between the values for the statistics used to run the allocation algorithm (i.e. reference and update probabilities, sizes of results, etc.) and the actual values observed while the system is running. For instance, if that difference gets above some prespecified percentage of the original estimates the system may decide to re-run the A* algorithm.

b) A different approach is to consider the values of given parameters only and try to approximate the optimal policy with an LRU-like policy. If, for example, we assume that the materialization cost, the size and the probabilities of referencing or updating an object are uniformly distributed over all objects, then LRU will be enough to guarantee a good caching behaviour. The point is that by making the above assumption the original problem has been reduced to the known page buffering problem in operating systems. However, in the general case LRU will not work. In that case, we propose the derivation of some experimental formula $rank(M_i, u_i, t_i, |R_i|)$ which would rank objects according to the values of their associated parameters, given some weights and scaling factors. The lowest ranked object(s) should be discarded at a point where space is needed. Examples

of *rank* are

(1) $rank(M_i,u_i,t_i,|R_i|) = M_i$

The assumption made here is that objects are very expensive to materialize and the rest of the parameters are uniformly distributed. Therefore, objects with low M_i values should be discarded to free space for objects with high M_i values.

(2) $rank(M_i,u_i,t_i,|R_i|) = \frac{1}{t_i}$

In this case objects are expected to be frequently referenced and very rarely updated. Then a pure LRU algorithm based on the times since last reference is a good choice.

(3) $rank(M_i,u_i,t_i,|R_i|) = \frac{1}{u_i}$

If some materialized results are very frequently updated, it may not be worth caching them or, for the purposes of a replacement policy, should be discarded to allow other less frequently updated objects be cached.

(4) $rank(M_i,u_i,t_i,|R_i|) = |R_i|$

Small objects should be discarded in case larger ones need be cached.

Trying to generalize *rank* by combining all four functions we suggest the following function for *rank*

$$rank(M_i,u_i,t_i,|R_i|) = \frac{1}{u_i}\cdot(w_1M_i + w_2|R_i|) + w_3\cdot\frac{1}{t_i}|R_i|$$

This formula is the simplest one that can be devised and incorporates in an easy way the effects of the various parameters. The specific format was chosen to agree with the formulas derived during the analysis of section 2.4.2. The first factor is based on

the fact that updates require materialization of objects as well as storing the results in the cache. The second part simply introduces the LRU-like behaviour. How to derive the weights w_1, w_2 and w_3 is an interesting open problem and should be attacked through extensive experimentation.

2.4.4. Checking the Validity of Cached Objects

Cached results of materialized QUEL field entries may become invalid when the relations used to compute these results are modified. Checking the validity of the cached objects amounts to identifying which results are affected from a given update. When such a result R_i is found to be affected, one of two actions can take place

a) One can simply invalidate the corresponding entry of the cache. The next query that tries to use the result, will find it invalidated and will have to re-evaluate the associated query. This is the scheme assumed in the analysis of the previous subsection.

b) One can use the updates performed to the underlying relations and *propagate* them to all cached entries affected by these updates. In this case, some algorithm must be used which, given an update and the query that was used to derive of a specific result, will provide a set of update operations that will bring the cached result up to date. Such algorithms are described in various articles where the same problem is attacked in different contexts [BUNE79,ADIB80,KUNG84,BLAK86].

In our environment however, the second approach suffers from two very serious drawbacks. First, it is the case that between two references to a specific cached result many updates to underlying relations may be performed. Clearly, for each of these updates significant effort will be spent doing propagation of the updates. Another possibility is to log all updates and propagate them at the time a retrieval is performed (batch update) [ROUS86]. The second drawback is due to the fact that updates may be propagated to bring up to date entries that *may never* be used in the future. From the above discussion, it is clear that a good caching scheme will discard these results and replace them with others more frequently used which makes any effort to propagate updates useless.

We take the approach that entries must be brought up to date *on demand*, that is, the next time the specific entry is requested in a query. Then the system can either *incrementally* propagate the modifications, assuming that we keep the updates in some kind of a log [ROUS86], or simply re-evaluate the query. That is an optimization question and depends on the specific characteristics of the query and the updates. We will not attempt here to discuss in more detail these algorithms.

The rest of this subsection discuss briefly the problem of detecting which cached results are affected by a given set of updates. [STON86a] presents a detailed discussion of the problem and the proposed solutions. The two approaches taken there, *Basic Locking* and *Predicate Indexing*, share the same properties with physical and predicate locking respectively [GRAY78,ESWA76] as used in concurrency control. Abstractly, a set of tuples is used to produce the result of some query and our goal is to be able to detect when a given update *conflicts* with this set. Hence, the similarity

with the concurrency control problem.

In Basic Locking all tuples used in processing a given query are marked with a special kind of marker which contains the identifier `Qid` of the query. If an index is used for accessing the data tuples these markers are set on data records *and* on the key interval inspected in the index. Index interval locks are required to correctly deal with insertion of new records (the *phantom problem* in concurrency control [ESWA76]). If a new tuple is inserted in one of the relations used to produce the result of a QUEL field entry, then the collection of markers must be found for the new tuple. To ascertain what collection of cached entries are affected by the insertion of a tuple t, one first collects all the markers on t and then determines which of the corresponding queries are really affected.

In Predicate Indexing the cache has a specific organization. A data structure is built allowing efficient search of the cache and detection of entries affected by the insertion of a specific tuple in one of the underlying relations. In [STON86a], a special kind of R-tree [GUTT84a] is used for that reason. Using Predicate Indexing implies no special treatment of insertions to ground relations but a search of the whole R-tree is required whenever one asks for the cached entries affected by an update.

Performance analysis results in [STON86a], show that it is not possible to choose one implementation to support efficiently any cache based environment. Depending on the probability of updating ground relations and the number of cached entries that overlap (in the sense that their read sets share some tuples from ground relations), the first or the second approach becomes more efficient. Basic Locking seems the most

promising because of its ease of implementation, performance in simple environments, and extensibility to join predicates. Analysis of these schemes and investigation of other extensions are a topic of future research.

2.4.5. Indexing the Cache

As a final issue in the caching problem we touch briefly the problem of indexing the cache. By "indexing" we mean an efficient way to detect if for a given query Q there is a cached result R that is the answer to Q. The problem therefore is to search for `Query_expression` values in the cache which are *identical* to Q, up to renaming of tuple variables used. In other words, the expressions are identical once we substitute the tuple variables with the names of relations they range over. Checking for identical queries is rather straightforward. It involves transforming the query to be checked into the canonical form that we assumed in subsection 2.4.1 and then a simple syntactic matching. But, clearly one does not want to compare all entries of the cache with Q. It is desirable to quickly reject all of the entries that do not relate at all to the given query. We therefore associate with each entry a *signature* that contains high level and easy to check information about the query. The relations involved and the fields that appear in the qualification and the target list of the queries are used to build the signature. If the signatures of a cached entry and the given query match we can then continue with a more detailed checking, the syntactic comparison of the two canonical representations. To have quick comparison of the signatures themselves, a hash table where hashed representations of the signatures are stored can be used.

This last subsection concludes our presentation on caching results of QUEL fields. A working version of extended INGRES has a very simplified cache which performed very well in the experiments of [STON85]. POSTGRES [STON86b] will be supported by a more sophisticated caching scheme which will use LRU for replacement and Basic Locking for checking the validity of the entries.

2.5. Indexing Results of QUEL Fields

Imagine a query that is frequently asked and has the following form

```
retrieve (EMP.name) where EMP.hobbies.average < constant
```

One would most probably like to build an index on `EMP.hobbies.average` in the same way indexes are built on simple attributes. However, there is a difficulty in using conventional indexing schemes to index results of QUEL fields. This would require the materialization of *all* entries in the QUEL field and, moreover, materialization must be done when a new tuple with a QUEL field is inserted. For example, if a new employee tuple is inserted in the EMP relation the *hobbies* field must be processed, the result cached if possible and the index on `EMP.hobbies.average` must be updated with the new values. This indexing scheme suffers from two serious drawbacks. First, insertion time increases significantly since it is no longer a simple addition of a tuple in a relation, but the execution of (possibly) many queries as well, the ones stored in QUEL fields. In particular, in the case of queries involving clauses with multi-dot expressions, response time may increase drastically. Second, by precomputing QUEL field entries the system materializes *all* objects and therefore spends a lot of time (and possibly space in the cache) in processing field entries that may be never referenced in

the future.

Another proposal that overcomes the above problems is presented here. The main idea is to have the index reflect only values that have been seen in the past and not all possible ones. Through this scheme, it is expected to achieve better performance in cases where the same set of queries is frequently asked. We are also willing to pay some penalty to update the index in the case where the set of queries changes. Given a field, the structure to be described, contains information on all values of that field that appear solely in results of materialized entries. These results do not have to exist in the cache; they can exist in the index even if the object that included them has been flushed out of the cache. In these cases, the index simply shows that some QUEL fields, even if not currently materialized, can produce the specific values stored. Moreover, some extra information is associated with the index; information that characterizes the class of tuples that are indexed. In summary, the indexing scheme proposed is a *partial index* in the sense that it indexes only a part of the relation.

Let us use an example to motivate the discussion on partial indexes that follows. The relation EMP (name,salary,mgr,hobbies) of section 2.2 has an index defined on EMP.hobbies.average. The following tuples are currently in EMP

name	salary	mgr	hobbies
Riggs	20	Smith	retrieve (SOFTBALL.position, SOFTBALL.average) where SOFTBALL.name = "Riggs"
Jones	30	Smith	catcher 4 pitcher 8
Felps	40	Moore	catcher 5 pitcher 4
..	....	....	.....

Assume also that there is a unique tuple identifier *TID* associated with each tuple in the EMP relation, with value 100,101 and 102 for the first, second and third tuple respectively. These values are stored in the EMP relation but are not visible to the user. The results of the second and third tuples have been materialized and stored in the cache. That is indicated in the above relation by representing them with small relations stored in the *hobbies* field of EMP. Suppose the query that has caused that materialization was

```
retrieve (EMP.name)
    where EMP.salary > 20
    and   EMP.hobbies.average < 6
```

and was processed by scanning EMP and materializing only the *hobbies* fields of employees with salary more than 20K. The index on EMP.hobbies.average was of

no use because no entries were materialized before the above query was executed. However, after the execution of the query the index was updated to

salary > 20	
average	TID
4	101
4	102
5	102
8	101
.....	.....

Notice that the above index differs in two ways from conventional indexes. First, there may be more than one *average* values for the same *TID* value. This cannot be true in conventional relations because all fields carry a single value (First Normal Form [ULLM82]). Second, there is a predicate associated with the index (`salary > 20`). This predicate uses *only* non-QUEL fields and is a simple way to identify the kind of tuples indexed by the given index. That predicate is also used to decide if an index is useful in answering a given query. For example, a future query that includes a restriction on `EMP.hobbies.average` and references employees with salaries more than xK, with $x > 20$, can use the index to avoid a full scan of `EMP`. However, for $x \leq 20$ the relation must be scanned and the entries with salary values under 20 will be materialized. As a side effect, the index table and the corresponding predicate will be updated.

Let us now describe the operation of a partial index. A partial index is a pair (*QUAL*,*INDX*), where *QUAL* is a disjunction of conjunctive one-variable selection clauses and *INDX* is a conventional index structure. We will say that a qualification $QUAL_1$ *covers* another qualification $QUAL_2$ if the set of tuples satisfying $QUAL_2$ is a subset of the set of tuples satisfying $QUAL_1$, for *any* instance of the database. In any other case we will say that $QUAL_1$ is *not useful* to $QUAL_2$. When an index is requested by a user on a field F of a QUEL field result, a pair (*QUAL*,*INDX*) is allocated with initial values *QUAL*=*false* and *INDX*=$\emptyset$. Then depending on the operation performed on the relation, the following actions will take place.

Queries that use F in an one-variable clause in the qualification:

Let $QUAL^-$ be the part of the qualification of the query that has no references to QUEL fields and is composed only from one-variable clauses on the relation that the index is built on. Then, if the predicate *QUAL* which is associated with the index covers $QUAL^-$, the query processor may consider using the available index on F for answering the query. If *QUAL* is not useful to $QUAL^-$, then the query cannot use the index on F. That index can be used to give only the tuples satisfying *QUAL* while the rest of the requested tuples must be retrieved from the relation by other means. However, in that case, once the QUEL field entries are materialized, the values of F are used to update the index and the associated qualification *QUAL* is changed to $(QUAL \vee QUAL^-)$.

Queries that do not use F in an one-variable clause in the qualification but materialize the QUEL field that contains F:

In this case we take the steps followed in the second case above where the index is updated after the materializations are performed.

Insertion of a new tuple in the indexed relation:

Given a new tuple to be inserted in the indexed relation, we check if this tuple satisfies *QUAL* and if so, the corresponding QUEL field is materialized and the index is updated. Otherwise, the index remains as is. In the former case, we may materialize entries that show no indication if they will be used in the future. Although this was one of our arguments against pre-materialization of all entries in the beginning of the section, there seems to be no easy way to get around that problem. If the predicate *QUAL* is satisfied it is required that the F value of the new tuple be in the index. Another approach, would be to change *QUAL* to $(QUAL \wedge (not\ QUAL^{-})$, where $QUAL^{-}$ is a qualification that describes the tuple inserted and can be built according to the above discussion. This way we avoid inserting the new values in the index. Although this solution is conceptually correct, it is very hard to check whether *QUAL* covers other predicates if negation is allowed [ROSE80].

Deletion of a tuple from the indexed relation

In case of deleting a tuple with tuple identifier *TID*, the entries of the index that contain the same *TID* value are also deleted. The predicate *QUAL* can then be changed to reflect the fact that the specific value is not any more represented in the index by introducing negative clauses in *QUAL*. Because of the above mentioned efficiency problems, we propose to leave the qualification part unchanged

and simply allow incoming new tuples to be inserted in the index even in the case they match deleted values. Hence, the predicate *QUAL* is only "increasing" by means of the number of tuples it covers.

Updates to ground relations may also affect the contents of a partial index. In the case where these updates are affecting results of QUEL fields, changes may have to occur in the index as well. Using a validation scheme similar to the one of the previous section, we can check which index entries must be changed after a given update to a ground relation.

The above are the only actions required to keep an index up to date. Clearly, the content of the index reflects the dynamics of the system by providing information only on data frequently asked. In that sense, partial indexing is also some kind of *session support* [KUCK86], where a user starts up a session and depending on the queries he/she uses, the system may create secondary structures to speed up common operations. Another comment is that the predicate *QUAL* associated with the index, may at some point get extremely complicated because of the number of disjuncts it may contain. At such a point the system may use some statistics to estimate the percentage of the tuples that have already been indexed. If that is above a predefined threshold (e.g. 80%), the system may select to index *all* QUEL entries. *QUAL* is then changed to *"true"* and all incoming new tuples will have to be indexed. We then arrive in the situation that was discussed in the beginning of the section where all materialized objects are guaranteed to have an entry in the index table.

Finally, we would like to mention another possible use of partial indexes. Many times users issue all their queries through specific views that they have defined over ground relations. Users are not allowed to keep materialized versions of the views in the system because of its high space cost, but they still would like queries to execute fast. Indexes on ground relations will be helpful for that. However, these indexes contain more information than what these users need, namely an index *only on the result of the view materialization*. A partial index seems like a clean solution to that problem. The *QUAL* part will be static since it will be the predicate that defines the view, but querying and updating will be performed under the guidelines outlined above. This idea can also be extended to normal relations, since these are special cases of views. Using partial indexes better performance can be achieved by allowing the index to keep information only on frequently accessed data.

2.6. Summary

This chapter first presented the language QUEL+ and its capabilities. Then, an extended decomposition algorithm based on the INGRES query processing algorithm was proposed. The extensions made were mainly due to the fact that one new operation was introduced, namely the materialization of QUEL fields. We showed how a general algorithm can be used to take under account the fact that materialization is very expensive and the number of times it is performed should be minimized. Also, some special case strategies were discussed that aim to reducing the sizes of materialized results.

Caching was then proposed as a way to avoid evaluating the queries found in QUEL fields more than once. Several issues associated with caching were discussed. Among others, replacement policies, invalidation algorithms and policies that decide which objects to cache were examined in detail. The discussion shows that caching is essential in the QUEL+ environment and various solutions to the above problems can be derived once the cached object characteristics are known. How to compute these characteristics and how to adapt the system caching policies according to these statistics is a very interesting open problem.

Lastly, a new indexing technique, Partial Indexing, was proposed to provide efficient access to results of QUEL field materializations. A partial index is a combination of both a conventional index table and a predicate. Predicates characterize the set of tuples that can be accessed through the corresponding index tables. We also described how the system can check if an index is useful in processing a given query and what are the necessary operations to maintain a partial index when queries and updates are performed.

CHAPTER 3

OPTIMIZING THE EXECUTION OF PROCEDURES

3.1. Introduction

The previous chapter introduced the language QUEL+ and suggested some ways to speed up query processing in the case where the commands stored in QUEL fields are exclusively queries (or `retrieve` commands in INGRES). This chapter is concerned with the more general problem of procedure optimization in the QUEL+ environment. To motivate the discussion that follows, we give an example drawn from [KUNG84].

Suppose that we are given a set of algorithms that can be used to solve the Shortest Path (SP) problem on a grid representation of a map. These algorithms find a sequence of points in the grid starting from a given point *S (source)* and ending to another point *D (destination)* such that the total cost of traveling through these points is minimal. This set of algorithms will be represented through the use of a relation

```
ALGORITHMS (alg_id, alg_type, code)
```

where *alg_id* is a unique identifier, *alg_type* indicates the general class that the given algorithm belongs to (e.g. Dynamic Programming [LARS78], Branch and Bound [RICH83], etc.) and *code* is a field of type QUEL that is used to store the actual set of database commands (procedure) that implement the algorithm. Therefore the form of

the relation ALGORITHMS will be

alg_id	alg_type	code
10	Dynamic Progr.	code line 1 code line 2 .....
15	Dynamic Progr.	code line 1 .....
20	Branch and Bound	code line 1 .....
..	.....	.....

To give an example of an entry in the *code* column of the above relation, we will present a database procedure that solves the Shortest Path problem using an algorithm based on Dynamic Programming. Assume the existence of a relation FEASIBLE (source,dest,cost) that provides the cost of getting from a node *source* to a neighbor node *dest*. Another relation STATES (dest,cost,open) is also used to record the cost of getting from the initial source point S to any already visited point *dest* in the map. The third field *open* indicates if the corresponding *dest* node has been visited in the past. If *open*=*0*, the algorithm will avoid visiting that node again. Based on these relations, the following is a database procedure that finds the shortest path between two points S and D of the map.

```
retrieve into STATES (dest = S, cost = 0, open = 1)

range of s,t is STATES
range of f is FEASIBLE
```

```
execute*
{
   append to STATES (dest = f.dest,
                     cost = f.cost+s.cost,
                     open = 2)
      where s.dest = f.source and s.open = 1

   delete s
      where s.dest = t.dest and s.cost > t.cost
      or    s.cost > t.cost and t.dest = D

   replace s (open = s.open - 1)
      where s.open ≥ 1
}
```

The details of the above algorithm and its particular implementation are further discussed in [KUNG84]. Suppose that the above is stored as the algorithm with unique identifier 15. As mentioned in the previous chapter, a user can request the execution of this specific algorithm, using the QUEL+ command

```
execute (ALGORITHMS.code)
   where ALGORITHMS.alg_type = "Dynamic Progr."
   and   ALGORITHMS.alg_id = 15
```

How to pass parameters and other issues that deal with the details of fully supporting database procedures will not be explored here. In [STON85] and [STON86b], Stonebraker *et al.* give an extensive analysis of these problems and suggest solutions.

Our focus here will be the problem of efficiently processing these QUEL fields. The system may consult the given set of commands and process them in a way that minimizes the total execution cost. Relational DBMSs were made efficient largely through the use of sophisticated optimization algorithms ([WONG76,SELI79]). This chapter suggests extensions to these optimization algorithms for the new extended query language QUEL+. Although QUEL+ is used as an example, the proposed

principles should be applicable to a wide variety of extended languages.

Given a set of database commands, it is a common practice in conventional DBMSs to optimize each command separately. To "optimize" a command means to choose among the various ways of executing the command. For example, there may be a choice of indexes to use, or a choice of strategies for executing a relational operator such as the join. We extend these ideas here for the case of multiple command processing by discussing interquery optimization techniques.

This chapter is a more detailed presentation of the ideas presented in [SELL85] and is organized as follows: In the next section we define the notion of an optimization unit. Then, in sections 3.3 and 3.4 various optimization tactics for use by a QUEL+ optimizer are described. Each of these tactics is related to corresponding techniques from some other area, in particular compiler construction and query optimization. Section 3.5 presents two new transformations, each of which transforms a sequence of QUEL+ commands into a single `replace` command. Finally, concluding the presentation of this chapter, section 3.6 summarizes the ideas discussed.

3.2. What is Optimization?

Optimization in database systems means to choose among the various ways of executing a command. In this section we will examine what optimization will mean for extended languages like QUEL+. We motivate our definition of optimization by reviewing some QUEL+ constructs.

The `execute` command, as presented in the previous chapter, gives recursive power to QUEL by allowing the system to execute relation fields. It is very useful in

its `execute*` form, where the given sequence of QUEL+ commands is executed repeatedly, until the database does not change. Generally, each new command of QUEL+ represents a sequence of one or more simple QUEL commands. This is also true in the other extended database languages mentioned in the introductory chapter. For example, in Guttman's thesis [GUTT84b], the new construct is the repetitive execution operator (*) of QUEL+. Also in GEM [ZANI83], processing of a multi-dot query has been implemented by translating it to QUEL queries [TSUR84]. Since a command in an extended language typically represents several commands in a classical database language, this section proposes that a QUEL+ query optimizer operates on a sequence of commands rather than the traditional approach of optimizing a single command at a time.

As a first attempt at designing a QUEL+ optimizer, one could merely optimize each corresponding QUEL command separately, using an existing QUEL optimizer. For example, a `replace*` command would be processed by generating one `replace` command, optimizing and executing it, and continuing until the execution of the `replace` command does not change the database. We use the term *optimization unit* to refer to the unit acted on by the optimizer. Thus in QUEL the optimization unit is a single QUEL command. We propose that for QUEL+ the optimization unit will be a single QUEL+ command, including even an `execute` or `execute*` operation. Therefore, the optimization unit has been effectively made equal to any sequence of QUEL+ commands, for any such sequence can be the argument of an `execute` command. In fact, if the programmer wishes, he/she can code an entire QUEL+ program (containing no programming language commands) inside a single `execute` statement

and the optimization unit will then be that entire program. There are at least two advantages to enlarging the optimization unit:

(a) The optimizer has more information on which to base its decision. For example, knowing that there will be several consecutive `replace` commands executed, the optimizer may elect to build an index which is not worthwhile for only one `replace`.

(b) The optimizer has more flexibility to rearrange the order and implementation of operations. For example, in an `execute*` which includes a `delete` command, it will be useful to perform the `delete` operations as early as possible, in order to reduce the size of the relation to be processed.

One possible disadvantage to this approach is the following:

As the size of the optimization unit grows, so does the complexity of the optimization task. The first comprehensive approach to query optimization [WONG76] proposed query decomposition as a method to avoid searching the exponentially growing space of query processing strategies. However, the most successful query optimization method has been that of System R [SELI79], which does perform essentially an exhaustive search of the strategy space. Even System R's strategy avoids searching the full strategy space by using some heuristics to prune down the cost of the decision process [SELI79]. Therefore by allowing the optimization unit to grow arbitrarily, the cost of searching the strategy space may exceed the savings in efficiency.

The benefit of these advantages, and the cost of the disadvantages, grows with the size of the optimization unit. Clearly this size is to a significant extent under the control of the programmer, who can enlarge it by placing several QUEL+ commands inside an `execute` command. Notice also that an optimization unit must be smaller than or equal to a transaction unit. This is because the optimizer may completely rearrange the order of execution of commands in an optimization unit. If there was an *end-transaction* statement inside the optimization unit, it would have a completely different meaning after a rearrangement.

Following the above discussion, the remaining sections examine various techniques that can be used by a QUEL+ optimizer. These are general database program transformations that aim at reducing the execution cost. The classes of optimization tactics presented in the following sections are each closely related to techniques used in other contexts, namely compiler design and query optimization.

3.3. Compiler Design Techniques

Optimization techniques in compiler design focus especially on two areas [AHO79]

- temporary storage management (space), and

- loop optimization (time)

Suppose a sequence of operations on an employee relation is given; in addition all qualifications restrict the initial relation to the set of tuples of employees working for Joe. Then it may be more efficient to create a temporary relation in advance that will contain only the tuples of those employees. We view this problem as the problem of *temporary storage management*. Managing temporary storage in the context of

database operations means optimizing the execution of commands by reusing results (e.g. temporary relations) produced during the execution of other commands. The major difficulty in solving this problem is caused by the presence of update commands. Updates prevent the optimizer from making significant predictions on the kind of data that is accessed during the execution of the commands. The most interesting and tractable case arises when all commands are retrievals from the database. It is then possible to even rearrange the order in which retrievals are performed so that accessing the same data pages is avoided. Algorithms for this special case are given in Chapter 4.

The focus of this section will be the second problem, i.e. loop optimization. In database operations loops are found in two levels, single queries and transitive closure (*) operations. In the former, repetitive execution is inherent in the commands. For example, a query involving a join between two relations can be implemented with a nested loop. On the other hand * operations are explicitly user defined loops.

The case of implicit loops has been studied in the past as the problem of finding query execution plans that minimize execution time and avoid evaluating the same expressions many times [BLAS76,EPST79]. This corresponds to identifying loop invariants in compiler design.

In the context of * operations some new problems arise. The following two subsections examine two of them.

3.3.1. Loop Invariants

An interesting version of the same problem discussed above is the problem of identifying loop invariants within a set of commands. For example, aggregate computations that involve relations not updated during the execution of an iteration can be evaluated outside the loop and be replaced with a constant in the body of the loop. The following command changes the salaries of employees with age more than the average employee age and satisfying some other condition `Qual`.

```
replace* EMP (salary=newsalary)
    where EMP.age > avg(EMP.age)
    and   Qual
```

Clearly the average employee age computation can be moved out of the qualification and be performed only once, turning the single `replace*` command into the following two, presumably more efficient, commands (the first one is assumed to be some system operation)

```
Set AVG = avg (EMP.age)

replace* EMP (salary=newsalary)
    where EMP.age > AVG
    and   Qual
```

The above transformation resembles the previous case of intraquery optimization described in [EPST79] and the gain in execution time is substantial, especially in cases where the result of the aggregate is involved in join clauses. An algorithm that transforms the first database procedure to the second one can be very easily derived; for each aggregate in the loop, it checks if all relations or relation fields involved are not updated during an iteration step. If this is the case then the aggregate can be

computed outside the loop and stored in a variable which then replaces every occurrence of the aggregate in the body of the procedure.

In general, there will be iterative database programs where invariants are not easily identified. Then another technique may prove useful.

3.3.2. Incremental Computation

A more careful treatment of aggregates in loops is also possible if after doing the simple transformation suggested above there are still aggregates to be calculated in each iteration. In that case, it may be worth incrementally computing those aggregates, i.e. computing them once in advance and then updating them every time the data involved changes. As an example, suppose a relation PARTS (pid,type,supplier,quantity) with the obvious meaning is given. the command

```
replace* PARTS (quantity=PARTS.quantity+2)
   where PARTS.type = "pipes"
   and   PARTS.supplier="Smith"
   and   PARTS.quantity ≤ avg(PARTS.quantity where PARTS.type="pipes")
```

modifies the PARTS relation according to the following semantics

> ***if the quantity of some type of pipes that Smith supplies is less than the average quantity of pipes supplied, repetitively increase his supply for that type of pipes by 2***

We can then define a "variable" AVG to hold the result of the above aggregate which computes the average quantity of pipes supplied by the various suppliers. Let also COUNT be another variable that holds the number of tuples satisfying the qualification of the aggregate, that is COUNT = | PARTS [type=pipes] |. Then, given a change

that modifies k tuples, the new value for AVG can be computed using the formula

$$\text{AVG} \leftarrow \text{AVG} + \frac{2k}{\text{COUNT}}$$

The number of tuples modified is always available at the end of performing an update. For example, in INGRES because deferred update is used for crash recovery reasons, the size of the intermediate relation created to hold the new tuples, is used to compute COUNT. The new database procedure that is equivalent to the one presented initially will be

```
/* Initialize the variables used */
Set AVG = avg(PARTS.quantity where PARTS.type="pipes")
COUNT = | PARTS [type=pipes] |

/* Then repeatedly process the PARTS relation */
execute*
   {
    replace PARTS (quantity=PARTS.quantity+2)
       where   PARTS.type = "pipes"
       and     PARTS.supplier="Smith"
       and     PARTS.quantity ≤ AVG

    AVG ← AVG + 2k/COUNT
   }
```

Similar formulas can be derived for all common aggregates, like MIN, MAX, SUM and COUNT. This technique will usually result in a more efficient implementation, if the number of modified tuples is small. Finding the formulas that compute the new value of an aggregate computation, given the previous value of the aggregate and the new values for the tuples, might be hard depending on the structure of the command. However, even if more effort is needed to construct the equivalent database pro-

cedures, it may pay off at execution time, especially if most of these procedures are "canned" transactions that are processed frequently. In this case optimization at *compile time* is possible; that is, preprocessing of the database command and derivation of a semantically equivalent but more efficient procedure is possible at the point where the transaction is coded rather than the time it is executed.

As a final comment it should be mentioned that the incremental computation of aggregates is in another sense a way to update cached results. The ideas discussed in the previous chapter can be therefore used to detect when an aggregate computation is affected by specific database updates. Similarly, one might also like to pull the whole subexpression

```
where PARTS.type = "pipes"
and   PARTS.name = "Smith"
```

out of the loop by building a temporary relation to hold only that data. This is part of the temporary management problem that was mentioned in the beginning of the section and is treated in more detail in Chapter 4.

3.4. Query Optimization Techniques

In this section we examine some ideas from query optimization that are useful in optimizing extended query language constructs. They are categorized as

- early restrictions, and

- combining operations.

3.4.1. Early Restrictions

It is usually advantageous to restrict the size of the relations involved in a query as early as possible in the execution plan. For example, INGRES selects to execute all one variable selection clauses in the first step of processing a query. In QUEL+, `delete` commands can be considered as restrictions since they reduce the size of relations involved in subsequent commands. Therefore, in a way analogous to one-variable clause detachment [WONG76], one may want to incorporate the effects of `delete` commands as early as possible. Unfortunately `delete`'s cannot simply moved earlier in the sequence without affecting the semantics of the procedure. For example, in the case of an `append` followed by a `delete`, if the second command is processed first it may remove from the updated relation tuples that are used in the `append` command. The absence of these tuples will clearly affect the result. Only in the very simple cases where the read set of the `append` command is non-overlapping with the read set of the `delete` it is possible to reverse their order. However, a safe modification would be to introduce the effects of a `delete` command earlier in the sequence by enchancing the qualifications of preceding commands. For example, the following sequence of operations on the relation `EMP (name,salary,mgr)`

```
range of EMP,EMP1 is EMP

/* make Joe the manager of all employees */
append to EMP (name=EMP.name,
               salary=EMP.salary,
               mgr="Joe")

/* but ... nobody can make more than his/her manager */
delete EMP
    where EMP.salary > EMP1.salary
    and   EMP1.name = EMP.mgr
```

can be changed to

```
/* append only tuples of employees that make less than Joe */
append to EMP (name=emp.name,
               salary=emp.salary,
               mgr="Joe")
    where emp.salary ≤ emp1.salary
    and   emp1.name = "Joe"

/* but ... still have to delete some old tuples */
delete EMP
    where EMP.salary > EMP1.salary
    and   EMP1.name = EMP.mgr
```

The savings introduced are due to the fact that new employee tuples inserted and then immediately deleted by the `delete` statement, are simply not inserted at all. An interesting case arises in situations where iterations of the same sequence are processed. For example, assume that the above two commands are executed every time a new super-manager is declared. Then, it is true that before executing the above commands no employee was making more than his/her manager which makes the `delete` statement unnecessary. Hence, the single append command

```
append to EMP (name=emp.name,
               salary=emp.salary,
               mgr="Joe")
    where emp.salary ≤ emp1.salary
    and   emp1.name = "Joe"
```

is equivalent to the initial sequence. Clearly, that one-command program is more efficient than the original `append-delete` pair. However, in general, the `delete` command must be added at the end to make sure that the old employee-manager pairs satisfy the restriction on their salaries. In that latter case, the savings achieved are not obvious. The size of the relation on which the `delete` command will act upon is

smaller (since the append statement has more restrictive qualification than the original one) which will make that command execute faster. In contrast, the append operation is more expensive to process because of the more complex qualification.

Formally, the above transformation can be expressed as follows. Given the sequence

range of $t,t_1,t_2,...,t_n$ is T
range of $s_1,...,s_m,r_1,...,r_l$ are other relations

append t ($f_1=F_1(t,t_1,...,t_n,s_1,...,s_m)$,
$f_2=F_2(t,t_1,...,t_n,s_1,...,s_m)$,
.... ,
$f_k=F_k(t,t_1,...,t_n,s_1,...,s_m)$)
where $QUAL1(t,t_1,\ldots,t_n,s_1,\ldots,s_m)$

delete t
where $QUAL2(t,t_1,\ldots,t_n,r_1,\ldots,r_l)$

it can be transformed to

append t ($f_1=F_1(t,t_1,...,t_n,s_1,...,s_m)$,
$f_2=F_2(t,t_1,...,t_n,s_1,...,s_m)$,
.... ,
$f_k=F_k(t,t_1,...,t_n,s_1,...,s_m)$)
where $QUAL1(t,...)$ and $\overline{QUAL2}(t',t'_1,...,t'_n,...)$

delete t
where $QUAL2(t,...)$

where

t'_j = the tuple t_j where its fields f_i ($1 \leq i \leq k$) are changed to $F_i(t,...)$

and

$\overline{QUAL2}$ is the negation of $QUAL2$

The hard part of the above transformation is the computation of $\overline{QUAL2}$. In the case

where $QUAL2$ is simply a conjunction of one-variable clauses, $\overline{QUAL2}$ is the disjunction of the same one-variable clauses with reversed operators. However, when joins are allowed, the task of producing $\overline{QUAL2}$ becomes very hard. Aggregate functions must then be used since the query language semantics cannot support negation in the same sense with first order logic. Stonebraker's proposal for the implementation of integrity constraints using query modification [STON75] can be used to construct $\overline{QUAL2}$. Finally, a similar transformation can be derived for `replace-delete` pairs of commands.

Generally, performing this syntactic rather than semantic transformation, does not require any specific knowledge about the pair of commands. On the other hand, the gain in performance is high, especially in cases where the number of tuples appended and immediately deleted by the next command, is large. Also, as mentioned above, sometimes the second operation need not be performed at all, i.e. its effects are totally introduced into the first command. This requires extra information that can be either derived from the form of the program (e.g. the example mentioned above about iterative programs) or from the use

3.4.2. Combining Operations

In conventional query optimization one might prefer to execute both a selection and a join in a relation at the same time, thus avoiding scanning the same tuples twice. In the extended environment of QUEL+ one might like, analogously, to combine the execution of multiple commands. In the case of `retrieve` only commands, merging is possible and practical in many cases (see the discussion of Chapter 4).

Alternatively, consider a sequence of two `replace` commands. We will show that there is a single `replace` operation that produces the same result. This new command is the composition of the two original `replace` commands, where composition in the context of database operations is defined in the same way as with mathematical functions. An update command like

`replace` T `(Target-list) where` $Q(T,R_1,R_2,...,R_n)$

can be thought as the operation

$$t \leftarrow \mathbf{f}\,(t,R_1,R_2,...,R_n)$$

on a tuple variable t ranging over T, where

$$\mathbf{f}\,(t,R_1,...,R_n) = \begin{cases} \mathbf{h}\ (t,R_1,...,R_n) & \text{if } Q = \textbf{true} \\ t & \text{if } Q = \textbf{false} \end{cases}$$

Here, **h** is a function that describes how values are assigned to fields of the relation T according to "Target-list", and R_1, R_2,...,R_n are relations not affected by the `replace` command.

The transformation we propose is as follows: given a relation $T\,(f_1,f_2,...,f_k)$ and the following two `replace` commands

`range of` $t,t_1,t_2,...,t_n$ `is` T
`range of` $s_1,...,s_m,r_1,...,r_l$ `are other relations`

(I) `replace` $t\,(f_1{=}F_1(t,t_1,...,t_n,s_1,...,s_m),$
$f_2{=}F_2(t,t_1,...,t_n,s_1,...,s_m),$
$\ldots\ldots\ \ldots\ldots\ ,$
$f_k{=}F_k(t,t_1,...,t_n,s_1,...,s_m))$
`where` $QUAL1(t,t_1,...,t_n,s_1,....,s_m)$

(II) `replace` $t\ (f_1{=}G_1(t,t_1,...,t_n,r_1,...,r_l),$
$f_2{=}G_2(t,t_1,...,t_n,r_1,...,r_l),$

$$\ldots\ldots \quad \ldots\ldots \quad ,$$
$$f_k = G_k(t, t_1, \ldots, t_n, r_1, \ldots, r_l))$$
where $QUAL2(t, t_1, \ldots, t_n, r_1, \ldots, r_l)$

transform it to the following replace command

replace t $(f_1 = f'_1 + (f''_1 - f'_1) * d_2.value,$
$f_2 = f'_2 + (f''_2 - f'_2) * d_2.value,$
$\ldots\ldots \quad \ldots\ldots \quad ,$
$f_k = f'_k + (f''_k - f'_k) * d_2.value)$

where
[$(QUAL1(t, t_1, \ldots, t_n, s_1, \ldots, s_m)$ and $d_1.value = 1)$
or
$(\overline{QUAL1}(t, t_1, \ldots, t_n, s_1, \ldots, s_m)$ and $d_1.value = 0)$
]
and
[$(QUAL2(t', t'_1, \ldots, t'_n, r_1, \ldots, r_l)$ and $d_2.value = 1)$
or
$(\overline{QUAL2}(t', t'_1, \ldots, t'_n, r_1, \ldots, r_l)$ and $d_2.value = 0)$
]

where

$f'_i = t.f_i + [\, F_i(t, t_1, \ldots, t_n, s_1, \ldots, s_m) - t.f_i \,] * d_1.value$

$f''_i = G_i(t', t'_1, \ldots, t'_n, s_1, \ldots, s_m)$

t'_j = the tuple t_j where its fields f_i $(1 \leq i \leq k)$ are changed to f'_i

$\overline{QUAL1}$ and $\overline{QUAL2}$ are the negations of $QUAL1$ and $QUAL2$ respectively, and

d_1 and d_2 are range variables over some dummy relation DUM(value) with a single field *value*. This relation contains only two tuples with values 0 and 1 respectively.

What the above transformation proposes is to simply propagate the updates of the first replace to the qualification of the second one and then merge the two operations into one, in the same way the composition of two functions is performed.

To give an example, the following two commands

```
replace EMP (dept="shoe")
    where EMP.age < 40                    [EMP←f(EMP)]

replace EMP (salary=1.1*EMP.salary)
    where EMP.age < 40                    [EMP←g(EMP)]
```

can be obviously replaced by

```
replace EMP (dept="shoe", salary=1.1*EMP.salary)
    where EMP.age < 40                    [EMP←g(f(EMP))]
```

As a harder example where more complex composition takes part, consider

```
replace EMP (salary=1.1*EMP.salary)
    where EMP.age < 40                    [EMP←f(EMP)]

replace EMP (dept="shoe")
    where EMP.salary < 25                 [EMP←g(EMP)]
```

These two will now be replaced by

```
replace EMP (dept=EMP.dept+("shoe"-EMP.dept)*d2.value,
             salary=EMP.salary+0.1*EMP.salary*d1.value)
    where
      [(EMP.age < 40 and d1.value=1) or (EMP.age ≥ 40 and d1.value=0)]
    and
      [(EMP.salary+0.1*EMP.salary*d1.value < 25 and d2.value=1)
       or
       (EMP.salary+0.1*EMP.salary*d1.value ≥ 25 and d2.value=0)]
                                          [EMP←g(f(EMP))]
```

Notice that we have generally expressed differences using the standard "-" operator. It is not difficult to define this operator for strings also, so that if s is a string

```
s-s=0
s+0=s
s-0=s
s*1=s
```

s*0=0

The importance of the above transformation relies on the fact that the relation updated is opened and accessed only once; moreover, the query processing engine can make the new command more efficient than the separate execution of the two initial commands. The undesirable effect of producing a much more complex qualification is mainly due to the fact that negation is not handled well in query languages like QUEL. Due to that, the variables d_1 and d_2 had to be introduced. The modified qualification will in many cases require accessing all tuples of the relation unless more clever processing techniques are used (e.g. techniques that recognize that a tuple will either satisfy *QUAL* or $\overline{QUAL}$ but never both and will therefore avoid searching all tuples more than once). We can also show with similar transformations that two append or two `delete` aggregate-free commands can be merged to a single append or `delete` respectively.

Examining other combinations of update commands it can be seen that there is no easy (and sometimes there is not at all a) way to combine two different commands in one. For example, an append followed by a `replace` cannot be generally changed to a single append or `replace`. The reason is that an append cannot be processed as an equivalent `replace` command since the latter only modifies existing tuples and cannot insert new values, and vice versa. Therefore, the effects of either the one or the other command cannot be reflected through a single operation. In cases where this combination is not possible, the discussion of the previous subsection gives some ideas on producing more efficient programs by changing the order in which the given com-

mands are executed or modifying the qualifications.

3.5. Some Special Case Transformations

In this section we present two new optimization techniques which extend the technique of combining operations mentioned above. Each transforms a sequence of QUEL+ commands to a single `replace` command. The transformation of several commands to a single `replace*` command can yield significant savings. It allows the optimizer to concentrate its efforts on processing one canonical type of command, namely `replace`. Since `replace` does not change the size of the relation, the optimizer need make no estimates about that size. Experimental evidence [KUNG84] also indicates that such a transformation does in fact save significant processing time for a particular class of problems.

3.5.1. Bounded Problem Space Problems

Consider a QUEL+ command where only one relation, say R, is modified and this relation is known to be a subset of some other relation S, where S is known before the execution of the given QUEL+ command. It is also known that R remains a subset of S throughout the execution of the given command. We will show that in this case it is possible to transform the given command to a single `replace` or, in the case of `execute*`, in a single `replace*` program.

In order to show that this transformation to a single command is possible, we first note the result of the previous section, which shows that any two `replace` commands can be combined to a single `replace`. Thus we need only show that any database operation on R can be expressed as a `replace` command. We do that by

constructing a relation S' which is equal to S with the addition of a new field, *Present*, with the following semantics :

- a tuple from S that is currently in R will have a 1 in its *Present* field in S'
- a tuple from S that is not currently in R will have a 0 in its *Present* field in S'

We will now show that every database operation on R is equivalent to a `replace` command on S' by giving a set of transformations.

T1. An `append` command is transformed to a `replace` command where the tuples that satisfy the qualification change their *Present* field value to 1. That is

```
range of r1,r2,...,rm is R
range definitions for other tuple variables

append to R (f1 = val1,...,fk = valk)
    where q (r1,r2,...,rm,....)
```

becomes

```
replace s (Present = 1)
    where s.f1 = val1
    and     .....
    and   s.fk = valk
    and   q (s1,s2,...,sm,...)
    and   s1.Present = 1
    and     .....
    and   sm.Present = 1
```

T2. A `delete` command is transformed to a `replace` command where the tuples that satisfy the qualification change their *Present* field value to 0.

```
range of r,r1,r2,...,rm is R
range definitions for other tuple variables

delete r
   where   q (r1,r2,...,rm,....)
```

becomes

```
replace s (Present = 0)
    where q  (s1,s2,...,sm,....)
    and    s1.Present = 1
    and      .....
    and    sm.Present = 1
```

T3. Finally, a `replace` command is transformed to two new `replace` commands: the first one corresponds to the deletion of the old tuples while the second one corresponds to the addition of the new version of the deleted tuples. As was shown in the previous section, these two commands can be merged to a single `replace`.

In all of the above cases the tuple variables s_i range over S'. Notice that the basic idea in the transformations proposed, is the addition in the qualification of the clauses

$$s_i.Present = 1$$

for all tuple variables r_i that range over the given relation R. This clause simply states that the tuples that should be referenced from S are only those that would normally be in R, i.e. those that result from append or `replace` commands ($Present$=1) and not those that have been deleted ($Present$=0). This *query modification* process is very similar to the one proposed in [STON86b] in support of data managers that use optical disks to store the data. There a deletion does not imply the removal of tuples. Deleted tuples are simply marked as being invalid. It is also clear that in the case of an append command one need not include all fields in the new qualification. Only those fields that constitute a key should be included. The number of such fields is in most of the cases less than the total number of fields and the size of S much less

than the cartesian product of the domains of the fields of R.

A problem that arises in some cases is that the relation S is not known in advance or it is an extremely large relation. In the first case this transformation simply cannot be used and other optimization techniques must be used to get a better version for the database procedure. In the second case it is possible to do the transformation but not necessarily beneficial because of the size of S. In such cases special algorithms may need to be devised. We see an example of this in the next subsection.

3.5.2. Dynamic Programming Problems

The problems discussed in this section share the property that all are some implementation of the dynamic programming paradigm. A STATES relation, which contains (in each tuple) the current best value of the cost to be maximized, is built using the usual dynamic programming method. The example used is the shortest path problem described in the beginning of this chapter. The same tactic to be presented here can be used with other standard applications of dynamic programming as well, like the knapsack problem or the reliability problem.

A complete QUEL+ program for the shortest path example appears in section 3.1. There, the relation FEASIBLE (source,dest,cost) is fixed and the relation STATES (dest,cost,open) contains at all times the current state of knowledge about the problem, i.e. the cost of getting from the original source point S to any point *dest* in the given map. If we were to apply the technique of the previous section, we would seek a fixed relation S which contains STATES for the lifetime of the algorithm's exe-

cution. The problem is that such an S would have to hold a large number of tuples for each node, namely one tuple for every number less than the current cheapest cost of getting to that node. We will propose here a way to overcome this problem. The program that solves the SP problem, like any program using the dynamic programming approach, consists of two phases.

> In the first phase the relation STATES is expanded with the introduction of new nodes, i.e. the ones that can now be reached in the search space
> *(expansion phase)*

> Then in the second phase, nodes with the same *dest* value are compared and all but one are deleted according to some criterion, e.g. the cost of getting from the initial node S to that specific node
> *(optimality phase)*

The main loop of the program would be

```
range of r,r',r1,r2,...,rm is STATES
range definitions for other tuple variables

execute*
  {
   /* expansion phase */
  append to STATES (dest = val0,
                    f1 = val1,
                    f2 = val2,
                      ....  ,
                    fk = valk)
       where q(r1,r2,...,rm,....)

   /* optimality phase */
   delete r
       where r.dest = r'.dest
       and   w(r,r',...)
  }
```

Moreover the condition w is such that $w(r,r',...)$ and $w(r',r,...)$ cannot be both true (antisymmetric relation of r and r'). This means that only one tuple with a specific

value of *r.dest* will remain in the STATES relation after the optimality phase.

Let us now show that the above program can be transformed to a single `replace` command. First, we add a field *Present* to the STATES relation and call the new relation NSTATES. Assume that initially all tuples in the NSTATES relation have their *Present* field value equal to 0. As was explained in the previous section an `append` command will set the corresponding *Present* value to 1 while a `delete` will reset it to 0. Then the first command of the above program will be transformed to

```
range of s,s',s1,s2,...,sm is NSTATES
range definitions for other tuple variables

replace s (f1 = val1,...,fk = valk, Present = 1)
    where  s.dest = val0
    and    q (s1,s2,...,sm,....)
    and    s1.Present = 1
    and        .....
    and    sm.Present = 1
```

Note that we have used the fact that *dest* is a key in order to identify the tuple from NSTATES to be updated. Hence, all restrictions of the form $s.f_i{=}val_i$ have been eliminated.

An attempt to transform the second command using the transformations from the previous section would fail since in the NSTATES relation there cannot be two tuples with the same *dest* value. So the second command should be translated as follows

> **if** the tuple appended during the expansion phase is the first one appended to NSTATES for that value of the *dest* field (i.e. $r.Present{=}0$), **then** do the update,
>
> **else** do the update only if the new tuple would not be deleted by the second

command, i.e. if $w(s,(v_0,v_1,v_2,...,v_k),....)$ is true, which guarantees that this tuple will not be deleted by the `delete` command.

This interpretation allows us to omit the optimality phase command by only enhancing the qualification of the `replace` command that the initial append operation was transformed to (see similarities with the example presented in section 3.4). The final one-replace command program will be

```
replace s (f_1 = val_1, ...,f_k = val_k, Present = 1)
    where   s.dest = val_0
    and     q (s_1,s_2,...,s_m, . . . . .)
    and     s_1.Present = 1
    and        . . . . .
    and     s_m.Present = 1
    and     (s.Present = 0  or  w(s,(val_0,val_1,...,val_k), . . . .))
```

We should also note here that the command shown above might now be ambiguous. There may be more than one value to be assigned to a single tuple (non-functional update). This corresponds to the case where many tuples with the same *dest* field value are appended to STATES, due to the existence of multiple paths from S to *dest*. However, this is a general problem of ambiguous updates and in our case is easy to solve by using the condition w to eliminate tuples with higher cost.

We have shown how the above dynamic programming problem for search spaces has been reduced to a single `replace*` program. The difference between the two programs is that the first one starts with a rather small relation which incrementally grows as the iterations are executed while the second one starts with the whole problem space and updates the information recorded about the nodes. What remains to be examined is how this new version compares in execution time and I/O operations with the initial version of the algorithm. The result of this comparison depends not only on

the size of the NSTATES relation but also on the fraction of it that will be used in the program. It has been shown through a series of experiments that Dynamic Programming problems is a class of problems that will gain in performance from this transformation [KUNG84]. The single replace command runs ten times faster than the initial two command program for a FEASIBLE relation with 100 nodes, almost 100 times faster for 400 nodes and infinitely faster for more than 500 nodes.

3.6. Summary

The problem of optimizing extended query language commands and in particular sequences of QUEL commands (procedures) was described. Our presentation included several optimization tactics, some based on similar tactics in other areas and some new ones.

Moving invariant aggregate computations out of loops and incremental computation of aggregates were used as examples to illustrate how iterative constructs can be made more efficient. Another aspect that can be found in compiler design as well, common subexpression analysis and reusal of common intermediate results, is discussed in detail in the chapter that follows. The ideas of performing early restrictions and combining of operations where drawn from conventional query optimization and abstracted in our environment as merging database commands. Physical database design techniques [SCHK78] are also applicable in the environment of QUEL+. The optimizer is given a set of data, namely the given relations and their organization, plus a set of commands, and some information about the frequency of the commands. It then seeks an optimal reorganization (perhaps none) of the physical database. What is

missing in the case of procedures is complete data on the frequency of the commands but estimation techniques like the ones referenced in [KUNG84] can be devised. Also, the QUEL+ optimizer must take care not to reorganize the database in a way which will degrade future performance, e.g. creating an index which will slow down updates for future commands which do not use that index. The solution to the above problem is to create temporary secondary structures (indexes) or primary organizations that will be used during the execution of a repetitive command or a procedure but they will not persist beyond that.

Finally, some special case transformations that are applicable to database procedures with a specific structure were discussed. Our new tactics include the somewhat surprising result that any QUEL program satisfying certain criteria is equivalent to a QUEL program which consists of one `replace` statement. We have also shown that a large class of problems, namely those which use the dynamic programming approach, satisfy these criteria. The transformations presented are useful not only in this context but in general transaction processing as well, since they are motivated solely by the need to expand the optimization unit from one database language command to a sequence of commands. Experimental results have shown that these transformations require minimal effort to be applied; in return, performance gains are substantial.

CHAPTER 4

MULTIPLE QUERY OPTIMIZATION

4.1. Introduction

The discussion of the previous chapter, suggested a set of transformations and tactics for optimizing collections of commands in the presence of updates. In this chapter we examine a special case which gives rise to more elegant and general solutions to the multiple command processing problem. The *retrieve-only* case where the set of commands to be evaluated is restricted to retrieve queries only is studied. Such sets arise in the QUEL+ environment if a procedure stored in a QUEL field is solely retrieving data from the database.

However, there are many other applications where more than one query are presented to the system in order to be processed. First, consider a database system enhanced with inference capabilities (*deductive database system)* [GALL78]. A single query given to such a system may result to more than one actual queries that will have to be run over the database. As an example, consider the following relation for employees

```
EMP (name,salary,experience,manager,dept_name)
```

Assume also the existence of a set of rules that define when an employee is well paid. We will express these rules in terms of `retrieve` commands.

```
/* An employee is well paid if he/she makes more than 40K */
```

Rule 1: `retrieve (EMP.all) where EMP.salary > 40`

/* An employee is well paid if he/she makes more than 35K provided he/she has no more than 5 years of experience */

Rule 2: `retrieve (EMP.all) where EMP.salary > 35 and EMP.experience ≤ 5`

/* An employee is well paid if he/she makes more than 30K provided he/she has no more than 3 years of experience */

Rule 3: `retrieve (EMP.all) where EMP.salary > 30 and EMP.experience ≤ 3`

Then a query that asks

Is Mike well paid?

will have to evaluate all three rules in order to come up with the answer. Because of the similarities that PROLOG [CLOC81] clauses have with the above type of rules, our discussion on multiple query processing applies to the optimization of PROLOG programs as well, assuming that secondary storage is used to hold a PROLOG database of facts. As a second example, consider cases where queries are given to the system from various users. Then *batching* all users' requests is a possible processing strategy. In particular, queries given within the same time interval r may be considered to be processed all together (we will see in the following what "all together" means). Finally, some proposals on processing recursion in database systems [NAQV84,IOAN86], suggest that a recursive Horn clause should be transformed to a set of other simpler Horn clauses (recursive and non-recursive). Therefore, the problem of multiple query processing arises in that environment as well. However, it is more complicated because of the presence of recursive queries.

Current query processors cannot optimize the execution of more than one queries. If given a set of queries, the common practice is to process each query separately. There are generally many possible ways of executing a query (*access plans*). For example, there may be a choice of indexes to use, or a choice of strategies for executing a relational operator such as the join. Access plans are simply sequences of such simple tasks as relation scans, index scans, etc. The query processor chooses the cheapest among these plans and then executes it to produce the result of the query. In the case where more than one query is given at the same time there is another possible optimization, namely sharing of common operations (or *tasks*). Examples of such tasks may be performing the same restriction on the tuples of a relation or performing the same join between two relations. Taking advantage of these common tasks, mainly by avoiding redundant page accesses, may prove to have a considerable effect on execution time.

The presentation of the multiple query optimization problem is the focus of this chapter and is organized as follows. Section 4.2 presents an overview of previous work done in similar problems while Section 4.3 first defines the query model that will be used throughout this chapter and then presents a formulation for the multiple (or *global*) query optimization problem. Section 4.4 presents our approach to the problem and introduces through the use of some examples, algorithms that can be used to solve the multiple query optimization problem. Then, Sections 4.5 through 4.7 present these algorithms in more detail. Section 4.5 suggests an algorithm which finds a serial sequence for executing the queries with better performance than any other serial execution which executes the queries in an arbitrary order. Then, in Section 4.6 we

describe an algorithm that goes one step further by allowing the executions of the queries to interleave, while Section 4.7 proposes a more general heuristic algorithm. Finally, Section 4.8 presents some experimental results and the last section concludes the presentation of the multiple query processing problem by summarizing our results.

4.2. Previous Work

Problems similar to the problem of multiple query processing have been examined in the past in various contexts. Hall [HALL74,HALL76] for example, uses heuristics to identify common subexpressions, especially within a single query. He uses operator trees to represent the queries and a bottom-up traversal procedure to identify common parts. In [GRAN80] and [GRAN81] Grant and Minker describe the optimization of sets of queries in the context of deductive databases and propose a two stage optimization procedure. During the first stage ("Preprocessor") the system obtains at *compile* time (i.e. at the time the queries are given to the system) information on the access structures that can be used in order to evaluate the queries. Then, at the second stage, the "Optimizer" groups queries and executes them separately as groups instead of one at a time. During that stage common tasks are identified and sharing of the results of such tasks is used to reduce processing time.

Roussopoulos in [ROUS82a] and [ROUS82b] provides a framework for interquery analysis based on query graphs [WONG76], in an attempt to find fast access paths for view processing (view indexing). The objective of his analysis is to identify all possible ways to produce the result of a view, given other view definitions and ground relations. Indexes are then built as data structures to support fast processing of views.

Other researchers have also recently examined the problem of global query optimization. Chakravarthy and Minker [CHAK82,CHAK85] propose an algorithm based on the construction of integrated query graphs. These graphs are extensions of the query graphs introduced by Wong and Youssefi in [WONG76]. Using integrated query graphs, Chakravarthy and Minker suggest a generalization of the query decomposition algorithm of [WONG76]; however, this algorithm does not guarantee that the access plan constructed is the cheapest one possible. Kim in [KIM84] suggests also a two stage optimization procedure similar to the one in [GRAN81]. The unit of sharing among queries in Kim's proposal is the relation which is not always the best thing to assume, except in cases of single relation queries.

The work of [FINK82] and [LARS85] on the problem of deriving query results based on the results of other previously executed queries, is also related to the problem of multiple query optimization. The solutions suggested are useful to our analysis because they include efficient algorithms to detect common subexpressions among queries. These subexpressions characterize the data that is shared and accessed by more than one query. Jarke also discusses in [JARK84b] the problem of common subexpression isolation. He presents several different formulations of the same problem under various query language frameworks such as relational algebra, tuple calculus and relational calculus. In the same article he also describes how common expressions can be detected and used according to their type (e.g. single relation restrictions, joins, etc).

The main objective of our approach to multiple query processing is to use existing query optimizers as much as possible. We would like to avoid making significant

changes to the query optimizer; instead, our goal is to provide a preprocessor that will reduce the execution cost as much as possible. This preprocessing phase is introduced as an extra step between the optimizer and the execution modules. However, since not all relational database systems have been designed based on the same query processing concepts, we will differentiate between two alternative architectures that can be used for a system with multiple query processing capability. Figure 4.1 illustrates these two approaches. Architecture 1 can be used with minimal changes to existing optimiz-

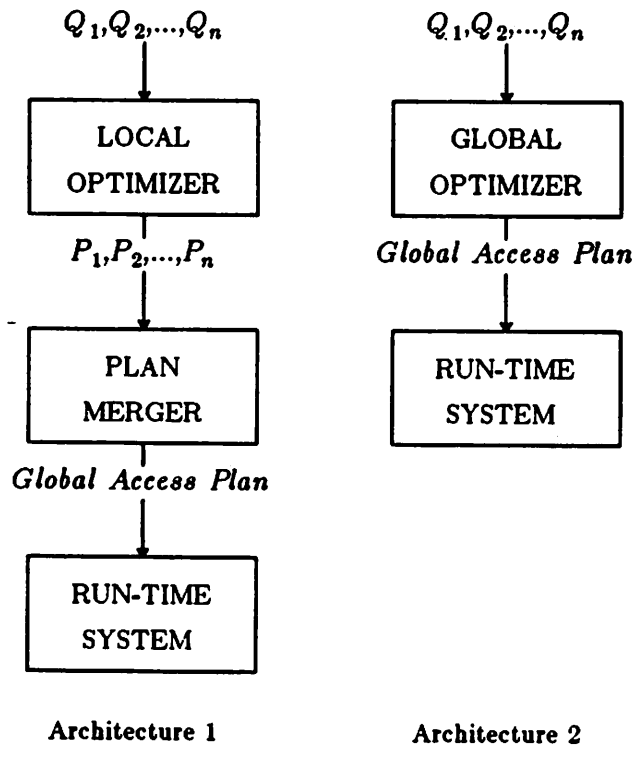


Figure 4.1: Multiple Query Processing Systems Architecture

ers. A conventional *Local Optimizer* generates one (*locally*) optimal access plan per query. The *Plan Merger* is a component which examines all n access plans and generates a larger plan, the global access plan, which is in turn processed by the *Run-Time Syetem*. In many existing systems queries are *compiled* and saved in the form of access plans (see for example System-R [ASTR76] and POSTGRES [STON86b]). It is then an interesting problem to derive procedures that, given a set of such plans, identify a sequence in which they must be run in order to reduce the I/O and/or CPU cost. More sophisticated procedures can also be used for that reason. For example, Chakravarthy and Minker [CHAK85] describe an algorithm to process multiple joins involving the same relation R by scanning R once and examining several restriction conditions in parallel. Using such a procedure though implies rewritting the query processor which, as we argued above, requires a major effort.

On the other hand, there are systems that do not store access plans for future reusal (e.g. INGRES [STON76]). To make our framework general enough to capture these systems as well, we introduce Architecture 2. The set of queries is processed by a more sophisticated component, the *Global Optimizer*, which in turn passes the derived global access plan to the *Run-Time Systeм* for processing. Architecture 2 therefore is not restricted to using locally optimal plans already stored in the system.

The purpose of the following sections is to exhibit a set of optimization algorithms that can be used for multiple query optimization either as Plan Mergers or as Global Optimizers. The algorithms to be presented differ on the complexity of the Plan Merger and on whether Architecture 1 or 2 is used. The trade offs between the complexity of the algorithms and the optimality of the global plan produced are also

discussed.

4.3. Formulation of the Problem

We assume that a *database* **D** is given as a set of *relations* $\{R_1, R_2, \ldots, R_m\}$, each relation defined on a set of *attributes* (or fields). A set of queries $\mathbf{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ on **D** is also given. A simple model for queries is now described. A *selection predicate* is a predicate of the form $R.A$ *op cons* where R is a relation, A a field of R, $op \in \{=, \neq, <, \leq, >, \geq\}$ and *cons* some constant. A *join predicate* is a predicate of the form $R_1.A = R_2.B$ where R_1 and R_2 are relations, A and B are fields of R_1 and R_2 respectively. For simplicity we will assume that the given queries are conjunctions of selection and join predicates and *all* attributes are returned as the result of the query (i.e. we assume no *projection* on specific fields). Clearly the above model excludes aggregate computations or functions as well as predicates of the form $R_1.A$ *op* $R_2.B = R_3.C$. Extending a system to support such predicates is possible but would require significant increase in its complexity. The restriction on conjunctive queries only is not a severe limitation since the result of a disjunctive query can be considered of as the union of the results of the disjuncts, i.e. each disjunct can be thought as a different query. Equijoins are also the only type of joins allowed among relations. This assumption is made in all the proposals mentioned in the previous section and seems quite natural considering the most common types of queries. Finally, not allowing projections enables us to concentrate on the problem of using effectively the results of common subexpressions rather than the problem of detecting if the result of a query can be used to compute the result of

another query. Assuming projection lists, does not increase the complexity of the algorithms that perform multiple query optimization. It only increases the complexity of the algorithms that detect common subexpressions among queries. The proposals of [LARS85] and [FINK82] provide such algorithms.

A *task* is an expression *relname* ← *expr*. *relname* is a name of a temporary relation used to store an intermediate result or the keyword *RESULT*, indicating that this task provides the result of the query. *expr* is a conjunction of either selection predicates over the same relation or joins between two, possibly restricted, relations. This latter type covers queries that are processed not by performing the selections first followed by the join, but in a "pipelining" way. For example, consider the following query on the relations EMP (name, age, dept_name) and DEPT (dept_name, num_of_emps)

```
retrieve (EMP.all, DEPT.all)
    where EMP.age ≤ 40
    and DEPT.num_of_emps ≤ 20
    and EMP.dept_name = DEPT.dept_name
```

One way to process the query is by scanning the relation EMP and having each employee tuple with qualifying age be checked across the DEPT relation. There is no need in storing intermediate results for both EMP and DEPT. To be able to include this kind of processing in our model, the second type of join tasks was introduced. In the remaining discussion, tasks will be referred to as if they were simply the *expr* part, unless otherwise explicitly stated.

Let us define now a partial order on tasks. A task t_i *implies* task t_j ($t_i \Longrightarrow t_j$) iff t_i is a conjunction of selection predicates on attributes A_1, A_2, ..., A_k of some relation

R, t_j is a conjunction of selection predicates on the same relation R and on attributes A_1, A_2, ..., A_l with $l \leq k$ and it is the case that for any instance of the relation R the result of evaluating t_i is a subset of the result of evaluating t_j.

A task t_i is *identical* to task t_j ($t_i \equiv t_j$) iff

a) *Selections* : $t_i \Longrightarrow t_j$ and $t_j \Longrightarrow t_i$

b) *Joins* : t_i is a conjunction of join predicates $E_1.A_1 = E_2.B_1$, $E_1.A_2 = E_2.B_2$,..., $E_1.A_k = E_2.B_k$ and t_j is a conjunction of join predicates $E'_1.A_1 = E'_2.B_1$, $E'_1.A_2 = E'_2.B_2$,..., $E'_1.A_k = E'_2.B_k$ where each of E_1, E_2, E'_1 and E'_2 is a conjunction of selections on a single relation and $E_1 \equiv E'_1$ and $E_2 \equiv E'_2$

Based on the above definition for tasks we now define the notion of an access plan. An *access plan* for a query Q is a sequence of tasks that produces the result of answering Q. Formally, an access plan is an acyclic directed graph $P=(V,E,L)$ (V, E and L being the sets of vertices, edges and vertex labels respectively) defined as follows :

- For every task t of the plan introduce a vertex v
- If the result of a task t_i is used in task t_j, introduce an edge $v_i \rightarrow v_j$ between the vertices v_i and v_j that correspond to t_i and t_j respectively
- The label $L(v_i)$ of vertex v_i is the processing done by the corresponding task t_i (i.e. *relname* $\leftarrow$ *expr*)

For example, consider the following query on the relations EMP (name,age,dept_name) and DEPT (dept_name,num_of_emps)

```
retrieve (EMP.all,DEPT.all)
    where EMP.age ≤ 40
    and DEPT.num_of_emps ≤ 20
    and EMP.dept_name = DEPT.dept_name
```

One way to process this query is

```
TEMP1   ← EMP.age ≤ 40
TEMP2   ← DEPT.num_of_emps ≤ 20
RESULT  ← TEMP1.dept_name = TEMP2.dept_name
```

The graph of Figure 4.2 shows the corresponding access plan.

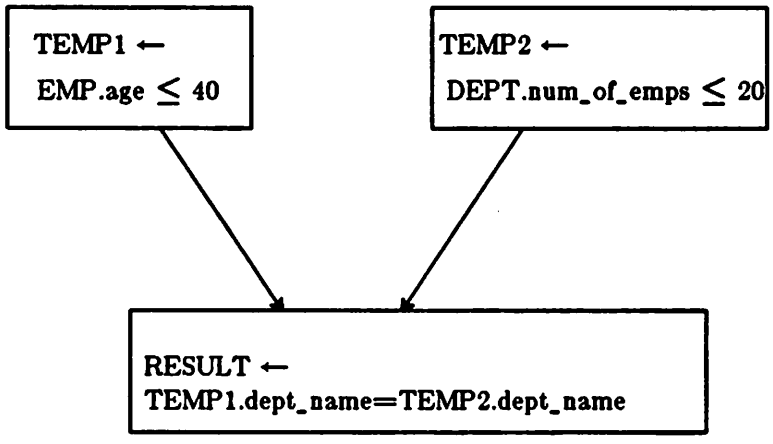


Figure 4.2: Example of an Access Plan

Notice that there are generally many possible plans that can be used in order to process a query.

Next we define a cost function $cost : V \rightarrow \mathbb{Z}$ for tasks. In general this cost depends on both the CPU time and the number of disk page accesses needed to process the given task. However, to simplify the analysis, we will consider only I/O costs. Including CPU costs would only make the formulas more complex (see for example

[SELI79]). Therefore,

$$cost(v_i) = \text{the number of page accesses needed to process task } t_i$$

The cost $Cost(P)$ of an access plan P is defined as

$$Cost(P) = \sum_{v_i \in V} cost(v_i)$$

We will refer to the minimal cost plans for processing each query Q_i individually, as *locally optimal* plans. Similarly, we use the term *globally optimal* plan to refer to an access plan that provides a way to compute the results of all n queries with minimal cost. The union of the locally optimal plans is generally different than the globally optimal plan. Finally, for a given query Q, $Bestcost(Q)$ gives the cost of the (locally) optimal plan P. Hence, $Bestcost(Q) = \min_{p \in \mathbf{P}}[Cost(p)]$, where $\mathbf{P}$ is the set of all possible plans that can be used to evaluate Q.

Let us now consider a system that given a set $\mathbf{Q}$ of queries it is required to execute them with minimal cost. According to the above definitions, a global access plan is simply a directed labeled graph that provides a way to compute the results of all n queries. Based on this formulation, the problem of global query optimization becomes

> **Given** *n sets of access plans $\mathbf{S}_1, \mathbf{S}_2, ..., \mathbf{S}_n$, with $\mathbf{S}_i = \{P_{i1}, P_{i2}, ..., P_{ik_i}\}$ being the set of possible plans for processing Q_i, $1 \leq i \leq n$,*
>
> **Find** *a global access plan GP by "merging" n local access plans (one out of each set $\mathbf{S}_i$) such that $Cost(GP)$ is minimal*

The Plan Merger or the Global Optimizer of Figure 4.1 performs the "merging" operation mentioned above. It is the purpose of the following sections to define this operation and derive algorithms that find GP.

4.4. A Hierarchy of Algorithms

The primary source of redundancy in multiple query processing is accessing the same data multiple times in different queries. Recognizing all possible cases where the same data is accessed multiple times requires in general a procedure equivalent to theorem proving, including retrieving data from the database. Our intention here is to detect common subexpressions looking only at the logical expressions used in the descriptions of queries, that is by simply isolating pairs of expressions e_1 and e_2 where $e_1 \Longrightarrow e_2$. Therefore, detection of sharing is done at a high level using only the query expressions (qualifications) and without going to the actual data stored in the database. For example, e_1 may be EMP.age $\leq$ 30 and e_2 may be EMP.age $\leq$ 40. Then $e_1 \Longrightarrow e_2$. However, we do not consider cases where e_2 may be EMP.dept_name = "shoe" and it happens in the specific instance of the database that all employees under 40 years old are the shoe department. Unless such a rule is explicitly known to the system in the form of an integrity constraint or functional dependency, it is not possible to detect that $e_1 \Longrightarrow e_2$ without looking at the actual data stored [JARK84a,CHAK84,CHAK86]. Hence, query expressions are considered to be the only source for detecting common subexpressions. Because several algorithms have been published in the past on the problem of common subexpression isolation [ROSE80,FINK82,LARS85] we will not attempt here to present a similar algorithm. It is assumed that a procedure which decides, given two expressions e_1 and e_2, if $e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$, is available.

Second, as it was stated in the previous section, many systems store in the database optimal local access plans that have been produced in the past (e.g. System-R

[ASTR76] and POSTGRES [STON86b] choose to do so). Because it is not realistic to expect from the system to store more than one plan for each query, it is assumed that only locally optimal access plans are stored. Then, if a set of queries is given, there is no need to generate new plans for those queries that have precomputed plans already stored in the database. However, for the rest of the queries, optimal plans are produced and saved for future reusal. When both precomputed and newly generated plans are available the global access plan is derived.

The various algorithms that can be used for global query optimization are grouped in a hierarchy shown in Figure 4.3. The reason the algorithms are organized

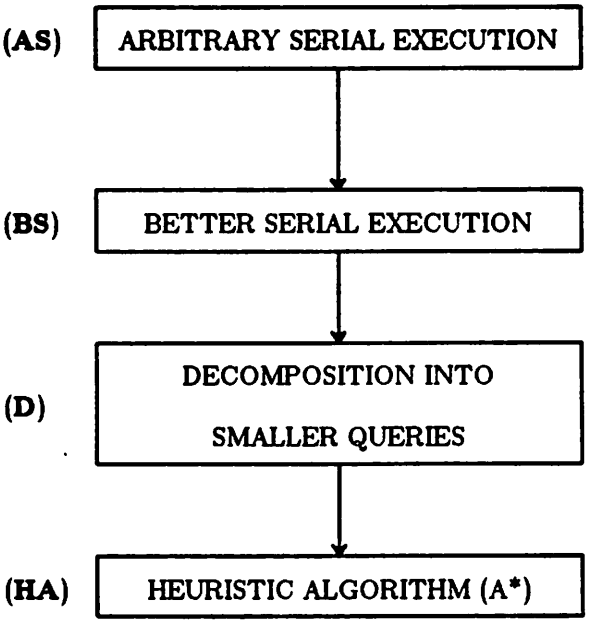


Figure 4.3: A Hierarchy of Multiple Query Processing Algorithms

in such a hierarchy is to indicate the interesting trade off between the time spent for optimization and the cost of executing the resulting global access plan. As we descend the hierarchy, the complexity of the algorithm increases while the access plan cost decreases. Algorithms **AS**, **BS** and **D** consider only access plans that are *locally* optimal. As mentioned above, the locally optimal plan for executing a query Q is derived by considering Q alone. Algorithm **AS** (Arbitrary Serial Execution) simply executes these plans in an arbitrary order. This corresponds to Architecture 1 of Figure 4.1 with the Plan Merger absent, i.e. no optimization is performed. Algorithm **BS** (Better Serial Execution) preprocesses the plans and generates a better order of execution so that intermediate results (temporaries) are reusable. In this case the Plan Merger of Figure 4.1 simply rearranges the order in which the plans are processed. Notice that in both algorithms **AS** and **BS** the unit of execution is a whole query, i.e. the second query is processed after the first one has been totally processed.

Algorithm **D** (Decomposition) presents a different paradigm. A query is decomposed into smaller subqueries which now become the unit of execution. Therefore, a query is not processed as a whole but rather in small pieces, the results of which are assembled at various points to produce the result. As an example why **D** might be a better algorithm than **BS**, consider the following database,

```
EMP (name,age,salary,job,dept_name)
DEPT (dept_name,num_of_emps)
JOB (job,project)
```

with the obvious meanings for EMP, DEPT and JOB. We also assume that there are no fast access paths for any of the relations, and that the following queries

```
(Q1)  retrieve (EMP.all,DEPT.all)
        where EMP.age ≤ 40
        and DEPT.num_of_emps ≤ 20
        and EMP.dept_name = DEPT.dept_name

(Q2)  retrieve (EMP.all,DEPT.all)
        where EMP.age ≤ 50
        and DEPT.num_of_emps ≤ 10
        and EMP.dept_name = DEPT.dept_name
```

are given. If we run either Q_1 or Q_2 first we will be unable to use the intermediate results from the restrictions on EMP and DEPT effectively. However, the following global access plan is more efficient

```
retrieve into tempEMP (EMP.all)
    where EMP.age ≤ 50

retrieve into tempDEPT (DEPT.all)
    where DEPT.num_of_emps ≤ 20

retrieve (tempEMP.all,tempDEPT.all)
    where tempEMP.age ≤ 40
    and tempEMP.dept_name = tempDEPT.dept_name

retrieve (tempEMP.all,tempDEPT.all)
    where tempDEPT.num_of_emps ≤ 10
    and tempEMP.dept_name = tempDEPT.dept_name
```

because it avoids accessing the EMP and DEPT relations more than once. It is drastically more efficient in the cases where restrictions reduce the sizes of the original relations significantly. The function of the Plan Merger, in the case of algorithm **D**, is to "glue" the plans together in a way that provides better utilization of common temporary (intermediate) results.

Finally, algorithm **HA** (Heuristic Algorithm) is based on searching among local (not necessarily optimal) query plans and building a global access plan by choosing one

local plan per query. Architecture 2 of Figure 4.1 applies to this case. The effectiveness of algorithm **HA** is illustrated with the following example. Suppose we have the queries

```
(Q3)  retrieve (JOB.all,EMP.all,DEPT.all)
         where EMP.dept_name = DEPT.dept_name
         and JOB.job = EMP.job

(Q4)  retrieve (EMP.all,DEPT.all)
         where EMP.dept_name = DEPT.dept_name
```

with optimal local plans

```
(P3)  retrieve into TEMP1 (JOB.all,EMP.all)
         where JOB.job = EMP.job
         retrieve (TEMP1.all,DEPT.all)
         where TEMP1.dept_name = DEPT.dept_name

(P4)  retrieve (EMP.all,DEPT.all)
         where EMP.dept_name = DEPT.dept_name
```

respectively. Notice that P_3 and P_4 do not share the common subexpression EMP.dept_name=DEPT.dept_name. Algorithm **HA** considers in addition to P_3 the plan that processes the join EMP.dept_name=DEPT.dept_name. It also uses some heuristics to reduce the number of permutations of plans it has to examine in order to find the optimal global plan. All the above algorithms are examined in more detail in the following three sections.

4.5. Serial Execution

Algorithms **AS** and **BS** of Figure 4.3 are based on some serial execution of the given queries $Q_1, Q_2, \ldots, Q_n$. As stated in the previous section we only consider the locally optimal plans P_i, $1 \leq i \leq n$. In the first case no restrictions are imposed on the

order in which the queries are processed; that is what a conventional query processor would do. In the second case some simple preprocessing is done aiming to better performance.

4.5.1. Arbitrary Serial Execution

In Algorithm **AS** the sequence in which the queries are run is chosen arbitrarily. We assume that all queries are processed without taking advantage of any common tasks that they may share. The global plan GP that is produced is simply the concatenation of the locally optimal plans for the queries in an arbitrary way. Therefore, for any order of processing $\mathbf{S} = \{Q_{i_1} Q_{i_2} \cdots Q_{i_n}\}$, with $Q_{i_k} \in \mathbf{Q}$ and all i_k distinct, the cost of the global access plan will be

$$Cost(GP) = \sum_{j=1}^{n} Bestcost(Q_{i_j})$$

As an example, consider the following queries Q_5 and Q_6

```
(Q5)  retrieve (EMP.all,DEPT.all)
          where EMP.age ≤ 40
          and EMP.salary ≤ 10
          and EMP.dept_name = DEPT.dept_name

(Q6)  retrieve (EMP.all,DEPT.all)
          where EMP.age ≤ 40
          and EMP.dept_name = DEPT.dept_name
```

Assume also that the sizes of the initial relations and temporary results are as follows

```
size (EMP) = 100 pages
size (DEPT) = 10 pages
size (EMP.age≤40) = 20 pages
size (EMP[age≤40 and salary ≤ 10]) = 10 pages
```

It is also assumed that the local plans for Q_5 and Q_6 store temporaries for the above

restrictions. Then, processing S would require 110+C_j(10,10) page accesses for Q_5 and 120+C_j(20,10) page accesses for Q_6, where $C_j(a,b)$ is the cost of processing a join between two relations of sizes a and b pages. Hence, the total cost would be 230+C_j(10,10)+C_j(20,10) page accesses.

The above algorithm does not consider at all of reusing results that are produced as intermediate (temporary) relations. A simple extension would be to keep temporary relations after they are used so that subsequent queries may use them. Better than that, with some simple preprocessing we can find a serial execution that makes use of such temporary results. The next subsection presents such an approach.

4.5.2. Better Serial Execution

The goal of algorithm **BS** is to look at the optimal local plans and derive a serial execution schedule **S** that makes use of common subexpressions. Checking if a given temporary result can be used by another query is done through the procedure proposed in [FINK82].

The first step in deriving the execution schedule **S** builds a directed graph that will eventually suggest **S** using the directed paths of the graph. This kind of graph is very similar to the precedence graphs used in concurrency control [ULLM82] and it is used to indicate how the read set of one query is related to the read sets of other queries. If some query Q_i does not share any of its input relations with any other query, it is put first in the sequence **S**. These queries are not amenable to any optimization other than what the locally optimal plan suggests. For the rest of the queries we define the following directed labeled graph $QG(V,E,L)$, with V being the set of

vertices, E the set of edges and L a set of labels associated with edges

- For each plan $P_i(V_i,E_i,L_i)$ a node q_i is defined
- A directed edge $q_i \rightarrow q_j$ is introduced if

  a) *Proper Implication* : There are $v_i \in V_i$ and $v_j \in V_j$ such that $L_j(v_j) \Longrightarrow L_i(v_i)$ and $L_i(v_i) \not\Longrightarrow L_j(v_j)$

  b) *Identical Nodes* : There are $v_i \in V_i$ and $v_j \in V_j$ such that $L_j(v_j) \equiv L_i(v_i)$ and $i < j$

- Assume that edge $q_i \rightarrow q_j$ is introduced because of nodes v_i of P_i and v_j of P_j respectively. Then the label of the edge $q_i \rightarrow q_j$ is the savings in the cost of executing $L_j(v_j)$ given the result of $L_i(v_i)$. This cost is estimated assuming that one or more of the relations used in v_j are substituted by the temporary relation that is created in the task v_i.

Edges of type (a) are introduced to indicate which queries (tail of an edge) can be used in the evaluation of other queries (head of an edge). The second rule for edge definition is introduced to break ties between identical expressions in a specified manner. Algorithm **BS** then proceeds in the following way :

[1] If multiple edges with the same direction are found between two nodes q_i and q_j replace them with a single edge with label the sum of the labels of the previous edges.

[2] If the resulting graph is acyclic then the execution order **S** is derived from the directed paths that are imposed on the graph.

[3] If the resulting graph has cycles, these are broken by omitting a set of edges with minimal sum of labels. S is then produced as in [2].

Let $QG'(V,E',L)$ be the resulting graph. The last step of the above algorithm is a well known NP-complete problem, known as "the feedback arc set problem" [GARE79]. However, in multiple query optimization the graph will have few nodes equal to the number of queries that access common data and not many cycles. Therefore, this problem has only minor effect on the performance of the algorithm. A simple analysis shows that the formula for computing the estimated cost of the global plan imposed by the sequence S is

$$Cost(GP) = \sum_{i=1}^{n} Bestcost(Q_i) - \sum_{e \in E'} L(e)$$

$$= \sum_{i=1}^{n} Bestcost(Q_i) - \sum_{s \in CS} n_s \cdot savings(s)$$

where CS is the set of common subexpressions s found among the queries and used in the final graph QG', n_s is the number of times the result of a common subexpression s is used in the final sequence and $savings(s)$ is the cost that is saved if temporary results instead of ground relations are used. That cost is defined as follows:

Let R be a relation and s_1 and s_2 two subexpressions defined on R such that s_2 can be processed using the result of s_1 instead of R. Let also C_R be the cost of accessing R to evaluate s_1 and C_{s_1} be the cost of accessing the result of s_1 to evaluate s_2. Then

$$savings(s_2) = \begin{cases} C_R - C_{s_1} & \text{if } s_2 \Longrightarrow s_1 \\ C_R + C_{s_1} & \text{if } s_2 \equiv s_1 \end{cases}$$

In order to construct graph QG, the above algorithm requires time in the order of $\prod_{i=1}^{k} |V_i|$, where $k=|V|$ is the number of vertices of graph QG and V_i are the sets of vertices for plans P_i, $1 \leq i \leq k$. Step [3] is the most expensive step and in the worst case requires time exponential on the number of the edges.

Let us show with an example how **BS** works. Suppose that the queries Q_5 and Q_6 of the previous subsection are given. The directed graph constructed is shown in Figure 4.4.

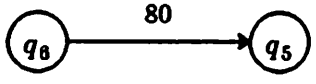


Figure 4.4: QG Graph for Queries Q_5 and Q_6

The edge $q_6 \rightarrow q_5$ is introduced because [EMP.age $\leq$ 40 and EMP.salary $\leq$ 10] $\Longrightarrow$ EMP.age $\leq$ 40. Therefore the serial execution will be $S=\{Q_6\, Q_5\}$ which uses 80 page accesses less than an arbitrary serial execution which was seen in the previous section, for a savings of 35%.

To give an example where a cyclic graph QG may occur, consider queries Q_1 and Q_2 of section 3

```
(Q1)  retrieve (EMP.all,DEPT.all)
            where EMP.age ≤ 40
            and DEPT.num_of_emps ≤ 20
            and EMP.dept_name = DEPT.dept_name

(Q2)  retrieve (EMP.all,DEPT.all)
            where EMP.age ≤ 50
            and DEPT.num_of_emps ≤ 10
            and EMP.dept_name = DEPT.dept_name
```

with optimal local plans

(P_1)

```
retrieve into tempEMP1(EMP.all)
    where EMP.age ≤ 40

retrieve into tempDEPT1(DEPT.all)
    where DEPT.num_of_emps ≤ 20

retrieve (tempEMP1.all,tempDEPT1.all)
    where tempEMP1.dept_name = tempDEPT1.dept_name
```

(P_2)

```
retrieve into tempEMP2(EMP.all)
    where EMP.age ≤ 50

retrieve into tempDEPT2(DEPT.all)
    where DEPT.num_of_emps ≤ 10

retrieve (tempEMP2.all,tempDEPT2.all)
    where tempEMP2.dept_name = tempDEPT2.dept_name
```

and sizes of relations and intermediate results

```
size (EMP) = 100 pages , size (DEPT) = 10 pages

size (tempEMP1) = 20 pages , size (tempEMP2) = 40 pages

size (tempDEPT1) = 3 pages , size (tempDEPT2) = 5 pages
```

Figure 4.5 shows the QG graph built for these queries. The edge $q_1 \rightarrow q_2$ is introduced because tempDEPT2 can be derived from tempDEPT1, while the edge $q_2 \rightarrow q_1$ is intro-

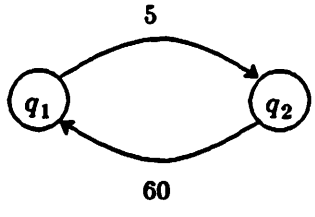


Figure 4.5: QG Graph for Queries Q_1 and Q_2

duced because tempEMP1 can be derived from tempEMP2. The cycle is broken by removing the edge $q_1 \rightarrow q_2$ for a total savings of 60 page accesses.

Although algorithm **BS** provides better plans than **AS** it still does not take advantage of all common subexpressions because of the requirement that queries must be run in some order and no interleaving is possible. In the next section we present another approach which takes advantage of all common subexpressions that can be identified in locally optimal plans.

4.6. Decomposition Algorithm

If query processing is done based on creating temporary intermediate relations, then it is known from existing algorithms [WONG76] that it is beneficial to break the query down to smaller and simpler subqueries. In the case of global query optimization, a similar approach seems promising also. Relaxing the assumption of the previous section which forced each plan to be processed totally before other plans start being processed, we will examine here the possibility of *interleaving* the execution of various access plans. Algorithm **D** (Decomposition) takes an approach based exactly on this idea of interleaved plan execution.

The main idea is to decompose the given queries into smaller subqueries and run those in some order depending on the various relationships among the queries. Then, the results of various subqueries are simply assembled to generate the answers to the original queries. The only restriction imposed is that the partial order defined on the execution of tasks in a local access plan, must be preserved in the global access plan as well. As it was the case in the previous algorithms, only locally optimal plans are con-

sidered. A final assumption made for algorithm **D** is that temporary intermediate results are replacing relations used in tasks and this is done without changing the operations performed in the local plans. That is, the only transformation allowed is renaming of input relations. This restriction makes the global access plan produced by **D** easier to derive. Allowing more complex transformations on query plans in order to achieve even better utilization of temporary results is also possible and is described in the context of the heuristic algorithm of the following section.

Algorithm **D** proceeds as follows. First, as in **BS**, the queries that possibly overlap on some selections and joins are identified by checking the ground database relations that are used. For all queries $Q_i \in \mathbf{Q}$ that overlap with some other queries, we consider the corresponding plans P_i (local access plans) and define a directed graph $GP(V,E,L)$ (global access plan) in the following way

- $V = \bigcup_{i=1}^{n} V_i$
- $E = \bigcup_{i=1}^{n} E_i$
- For every $v_i \in V$, $L(v_i) = L_i(v_i)$

GP is in a sense the *union* of the local plans. We also define a function $Res : \mathbf{Q} \rightarrow V$ such that $Res(Q_i)=v_i$, where v_i is the node of plan P_i that provides the result to Q_i. Based on this graph, the decomposition algorithm performs some simple steps that introduce the effects of sharing among various tasks. The main idea is to avoid accessing the same data pages multiple times. Hence, the transformations that are done on the graph are based on changing the input relations to subqueries, to previously

computed temporary relations. Figure 4.6 illustrates the basis of our transformations. In the following figures *nemps* and *dept* are used in place of *num_of_emps* and *dept_name* respectively. The temporary relation TEMP1 created by subquery SQ_1 can be further restricted to give the result of subquery SQ_2 ($SQ_2 \Longrightarrow SQ_1$). Therefore, TEMP1 can be used as the input to that last subquery, instead of EMP. This is accomplished by adding a new edge from the node representing SQ_1 to the corresponding node for SQ_2. Also the relation name in SQ_2 is changed to TEMP1.

Formally algorithm **D** proceeds as follows. After building the graph GP, the following transformations are performed in the order they are presented

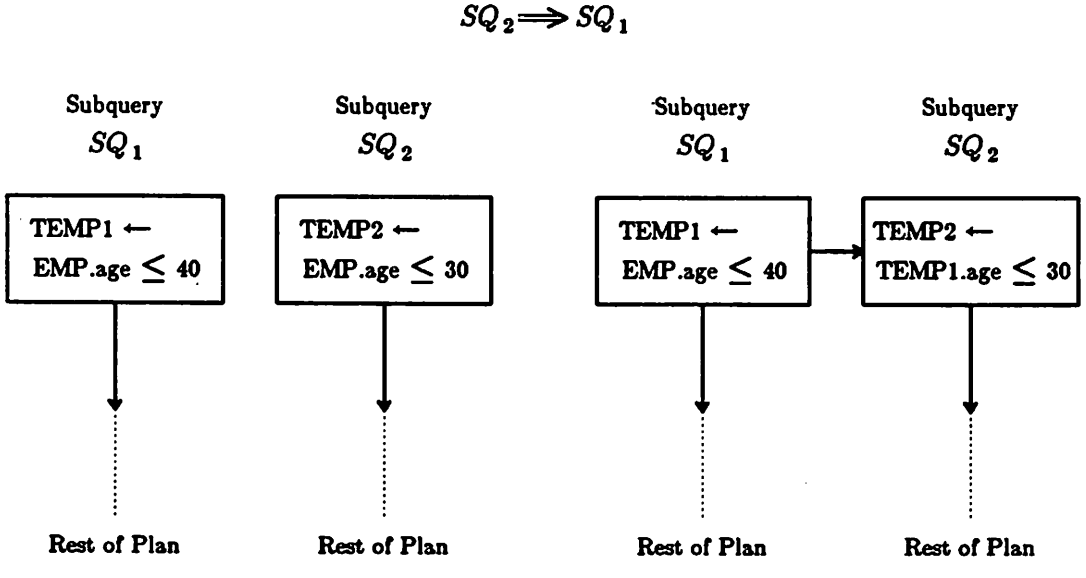


Figure 4.6: Basic Merge Operation

[1] *Proper Implications* : Let $PI(v_i) = \{v_j \mid L(v_i) \Longrightarrow L(v_j)$ and $L(v_j) \not\Longrightarrow L(v_i)\}$. For a given task v_i, $PI(v_i)$ gives the set of tasks v_j, the results of which can be used by v_i as inputs instead of other base relations. Let $c_i \in PI(v_i)$ be the task such that $\forall\, v_j \in PI(v_i)$, $L(c_i) \Longrightarrow L(v_j)$ (if more than one such task exists, let c_i be the one belonging to the plan P_k with the least k). In other words c_i is the *strongest* condition that can be performed on some input relation(s) so that the result of this condition can still be used to answer v_i. Then, replace the occurrences of base relations used in tasks v_i with the corresponding temporary relations $TEMP_k$ found in the tasks $c_i = [TEMP_k \leftarrow expr]$. This is accomplished by adding an edge $c_i \rightarrow v_i$ and changing $L(v_i)$ by substituting the relation name involved in the selection or join to the name of the temporary relation which holds the result in $L(c_i)$ (i.e. $TEMP_k$).

[2] *Identical Nodes* : In the case of nodes that produce identical temporary relations we use a simple step to compute that temporary relation result only once and then change relation names to the one selected to hold the result. First, the equivalence classes C_i are determined, each composed of nodes from V, such that for every $v_j, v_k \in C_i$, $L(v_j) \equiv L(v_k)$. Select the vertex v_j belonging to the plan P_j with the least index j as the representative c_i of class C_i. Then, for each equivalence class C_i, remove from the graph GP all nodes $v_j \in C_i - \{c_i\}$ and substitute each edge $v_j \rightarrow v_k$ with a new edge $c_i \rightarrow v_k$. Let $L(v_k) = [TEMP_k \leftarrow expr_k 1]$, for all such v_k.

Also let c_i=[$TEMP_i \leftarrow expr_i$]. Change all occurrences of relation name $TEMP_k$ in v_k to $TEMP_i$. Finally, if for some query Q_m, $v_j = Res(Q_m)$ and $v_j \in C_i$, set $Res(Q_m)$ to c_i. This last step makes sure that identical final results are never computed more than once.

[3] *Recursive Elimination* : Because steps [1] and [2] may have introduced new nodes that are now identical, we apply step [2] repeatedly until it fails to produce any further reduction to the graph GP. Example of such a case is a join performed on two relations that are restricted with identical selection clauses. Step [2] will merge each pair of identical selections to a single one and then in the next iteration the two join nodes will also be merged into a single node.

The result of the above transformation is a directed graph GP' which is guaranteed to be acyclic if the initial graphs P_i are acyclic. This is due to the fact that any transformation performed on the graph in all cases adds new edges that go always from less to more restrictive tasks. Therefore a cycle is not possible, for it would introduce a chain of proper implications of the form $v_1 \Longrightarrow v_2 \Longrightarrow \cdots \Longrightarrow v_1$. Finally, using the directed arcs of GP' a partial order on the execution of the various tasks can be imposed. That is the global access plan that algorithm D suggests. The function Res also gives the nodes that hold the results for all queries.

To give an example of the algorithm, Figures 4.7, 4.8 and 4.9 show the initial access plan graphs, the graph GP after transformation [1] and the final global access plan graph (as a sequence of operations) respectively for the two queries Q_1 and Q_2 of

section 3.

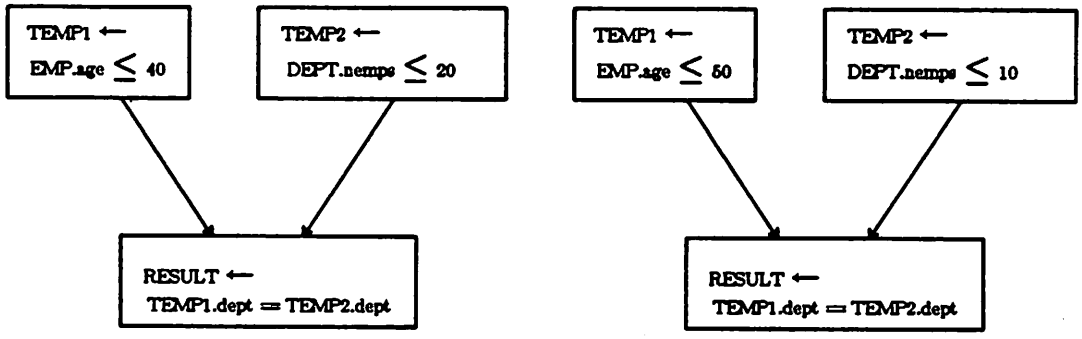


Figure 4.7: Initial Global Access Plan

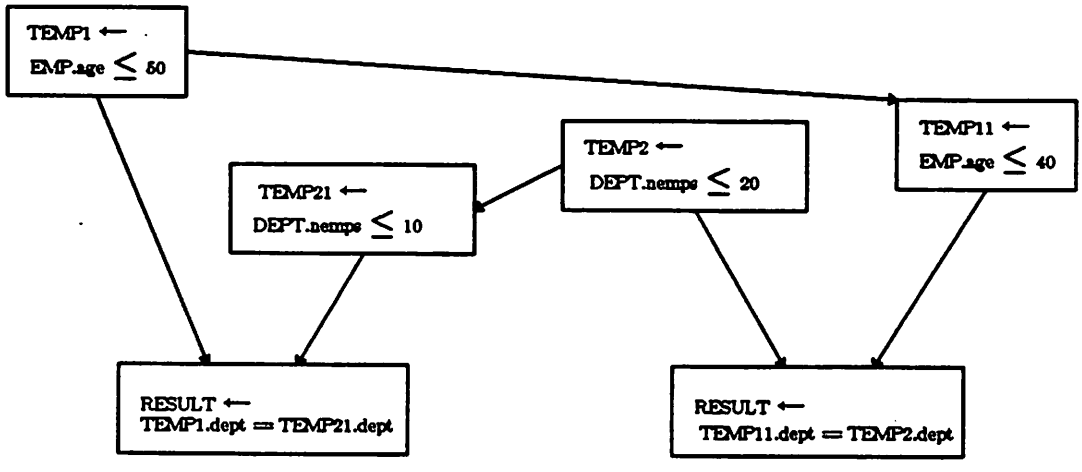


Figure 4.8: Global Access Plan after Transformation [1]

```
retrieve into TEMP1 (EMP.all)
   where EMP.age ≤ 50

retrieve into TEMP2 (DEPT.all)
   where DEPT.num_of_emps ≤ 20

retrieve into TEMP11 (TEMP1.all)
   where TEMP1.age ≤ 40

retrieve into TEMP21 (TEMP2.all)
```

```
   where TEMP2.num_of_emps ≤ 10
retrieve (TEMP11.all,TEMP2.all)
   where TEMP11.dept_name = TEMP2.dept_name
retrieve (TEMP1.all,TEMP21.all)
   where TEMP1.dept_name = TEMP21.dept_name
```

Figure 4.9: Final Global Access Plan

Estimating the cost of the global plan imposed by the graph GP', we have

$$Cost(GP') = \sum_{i=1}^{n} Bestcost(Q_i) - \sum_{s \in CS} n_s \cdot savings(s)$$

where CS is now the set of *all* common subexpressions found in the local access plans and n_s and $savings(s)$ are defined in the same way as in the previous section. For example, for the queries Q_1 and Q_2, $Cost(GP') = 223 + C_j(20,5) + C_j(40,3)$, where $C_j(a,b)$ is the cost function for a join between two relations as introduced in the previous section. This cost represents a savings of 65 page accesses compared to an arbitrary serial execution. Concerning the complexity of the algorithm, it can be observed that steps [1] and [2] of the above algorithm require time in the order of $\prod_{i=1}^{k} | V_i |$, where k is the number of queries represented by their representative plans in graph GP and V_i is the set of vertices for plans P_i, $1 \leq i \leq k$. The number of times N step [2] is executed as a result of the recursive elimination of common subgraphs, generally depends on the size of common subexpressions and in the worst case is the depth of the longest query plan. The total time required by the algorithm is therefore in the order of $N \cdot \prod_{i=1}^{k} | V_i |$.

We now move on to discuss the most general algorithm that can be used to process multiple queries. As mentioned in the beginning of this section, the heuristic algorithm to be described also captures more general transformations than the ones allowed here (simple relation name change).

4.7. Heuristic Algorithm

As it was illustrated through an example in section 4, merging locally optimal plans to produce the global access plan is not always the optimal strategy. The main reason is that there are more than one possible plans to process a query, yet the algorithms presented in the previous sections consider only one of them, i.e. the optimal in terms of execution time. Using suboptimal plans may prove to be better. Grant and Minker in [GRAN80] present a Branch and Bound algorithm [RICH83] that uses more than locally optimal plans. One assumption they make is that queries involve only equijoins while all selections are of the form $R.A=cons$. This section presents a similar algorithm which is defined as a state space search algorithm (A* [RICH83]) with better average case performance than the one of [GRAN80]. To simplify the presentation of the algorithm we will also make here the assumption that all queries have equality predicates. At the end of the section extensions that can be made to include more general predicates in queries are discussed.

As shown in Figure 4.1, the Global Optimizer receives as input a set of queries $\mathbf{Q}=\{Q_1,Q_2,\cdots,Q_n\}$. Then for each query Q_i a set of possible plans that can be used to process that query is derived. Let that set be $\mathbf{S}_i=\{P_{i1}, P_{i2}, ..., P_{ik_i}\}$. For a given query Q_i, $\mathbf{S}_i$ contains the optimal plan to process Q_i along with all other possi-

ble plans that share tasks with plans for other queries. For example, for the two queries Q_3 and Q_4 of section 3, in addition to the plans P_3 and P_4 presented there the plan

(P_{32})
```
retrieve into TEMP1 (EMP.all,DEPT.all)
    where EMP.dept_name = DEPT.dept_name
retrieve (JOB.all,TEMP1.all)
    where JOB.job = TEMP1.job
```

should also be considered for query Q_3 because it shares the join EMP.dept_name=DEPT.dept_name with P_4. Hence, the sets of plans S_3 and S_4 will be $S_3=\{P_3,P_{32}\}$ and $S_4=\{P_4\}$. Generally, this algorithm considers optimizing a set of queries instead of a set of plans, which was the case with algorithms **BS** and **D**. Considering more than one candidate plans per query has the desirable effect of detecting and using effectively all common subexpressions found among the queries.

This section is organized as follows: in the first subsection a state space is defined and an A* algorithm that finds the solution by searching that space is described. Then subsection 4.7.2 presents a preprocessing step that can be applied in order to improve the average case performance of the algorithm. Finally, the last subsection discusses the performance of the algorithm and suggests some possible extensions.

4.7.1. The Heuristic Algorithm

In order to present an A* algorithm, one needs to define a state space $\mathbb{S}$, the way transitions are done between states and the costs of those transitions.

Definition 1 : A *state* s is an n-tuple $<p_1,p_2,\ldots,p_n>$, where $p_i \in \{\text{NULL}\} \cup S_i$. If p_i = NULL it is assumed that state s suggests no plan for evaluating query Q_i.

Definition 2 : Let $s_1=<p_1,p_2,\ldots,p_n>$ and a function $next : \mathbb{S} \rightarrow \mathbb{Z}$ with

$$next(s_1) = \min\{j \mid p_j=\text{NULL}\} \quad \text{if } \{j \mid p_j=\text{NULL}\} \neq \emptyset$$

A *transition* $T(s_1,s_2)$ from state s_1 to s_2 exists iff s_1 has at least one NULL entry and $s_2=<q_1,q_2,\ldots,q_n>$, with $q_i=p_i$ for $1 \leq i < next(s_1)$, $q_{next(s_1)} \in S_{next(s_1)}$ and q_j=NULL, for $next(s_1)+1 \leq j \leq n$.

Definition 3 : The *cost* $tcost(t)$ of a transition $t=T(s_1,s_2)$ is defined as the *additional* cost needed to process the new plan q_m introduced at t (according to Definition 2), given the (intermediate or final) results of processing the plans of s_1.

From the above definition it can be seen that the way transitions are defined, the first NULL entry of a state vector, say at position i, will always be replaced by a plan for the corresponding query Q_i. Finally, we define the initial and final states for the algorithm. The state $s_0=<\text{NULL,NULL},\ldots,\text{NULL}>$ is the initial state of the algorithm and the states $s_f=<p_1,p_2,\ldots,p_n>$ with $p_i \neq$NULL, for all i, are the final states.

The A* algorithm starts from the initial state s_0 and finds a final state s_F such that the cost of getting from s_0 to s_F is minimal among all paths leading from s_0 to any final state. The cost of such a path is the total cost required for processing all n

queries. For brevity it will be assumed that each plan is an unordered set of tasks instead of a directed graph. In order for an A* algorithm to have fast convergence, a heuristic function h is introduced on states [RICH83]. This function is used to prune down the size of the search space that will be explored. Such a function $h : \mathbb{S} \rightarrow \mathbb{Z}$ was introduced in [GRAN80] in the following way : let $s = <p_1, p_2, \ldots, p_n>$ be some state. Then

$$h(s) = \sum_{i=next(s)}^{n} \min_{j} \left[est\_cost(P_{ij}) - \sum_{t} n_t \cdot est\_cost(t) \right]$$

where t are common tasks found in plans already in s and n_t is the number of times task t appears in these plans. The function *est_cost* is defined on tasks as follows

$$est\_cost(t) = \frac{cost(t)}{n_q}$$

where n_q is the number of queries the task t occurs in. The idea behind defining such a function is that the cost of a task is amortized among the various queries that will *probably* make use of it. For a plan p, it is assumed that

$$est\_cost(p) = \sum_{t \in p} est\_cost(t)$$

If it is true that $est\_cost(p) \leq Cost(p)$ then the convergence of the A* algorithm is guaranteed [RICH83]. Therefore, one significant issue is to define a correct function *est_cost*, "correct" meaning that it underestimates the actual cost. Let us give an example, also drawn from [GRAN80], which will motivate the discussion of the following subsection.

Two queries Q_1 and Q_2 are given along with their plans : P_{11}, P_{12}, P_{21}, P_{22}, P_{23}. We will use t_{ij}^k to indicate the $k-$th task of plan P_{ij}. The table below gives the costs for the tasks involved in each plan

Plan	Task	Cost	Task	Cost	Task	Cost	Total
P_{11}	t_{11}^1	40	t_{11}^2	30	t_{11}^3	5	75
P_{12}	t_{12}^1	35	t_{12}^2	20			55
P_{21}	t_{21}^1	40	t_{21}^2	10	t_{21}^3	5	55
P_{22}	t_{22}^1	10	t_{22}^2	30	t_{22}^3	10	50
P_{23}	t_{23}^1	30	t_{23}^2	20			50

and the identical tasks are

$t_{11}^1 \equiv t_{21}^1$; $t_{11}^2 \equiv t_{22}^2$; $t_{12}^2 \equiv t_{23}^2$;

Given the actual task costs and the sets of identical tasks, the estimated costs (*est_cost*) for these tasks are

Task	t_{11}^1	t_{11}^2	t_{11}^3	t_{12}^1	t_{12}^2	t_{21}^2	t_{21}^3	t_{22}^1	t_{22}^3	t_{23}^1
Estimated Cost	20	15	5	35	10	10	5	10	10	30

and the estimated costs for the plans are,

Plan	P_{11}	P_{12}	P_{21}	P_{22}	P_{23}
Coalesced Cost	40	45	35	35	40

Based on the above numbers and the construction procedure outlined, Figure 4.10

shows the search space $\mathbb{S}$ along with the costs of transitions between states and estimated costs of going from intermediate to final states.

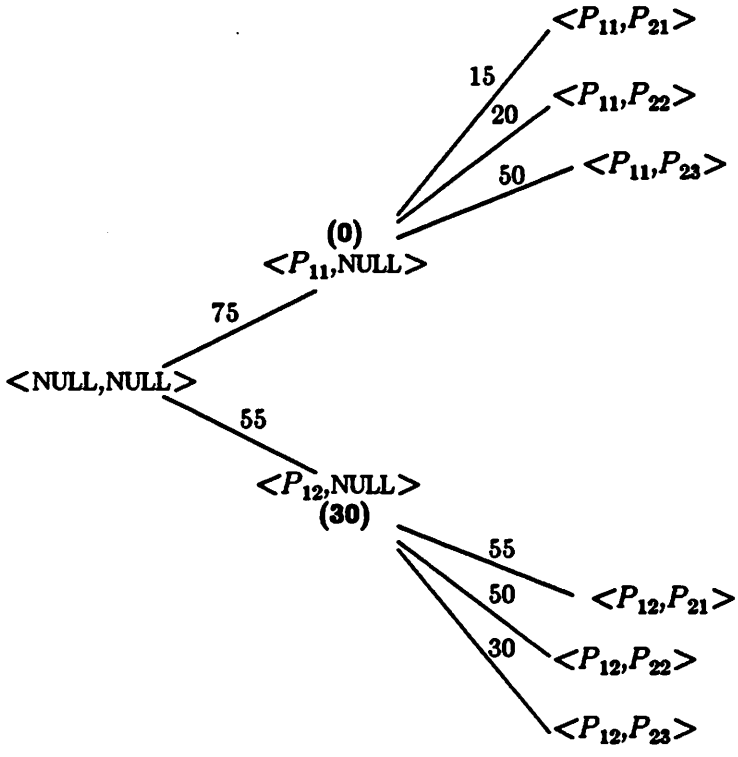


Figure 4.10: Example Search Space for A* Algorithm
(numbers in parentheses show estimated costs)

Tracing the A* algorithm we get

$s_0 =$ <NULL,NULL> /* *expand state* s_0 */
$s_1 =$ <P_{11},NULL> /* *expand state* s_1 */
$s_2 =$ <P_{21},NULL> /* *expand state* s_2 */
$s_F =$ <P_{12},P_{23}> /* *the final solution* */

yielding $<P_{12}, P_{23}>$ as the best solution. Notice that with this set of estimators the algorithm exhaustively searches all possible paths in the state space. It is exactly this

bad behaviour of the algorithm that we will try to improve by examining more closely the relationships among various tasks. For example, in the case presented above, it is clear right from the beginning that plan P_{11} will not be able to share both of its tasks t_{11}^1 and t_{11}^2 with plans P_{21} and P_{22} respectively, since only one of these two latter plans will be in the final solution (final state). Therefore, the value $est\_cost(P_{11})$ is less than what could be predicted after looking more carefully at the query plans. It is a known theorem in the case of A* algorithms, that the higher the estimator values the faster the convergence [RICH83]. Hence, estimating the cost function better will enable the algorithm to converge faster to the final solution.

4.7.2. The Modified Algorithm

The goal of this subsection is to describe a preprocessing phase which provides a way to compute a better cost estimation function. Suppose that n sets of plans $S_1, S_2, ..., S_n$ are given, with $S_i=\{P_{i1}, P_{i2}, ..., P_{ik_i}\}$. Assume also that the pairs of tasks $t_i \in P_{il}$ and $t_j \in P_{jm}$ such that $t_i \equiv t_j$ are known. We then define a directed graph $G(V,E)$ in the following way

- For each plan P_{ij} that has a task t_{ij}^k identical to task(s) used for evaluating other than the i-th query, introduce a vertex v_{ij}
- For each pair $t_{kl}^i \in P_{kl}$, $t_{pq}^j \in P_{pq}$ of such identical tasks there is an edge connecting the two vertices $(v_{kl} \rightarrow v_{pq})$ if there is no other plan P_{pr} with a task t_{pr}^s such that $t_{pr}^s \equiv t_{pq}^j$

Given the above definition a unique graph can be built based on a set of plans and a set of identities among tasks. Notice that not *all* plans are needed to build the graph. Only those having identical tasks among them are considered. Also, there may be more than one directed edge ($v_{ij} \rightarrow v_{kl}$) going from v_{ij} to v_{kl} if there are more than one pair of identical tasks involved in plans P_{ij} and P_{kl}. In order to reduce the size of the graph, only one edge $v_{ij} \rightarrow v_{kl}$ is recorded for any two vertices v_{ij} and v_{kl} that have at least one edge between them. No information is lost that way. The number of identical tasks found between the two plans is of no importance.

The goal of the preprocessing phase is to find plans that are most probably not sharing their tasks with other plans. The algorithm used is a slightly modified Depth-First-Search (DFS) algorithm. The difference is that in the course of backing up to the vertex v_{ij} from which another vertex v_{kl} was reached using the edge $v_{ij} \rightarrow v_{kl}$, the identification (subscript) kl is stored in some set associated with vertex v_{ij}. Call that set the *Need* set of vertex v_{ij}. Then, at the end of the algorithm, delete from G all vertices that have two or more members $k'l'$ and kl in their *Need* sets, such that $k'=k$. Along with the vertex, its edges (both out- and in-going) are also marked as OUT. This deletion process is continued by deleting vertices that have at least one out-going edge marked OUT. The edge and vertex elimination process stops when no more deletions are possible. Call the final graph $G'(V',E')$ and let $\mathbf{S}'$ be the set of plans P_{ij} that have a corresponding vertex v_{ij} in G'.

What is achieved through that preprocessing phase, is to reduce considerably the size of the search space for the A* algorithm. Only plans in $\mathbf{S}'$ are considered in order to derive the *est_cost* values. To give an example of the preprocessing phase along

with a run of the A* algorithm, we will redo the example of the previous subsection.

We are given again the same two queries and five plans : P_{11}, P_{12}, P_{21}, P_{22}, P_{23}. The graph of Figure 4.11 gives the graph G for the set of plans given.

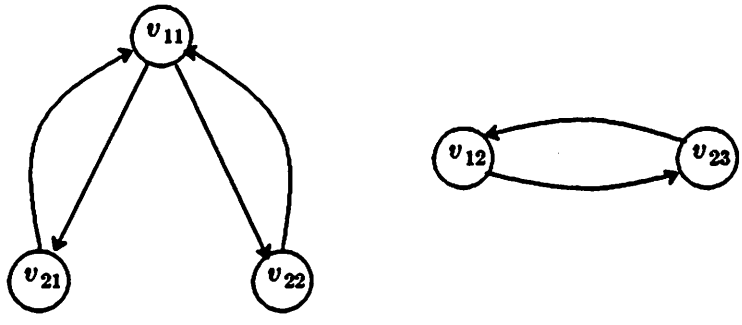


Figure 4.11: Graph G for Queries Q_1 and Q_2

After the DFS s performed the *Need* sets for the various vertices, will be

Vertex	*Need*
v_{11}	{11,21,22}
v_{12}	{12,23}
v_{21}	{11,21}
v_{22}	{11,22}
v_{23}	{12,23}

From the above table it can be seen that vertex v_{11} must be eliminated since it can reach both 21 and 22 through directed paths. After that, the edges $(v_{11} \rightarrow v_{21})$, $(v_{21} \rightarrow v_{11})$, $(v_{11} \rightarrow v_{22})$ and $(v_{22} \rightarrow v_{11})$ are marked as OUT. This causes vertices v_{21} and v_{22} to be deleted also. Finally, we see that no more vertices can be deleted. The

remaining graph is shown in Figure 4.12.

Figure 4.12: Final Graph G'

Finally, $\mathbf{S}' = \{P_{12}, P_{23}\}$.

Using the result of the preprocessing phase, we next compute the new estimated costs for tasks and plans. First, based on the cost function *cost* defined for tasks, the following function *coalesced_cost* on tasks t [GRAN80] (which is identical to the estimator used in the previous subsection) is defined

$$coalesced\_cost(t) = \frac{cost(t)}{n_q}$$

where n_q is the number of queries this task occurs in, and for plans

$$coalesced\_cost(P_{ij}) = \sum_{t \in P_{ij}} coalesced\_cost(t)$$

Now, given a plan P_{ij} and a specific task t_{ij}^k, let Q_{ij} be the set of, other than i, queries q that have common tasks with P_{ij}. Also, let n_{ij}^q be the number of plans P_{qr} that correspond to query q in Q_{ij}. Then, *est_cost* is defined as follows

a) If the plan P_{ij} is not in $\mathbf{S}'$ and $n_{ij}^q > 1$ for at least one query q, then

$$est\_cost(P_{ij}) = Cost(P_{ij}) - \sum_{q \in Q_{ij}} \max\left[coalesced\_cost(t_{ij}^k)\right]$$

where $t_{ij}^k \equiv t_{qr}^s$, for some r and s.

b) If the plan is in S' or it is not in S' but the above condition on n_{ij}^{l} is not true, then

$$est\_cost(P_{ij}) = coalesced\_cost(P_{ij})$$

Finally, we show how to compute the function $h(s)$. First, define

$$add\_cost(i) = \min_{j} \left[est\_cost(P_{ij}) - \sum_{t \in O_i \cap P_{ij}} n_t \cdot est\_cost(t) \right]$$

where $O_i = \bigcup_{l=1}^{i-1} P_{lk}$ and P_{lk} is the plan that provides, for a given l, the above minimum value in the computation of $add\_cost(l)$. Also, t are common tasks that belong to plans already in the state s and n_t is the number of times task t appears in these plans. Then, define

$$h(s) = \sum_{i=next(s)}^{n} add\_cost(i)$$

The A* algorithm can then be applied using these new estimators. For example, processing the two queries Q_1 and Q_2 given above, the following are the computed estimated costs for the plans

Plan	P_{11}	P_{12}	P_{21}	P_{22}	P_{23}
Estimated Cost	55	45	35	35	40

Tracing the A* algorithm, we see that it explores the following states

s_0 = <NULL,NULL> /* *expand state* s_0 */

s_1 = <P_{12},NULL> /* *expand state* s_1 */

$s_F = <P_{12}, P_{23}>$ /* *the final solution* */

yielding again $<P_{12}, P_{23}>$ as the optimal solution with cost 85. Notice that if the commands were executed sequentially it would have costed $Cost(P_{12}) + Cost(P_{23}) = 105$. Therefore, a total savings of 19% was achieved using the global optimization algorithm. Moreover, compared to the trace of the previous subsection, it can be seen that exhaustive search is avoided because of the high cost estimates for some paths.

Summarizing, the final algorithm is the following

ALGORITHM HA

[1]. Build graph G and apply the preprocessing DFS algorithm

[2]. For all queries with no representative plan in the initial graph G, find the originally cheapest plan and put it in the final solution

[3]. Based on the result set S', compute the function *est_cost*

[4]. For the rest of the queries run the A* algorithm described in the previous subsection

4.7.3. Discussion and Extensions

The global access plan is derived from integrating the local plans found in the final state s_F returned by the A* algorithm. The integrating process is very similar to the one described for the decomposition algorithm where local plan graphs are merged together. Examining the estimated cost of the global access plan, we have

$$Cost(GP) = \sum_{p \in GP} Cost(p) - \sum_{s \in CS} n_s \cdot savings(s)$$

where CS represents the total number of subexpressions found in the *n queries* (not *plans* as it was the case in algorithm **D**) and n_s and $savings(s)$ are defined in section 4.5. Regarding the complexity of the algorithm **HA** we must notice that it is very hard to analyze the behaviour of an A* algorithm and give a very good estimate on the time required. In the worst case of course it may require time exponential on the number of queries but on the average the complexity depends on how close the cost estimation function is to the actual cost. However, the A* algorithm with the new estimator function we proposed will not take more steps than the original A* algorithm presented in subsection 4.7.2 (which uses *coalesced_cost* as its estimator function). This is based on the fact that for any task t it is true that $est\_cost(t) \geq coalesced\_cost(t)$. Therefore with the help of a known theorem [RICH83] our algorithm will give a solution in at most the same number of steps as the algorithm of [GRAN80].

Finally, note that the algorithm described is correct only in the cases where queries use solely equijoins and equality selection clauses. If arbitrary selection clauses are used, the A* algorithm presented above will not find the optimal solution. This is true because the imposed order in which the state vectors are filled (i.e. in ascending query index) may not result to the best utilization of common subexpression results. As an example, consider two queries Q_1 and Q_2, such that Q_1 has a more restrictive selection than Q_2. Then clearly, it would be better to consider executing Q_2 first since in that case the result of Q_2 can be used to answer Q_1, the opposite being impossible.

This problem with the heuristic algorithm can be easily fixed by changing the transitions to fill not the next available NULL slot in a state s, as it was before done through the use of $next(s)$, but rather any available (NULL) position of s. This results to larger fanout for each state and clearly more processing for the A* algorithm. The heuristic cost function *est_cost* is defined similarly with the difference that in addition to identical tasks, pairs of tasks t_i and t_j such that $t_i \Longrightarrow t_j$ and $t_j \not\Longrightarrow t_i$ must be considered as well.

4.8. Some Experimental Results

We expect that for a large number of applications and query environments global query optimization will offer substantial improvement to the performance of the system. In a series of experiments, the algorithms of the previous sections have been simulated using EQUEL/C [RTI84] and the version of INGRES that is commercially available. The experiments were run over the set of queries that Finkelstein used in [FINK82]. The database schema used was modeling a world of employees, corporations and schools that the employees have attended, the relations being `Employees`, `Corporations` and `Schools` respectively. All eight queries along with a brief description of the data they return are shown in Appendix A. Seven different sets of queries `QSET1-QSET7` where chosen and the queries within each of these sets were processed

a) as independent queries

b) as the Better Serial Execution Algorithm suggests

c) as the Decomposition Algorithm suggests, and finally

d) as the Heuristic Algorithm suggests.

Table 4.1 describes some characteristics of the sets QSET1 to QSET7.

Query Set	Number of Queries	Queries	BS	D	HA
QSET1	2	{1,7}	X		
QSET2	2	{1,6}	X		
QSET3	4	{1,2,6,7}	X		
QSET4	2	{6,7}	X		
QSET5	4	{2,3,4,6}	X	X	
QSET6	7	{1,2,3,4,5,6,7}	X	X	
QSET7	2	{7,8}	X	X	X

Table 4.1: Query Sets Used in Experiments

The second column indicates the number of queries used in each set while the third column shows which queries from Appendix A were specifically used. The rest three columns indicate which algorithms were applicable and gave distinct access plans to each of the given query sets. The reason that some query sets do not have an entry in some of these columns is that not all algorithms gave distinct global access plans. For example, in section 4.5.2 it was shown that if the query graph QG is acyclic, algorithms **BS** and **D** will produce identical plans.

The above sets of queries were tested in various settings. First, unstructured relations were used with their sizes varied according to Table 4.2.

Relation	Number of tuples
Employees	100 - 10,000
Corporations	10 - 500
Schools	20 *(fixed)*

Table 4.2: Sizes of relations

Second, the same experiments were performed with structured relations. Specifically, the following structures were used

```
isam secondary index on Employees(experience)
isam primary structure on Corporations(earnings)
hash primary structure on Schools(sname)
```

Finally, in a another series of experiments the given queries were slightly modified by changing the constants used in one-variable selection clauses. The goal was to introduce higher sharing among the queries. Higher sharing is achieved when more queries can take advantage of the same temporary result. As it was indicated in section 4.5.2, the formula that provided an estimate on the cost savings using a global optimization algorithm is (for n queries Q_1, ..., Q_n)

$$\sum_{i=1}^{n} Bestcost(Q_i) - \sum_{s \in CS} n_s \cdot savings(s)$$

where CS is the set of common temporary results s and n_s is the number of queries using the same temporary result s. Therefore, higher cost reduction is achieved if more queries can use the same temporary result. By changing the constants in the

qualification of the queries it was possible to check how n_s affected the cost of processing the global access plans.

The measure used in this performance analysis was

$$PERCI = \frac{Cost_1(I/O) - Cost_2(I/O)}{Cost_1(I/O)} \qquad (F)$$

where $Cost_1(I/O)$ is the number of I/O's required to process all queries assuming no global optimization is performed. $Cost_2(I/O)$ is the corresponding figure in the case where a global access plan is constructed according to some of the presented optimization algorithms. The analogous CPU measure was also recorded; however, the numbers were almost the same and will not be shown. In the following, the results of the experiments are described in detail.

4.8.1. Unstructured Relations

As indicated in Table 4.1, some query sets were processed using only one or two of the algorithms. Because of the similarity of the results the diagrams will be grouped according to the algorithm used for optimization. Hence, three diagrams are presented. One for query sets QSET1, QSET2 and QSET3, one for QSET4, QSET5 and QSET6 and another for QSET7. The first group was optimized using only **BS** because **D** and **HA** were not applicable. The second group was optimized using **BS** and **D** while for the last group all three algorithms were used. Figures 4.13, 4.14 and 4.15 illustrate how *PERCI* varies for the three above mentioned groups according to the size of the database in the case of unstructured relations. Also, Figure 4.16 gives the overall average improvement in the performance of the system for all query sets.

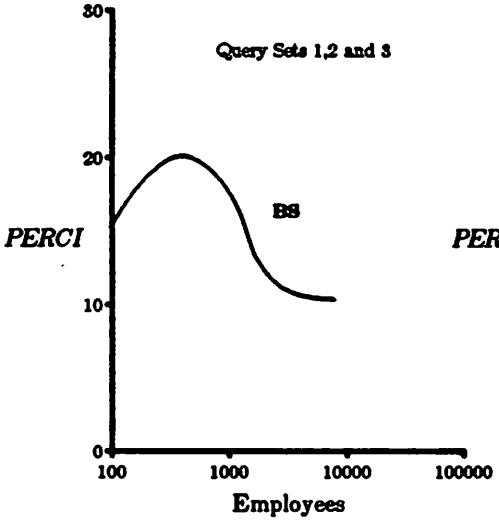


Figure 4.13

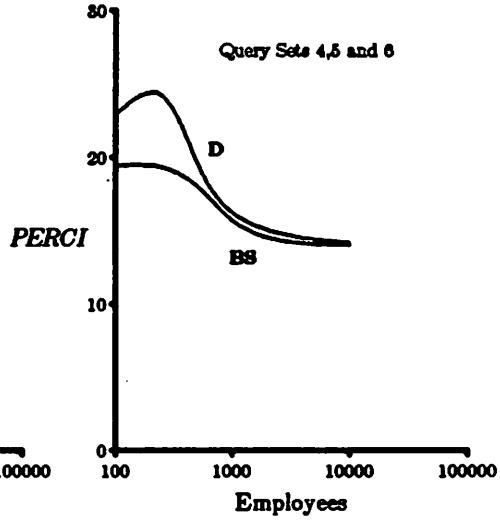


Figure 4.14

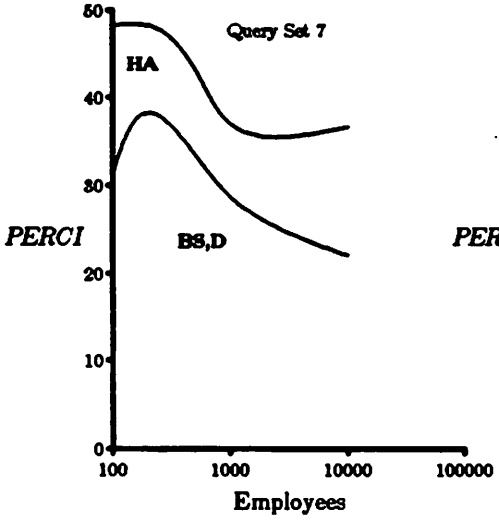


Figure 4.15

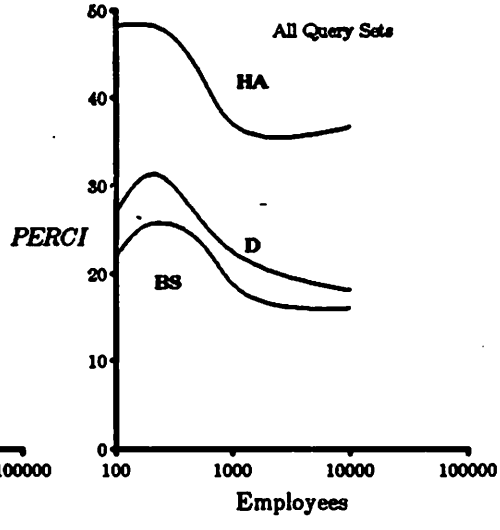


Figure 4.16

Performance Improvement for Unstructured Relations

The size of the database is represented by the size of the Employees relation. The reasons for choosing that relation was first that all queries were using Employees

(compared to Corporations or Schools) and second the fact that the diagrams are similar for the Corporations relation as well.

Some comments can be made here for these diagrams. First, it is clear that there is always a gain in performance by doing global query optimization, i.e. $PERCI \geq 0$ in all the above figures. Second, after some size of the relations, *PERCI* starts to decrease. This was due to the specific type of queries used. In particular, because of queries involving joins, the denominator of the formula (F) grows faster than the numerator. In the given queries, the selection clauses were responsible for the savings in the numerator. That savings increases with rate proportional to the factor by which a relation is reduced as a result of performing a restriction on it (i.e. $1-S$, where S is the selectivity of the selection clause). On the other hand, if joins are included in the queries, $Cost_1(I/O)$ increases with a rate which depends on the cost of the join operation. It turns out that for small sizes of the relations the latter factor is less than the former while after some size this relationship is reversed. Hence, the slight increase followed by a decrease in the values of *PERCI* indicated in the above diagrams.

The diagrams also show that there was no significant difference between the improvements achieved by the **BS** and **D** algorithms. In order to have a difference in the global plans generated by the two algorithms, as discussed in section 4.6, cycles must occur in the query graph QG. Even in that case though, the difference may not be significant depending on the sizes of the temporary results. In the experiments ran, the temporary relations not shared by more than one queries in the global access plan constructed by **BS** but shared in the corresponding plan generated by **D**, were rather

small. Hence, sharing of these relations contributed only marginally to the performance improvement. Finally, for the last query set QSET7, the plan generated by **HA** was significantly better than the one generated by **BS** (or **D** since these are the same for QSET7). By allowing the result of the join

```
e.employer = c.cname
```

to be shared by both queries 7 and 8, significantly better performance was achieved.

4.8.1. Structured Relations

The same set of experiments was run over a structured database. Relations were indexed as mentioned in the beginning of this section. The reason for doing these experiments was to check if the overhead of accessing a relation through a secondary structure might be higher than the overhead of accessing an unstructured intermediate result. For example, suppose that retrieving the part of a relation that satisfies a simple one-variable restriction requires 10 page accesses. That includes the cost of searching first the index table and then accessing the data pages. Suppose now that there is an intermediate result, produced by some other query, that can be used to answer the same restriction clause. If the size of that intermediate result is less than 10 pages then it will be more efficient to process the restriction by scanning the unstructured temporary result than going through the index table.

Figures 4.17, 4.18 and 4.19 illustrate how *PERCI* varies for the three above mentioned groups according to the size of the database in the case of structured relations. Also, Figure 4.20 gives again the overall average improvement in the performance of the system for all query sets.

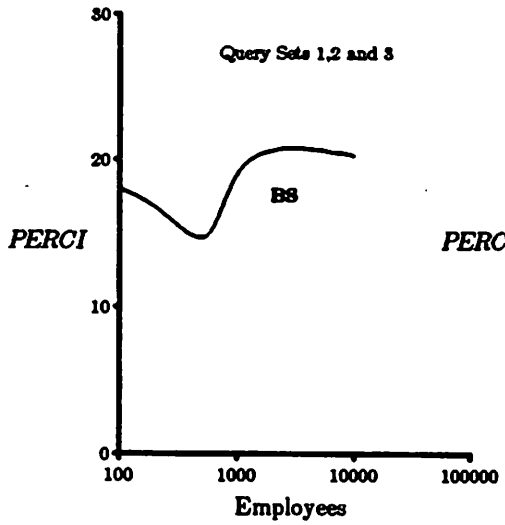


Figure 4.17

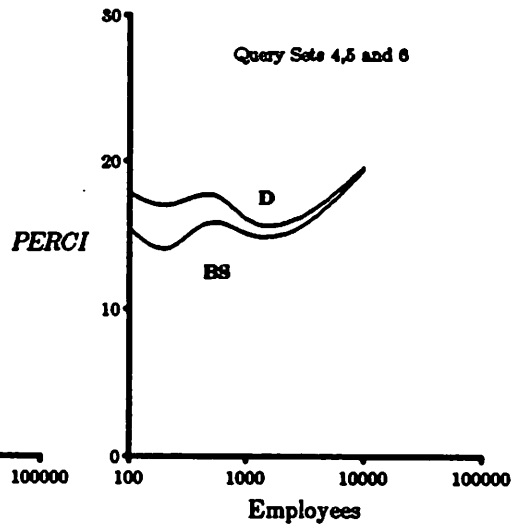


Figure 4.18

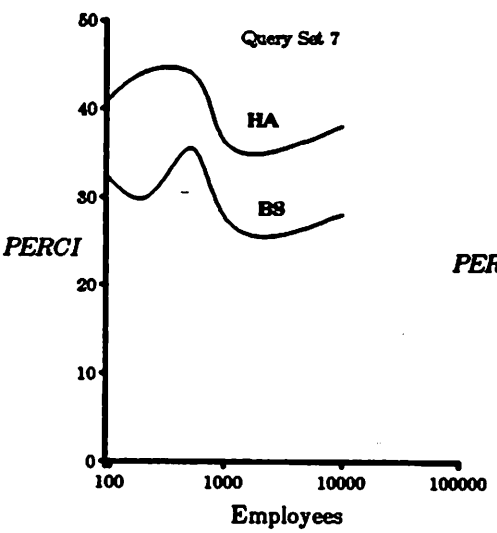


Figure 4.19

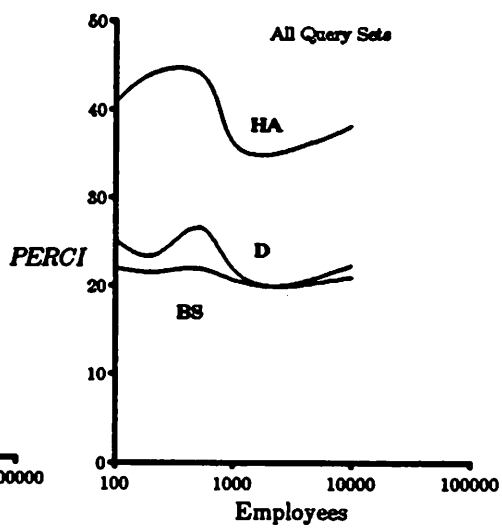


Figure 4.20

Performance Improvement for Structured Relations

Comparing the values of *PERCI* with the corresponding ones of the previous subsection, some decrease of 10-20% can be observed for all three algorithms depending on

the size of the involved relations. This was expected since using indexes reduces $Cost_1(I/O)$. However, after some size of Employee, *PERCI* starts increasing instead of decreasing, which was the case in the experiments of the previous subsection. This behaviour is due to the fact mentioned above, i.e. the overhead involved in using an index to access a relation. Moreover, the above effect is more obvious in cases where the involved relations are large. Then the size of the secondary indexes is in many cases significantly larger than the sizes of temporary results. Notice also that for small sizes of the relation Employee *PERCI* is decreasing. That was expected because for small relations temporary results grow faster in size than the index tables. Finally, notice that the relative performance of the three algorithms is not affected by the existence of indexes, i.e. **HA** still performs better than the other two and **D** provides better plans than **BS**.

4.8.2. Higher Sharing

In this last experiment, the given query sets were run over the same database with a modification in the queries so that higher degree of sharing is possible. That effect was introduced by changing the restrictions `experience` ≥ 20 found in queries 2,4,5 and 7 to `experience` ≥ 10. This way the same temporary result could be used in the evaluation of more queries, compared to the ones in the experiments of the previous two subsections. Figure 4.21 illustrates how *PERCI* varied with the size of the database in the case of unstructured relations and for the second group of query sets (i.e. QSET4, QSET5 and QSET6). The rest of the query sets were not affected by this modification in the selection clauses in the sense that no increase in sharing was possi-

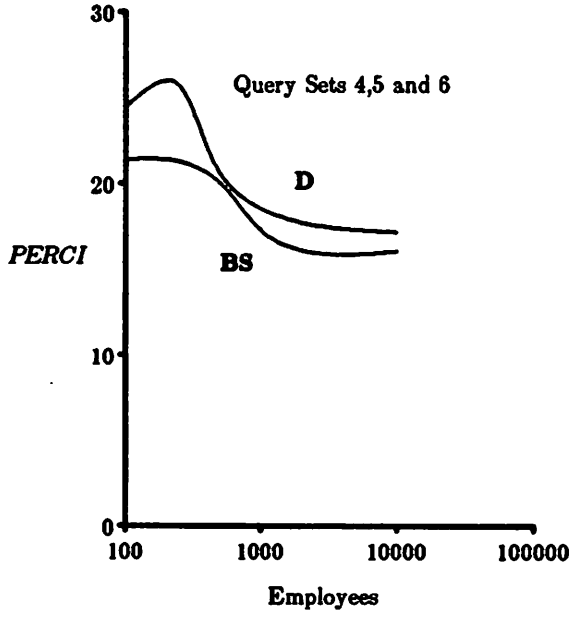


Figure 4.21: Performance Improvement for Higher Sharing

ble. Notice that the curve is similar to the one of Figure 4.14. However, because of the higher degree of sharing among queries an increase of about 10% in the performance improvement was observed.

4.9. Summary

This chapter presented a set of algorithms that can be used for multiple query processing. The main motivation for doing such interquery analysis is the fact that common intermediate results may be shared among various queries. We showed that various algorithms can be used for global query optimization. These algorithms were presented as parts of an algorithm hierarchy; descending the hierarchy more sophisticated algorithms can be used that give better access plans at the expense of increased complexity of the algorithm itself.

Some of the algorithms proposed were based simply on the idea of reusing temporary results from the execution of queries, where the processing of each individual query is based on a locally optimal plan. Using plans instead of queries enabled us to concentrate on the problem of using efficiently common results rather isolating common subexpressions. The last (heuristic search) algorithm, is a variation of the algorithm for optimizing a set of relational expressions originally proposed by Grant and Minker in [GRAN80]. The preprocessing phase added to the algorithm intends to derive a better cost estimator function used in the A* algorithm.

It is expected that for a large number of applications and query environments global query optimization will offer substantial improvement to the performance of the system. In a series of experiments, we have simulated these algorithms and checked the performance of the resulting global access plans under various database sizes and physical designs. This enabled us to check the usefulness of these algorithms even in the presence of fast access paths for relations. The results were very encouraging and showed a decrease of at least 20-50% in both I/O and CPU time. It should also be mentioned that the methods proposed do not pose any problems to the concurrency control and recovery modules. Since the given set of queries is thought as a transaction itself, changing the way processing is done has no effect on the system. The transaction boundaries are preserved. In terms of concurrent access, it should also be clear that our transformations do not affect the degree of concurrency. The data that each query processes is exactly the same as in any arbitrary serial execution of the queries. Hence, the size of the data sets that each query competes for neither increases nor decreases.

CHAPTER 5

CONCLUSIONS AND FUTURE DIRECTIONS

5.1. Summary of Thesis

The goal of this thesis was to develop techniques that can be used to improve the performance of extended relational database management systems. We summarize the results of this investigation in this last chapter.

In Chapter 1 we discussed various approaches that can be taken in developing systems to support non-business applications. Because the volume of data that these applications handle is constantly increasing, main memory may become insufficient. Secondary storage is then used and data managers are employed to efficiently store and access the data. We have argued that extending DBMSs with new features is the most adequate solution to the problem of supporting large databases used even by non-business applications. The remaining chapters where then used to propose such extensions to INGRES and discuss query processing issues.

Chapter 2 started by describing the language QUEL+ [STON85], an extension to QUEL. The most interesting new feature introduced to QUEL is allowing queries to be stored in QUEL fields, thus incorporating procedures as database objects [STON85]. Then, an extended decomposition algorithm based on the INGRES query processing algorithm was proposed. The extensions made were mainly due to the fact that one new operation was introduced, namely the materialization of QUEL fields.

We showed how a general algorithm can be used to take under account the fact that materialization is very expensive and the number of times it is performed should be minimized. To reduce the cost of processing queries, caching was also proposed as a way to avoid evaluating QUEL fields more than once and several issues associated with caching were discussed. Among others, replacement policies, invalidation algorithms and policies that decide which objects to cache were examined in detail. Our discussion showed that caching is essential in the QUEL+ environment. We suggested and analyzed various solutions to the above problems associated with caches. Lastly, a new indexing technique, Partial Indexing, was proposed. Partial indexes can be used for efficiently accessing results of QUEL field materializations. The proposed construct is a combination of both a conventional index table and a predicate, the latter characterizing those tuples that can be accessed through the former. Using partial indexes the system avoids the overhead of evaluating QUEL field entries before these are actually referenced in queries.

All the techniques described above were used to improve the performance of an extended query processor. If a QUEL field is accessed, the procedure stored in it must be executed. Since all database commands that constitute the body of the procedure are known, we have argued that some interquery analysis is possible. The objective of this analysis is to transform a sequence of commands to another sequence that can be processed more efficiently. In Chapter 3 several such optimization tactics were presented, some based on similar tactics in other areas (like compiler design or conventional query optimization) and some new ones. The level at which multiple command optimization was performed ranged from simple syntactic transformations (e.g. moving

loop invariants outside the loops) to harder semantic ones (e.g. changing an append followed by a `delete` command to a single append). Our new tactics include the somewhat surprising result that any QUEL program satisfying certain criteria is equivalent to a QUEL program which consists of one `replace` statement. We have also shown that a large class of problems, namely those which use the dynamic programming approach, satisfy these criteria. The transformations presented are useful not only in this context but in general transaction processing as well, since they are motivated solely by the need to expand the optimization unit from one database command to a sequence of commands. Our optimization techniques can be also applied as a preprocessing phase, i.e. given a set of applications and the corresponding database procedures that implement them, one can apply these techniques to design more efficient execution patterns. Experimental results [KUNG84] have shown that the gain in performance is significant.

Finally, in Chapter 4 we restricted the problem of procedure optimization to the case where all commands are retrievals from the database (global query optimization). This case is of significant interest because it can be used for efficient processing of queries in a rule based environment. The main motivation for doing interquery analysis is the fact that common intermediate results may be shared among different queries. We showed that various algorithms can be used for global query optimization. These algorithms were presented in the form of hierarchy; as we descend this hierarchy more sophisticated algorithms can be used giving better access plans at the expense of increased complexity of the algorithm itself. Some of the algorithms proposed were based simply on the idea of reusing temporary results from the execution

of queries, where the processing of each individual query is based on a locally optimal plan. The last (heuristic search) algorithm, is a variation of the algorithm for optimizing a set of relational expressions initially proposed by Grant and Minker in [GRAN80]. Through a preprocessing phase added to the algorithm we manage to achieve better average case performance. Finally, in a series of experiments, we simulated the proposed algorithms and checked the performance of the resulting global access plans under various database sizes and physical designs. Our results were very encouraging and showed a decrease 20-50% in both I/O and CPU time.

5.2. Future Directions

Relational DBMSs are very efficient in storing and accessing simple data like those used in business applications. Our results show that even more complex applications can be handled once the appropriate extensions are introduced. Although, solutions were proposed to several problems associated with extended relational database systems, there is still a lot of work that needs to be done in the area.

The most interesting and top priority issue should be the implementation of applications using QUEL+ and the experimentation and monitoring with a prototype system. POSTGRES [STON86b] can be used as a testbed for all our proposals. Also, in many points of our discussion we mentioned various parameters that should be known for the system to be better "tuned." One needs to collect a lot of statistical information and modify the algorithms we proposed so that the dynamics of the various applications are better reflected. Finally, dynamically adaptive caching schemes like the ones we proposed in Chapter 2 should be implemented and checked in real

application environments.

Our work on optimizing the execution of general procedures, was mostly influenced by examples used in engineering and heuristic search applications. We hope that future work will investigate the usefulness of our strategies in other environments as well, especially in rule based systems and more generally production systems [FORG79]. Transformations like the one we derived for dynamic programming problems, although not applicable to all procedures, add to our knowledge on the type of transformations one should be looking for. Future work in database procedure optimization should also look for more such special case transformations.

As interesting future research directions in the area of global query optimization we view the development of efficient algorithms for common subexpression identification and the extension of the algorithms presented to cover more general predicates. Also the application of our method in rule-based systems in general seems like a very interesting problem for investigation. For example, PROLOG and database systems based on logic [ULLM85] can easily be extended to perform global query optimization. Finally, some of the techniques that we developed here, can be applied in processing recursion in database environments [IOAN86]. This is mainly due to the fact that in evaluating recursive queries one usually processes iteratively similar operations. These operations often access the same data, for the relations accessed are always the same. Investigating how our algorithms can be used in this recursive query processing environment seems to be a very interesting problem for future research.

In summary, we think that there is a lot of work that can be done in database query processing and optimization. The introduction of new constructs and extensions gives rise to new interesting problems, especially if performance must be kept in sufficiently high levels.

BIBLIOGRAPHY

[ADIB80] Adiba, M.E. and Lindsay, B.G., *"Database Snapshots"*, Proceedings of the 6th International Conference on Very Large Data Bases, Montreal, October 1980.

[AHO79] Aho, A., Ullman, J., Principles of Compiler Design , Addison Wesley Co., 1979.

[ALLM76] Allman, E. et al, *"EQUEL Reference Manual"*, University of California, Technical Report UCB/ERL, Berkeley, CA, 1976.

[ASTR76] Astrahan, M. et al, *"System R: A Relational Approach to Database Management"*, ACM Transactions on Database Systems, (1) 2, June 1976.

[BERN79] Bernstein, P. and Goodman, N., *"Approaches to Concurrency Control in Distributed Data Base Systems"*, Proceedings of the 1979 National Computer Conference, 1979.

[BLAK86] Blakeley, J.A. , Larson, P. and Tompa, F.W., *"Efficiently Updating Materialized Views"*, Proceedings of the 1986 ACM-SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[BLAS76] Blasgen, M., Eswaran, K., *"On the Evaluation of Queries in a Relational Data Base System"*, IBM Research, Technical Report RJ-1745, San Jose, CA, April 1976.

[BROD84] Brodie, M. and Jarke, M., *"On Integrating Logic Programming and Databases"*, in [KERS84].

[BUNE79] Buneman, O.P. and Clemons, E.K., *"Efficiently Monitoring Relational Databases"*, ACM Transactions on Database Systems, (4) 3, September 1979.

[CARE84] Carey, M.J., DeWitt, D.J. and Graefe, G., *"Mechanisms for Concurrency Control and Recovery in Prolog - A Proposal"*, in [KERS84].

[CERI86] Ceri, S., Gottlob, G. and Wiederhold, G. , *"Interfacing Relational Databases and Prolog Efficiently"*, in [KERS86].

[CHAK82] Chakravarthy, U.S. and Minker, J., *"Processing Multiple Queries in Database Systems"*, in Database Engineering , (1), 1983.

[CHAK84] Chakravarthy, U.S., Fishman, D.H. and Minker, J., *"Semantic Query Optimization in Expert Systems and Database Systems"*, in [KERS84].

[CHAK85] Chakravarthy, U.S. and Minker, J., *"Multiple Query Processing in Deductive Databases"*, University of Maryland, Technical Report TR-1554, College Park, MD, August 1985.

[CHAK86] Chakravarthy, U.S., Minker, J. and Grant, J. *"Semantic Query Optimization: Additional Constraints and Control Strategies"*, in [KERS86].

[CHAN77] Chandy, K.M., *"Models of Distributed Systems"*, Proceedings of the 3rd International Conference on Very Large Data Bases, Tokyo, October 1977.

[CHOU85] Chou, H. and DeWitt, D.J., *"An Evaluation of Buffer Management Strategies for Relational Database Systems"*, Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.

[CLOC81] Clocksin, W. and Mellish, C., Programming in PROLOG , Springer-Verlag, New York, NY, 1981.

[CODD70] Codd, E., *"A Relational Model for Large Shared Data Banks"*, Communications of the ACM, (13) 6, 1970.

[KOPE84] Copeland, G. and Maier, D., *"Making Smalltalk a Database System"*, Proceedings of the 1984 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[DAYA85] Dayal, U. and Smith, J.M., *"PROBE: A Knowledge-Oriented Database Management System"*, Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, February 1985.

[DERR86] Derrett, N.P. *et al*, *"An Object-Oriented Approach to Data Management"*, Proceedings of the 1986 IEEE Spring Compcon Conference, San Francisco, CA, March 1986.

[EPST79] Epstein, R., *"Techniques for Processing Aggregates in Relational Database Systems"*, University of California, Technical Report UCB/ERL/M79/8, Berkeley, CA, 1979.

[ESWA76] Eswaran, K.P. et al, *"The Notions of Consistency and Predicate Locks in a Database System"*, Communications of the ACM, (19) 11, 1976.

[FINK82] Finkelstein, S., *"Common Expression Analysis in Database Applications"*, Proceedings of the 1982 ACM-SIGMOD International Conference on the Management of Data, Orlando, FL, June 1982.

[FORG79] Forgy, C., *"On the Efficient Implementation of Production Systems"*, PhD Thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1977.

[GALL78] Gallaire, H. and Minker, J., Logic and Data Bases , Plenum Press, New York, 1978.

[GARE79] Garey, M.R. and Johnson, D.S., Computers and Intractability , W.H. Freeman and Co, San Francisco 1979.

[GRAN80] Grant, J. and Minker, J., *"On Optimizing the Evaluation of a Set of Expressions"*, University of Maryland, Technical Report TR-916, College Park, MD, July 1980.

[GRAN81] Grant, J. and Minker, J., *"Optimization in Deductive and Conventional Relational Database Systems"*, in Advances in Data Base Theory , vol.

1, H. Gallaire, J. Minker and J.-M. Nicolas, Eds., Plenum Press, New York, 1981.

[GRAY78] Gray, J.N., *"Notes on Data Base Operating Systems"*, IBM Research, Technical Report RJ-2254, San Jose, CA, August 1978.

[GUTT84a] Guttman, A., *"R-Trees: A Dynamic Index Structure for Spatial Searching"*, Proceedings of the 1984 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[GUTT84b] Guttman, A., *"New Features for Relational Database Systems to Support CAD Applications"*, PhD Thesis, University of California, Berkeley, June 1984.

[HALL74] Hall, P.V., *"Common Subexpression Identification in General Algebraic Systems"*, IBM United Kingdom Scientific Centre, Technical Report UKSC 0060, November 1974.

[HALL76] Hall, P.V., *"Optimization of a Single Relational Expression in a Relational Data Base System"*, IBM Journal of Research and Development, **(20)** 3, May 1976.

[IOAN84] Ioannidis, Y. et al, *"Enhancing INGRES with Deductive Power"*, Position Paper, in [KERS84].

[IOAN86] Ioannidis, Y., *"Processing Recursion in Deductive Database Systems"*, PhD Thesis, University of California, Berkeley, July 1986.

[JARK84a] Jarke, M., Clifford, J. and Vassiliou, Y., *"An Optimizing PROLOG Front-end to a Relational Query System"*, Proceedings of the 1984 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[JARK84b] Jarke, M., *"Common Subexpression Isolation in Multiple Query Optimization"*, in Query Processing in Database Systems , W. Kim, D. Reiner and D. Batory, Eds., Springer-Verlag, New York, 1984.

[KATZ82] Katz, R.H., *"A Database Approach for Managing VLSI Design Data"*, Proceedings of the 19th Design Automation Conference, June 1982.

[KERN78] Kernighan, B. and Ritchie, D., The C Programming Language , Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KERS84] Kershberg, L., Editor, *Proceedings of the First International Workshop on Expert Database Systems*, Kiawah Isl., SC, October 1984.

[KERS86] Kershberg, L., Editor, *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, April, 1986.

[KIM84] Kim, W., *"Global Optimization of Relational Queries : A First Step"*, in Query Processing in Database Systems , W. Kim, D. Reiner and D. Batory, Eds., Springer-Verlag, New York, 1984.

[KOOI82] Kooi, R. and Frankfurth, D., *"Query Optimization in INGRES"*, Database Engineering, (5) 3, September 1982.

[KOWA74] Kowalski, R., *"Predicate Logic as a Programming Language"*, Information Processing, North Holland, 1974.

[KUCK86] Kuck, S., Private Communication, University of Illinois, Urbana, IL, March 1986.

[KUNG84] Kung, R. et al, *"Heuristic Search in Data Base Systems"*, in [KERS84].

[LARS78] Larson, R.E. and Casti, J.L., Principles of Dynamic Programming, Marcel Dekker, Inc, New York, 1978.

[LARS85] Larson, P. and Yang, H., *"Computing Queries from Derived Relations"*, Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.

[LORI79] Lorie, R., Casajuana, R. and Becerril, J., *"GSYSR: A Relational Database Interface for Graphics"*, IBM Research, Technical Report RJ-2511, San Jose, CA, April 1979.

[LORI81] Lorie, R., *"Issues in Database for Design Applications"*, IBM Research, Technical Report RJ-3176, San Jose, CA, July 1981.

[LORI82] Lorie, R. and Plouffe, W., *"Complex Objects and their Use in Design Transactions"*, IBM Research, Technical Report RJ-3706, San Jose, CA, December 1982.

[LORI83] Lorie, R. and Plouffe, W., *"Relational Databases for Engineering Data"*, IBM Research, Technical Report RJ-3847, San Jose, CA, April 1983.

[MATT70] Mattson, R.L. et al, *"Evaluation Techniques for Storage Hierarchies"*, IBM Systems Journal, (**9**) 2, 1970.

[MYLO80] Mylopoulos, J. et al, *"A Language Facility for Designing Database Intensive Applications"*, ACM Transactions on Database Systems, (**5**) 2, June 1980.

[NAQV84] Naqvi, S. and Henschen, L., *"On Compiling Queries in Recursive First-Order Databases"*, Journal of the ACM, (**31**) 1, January 1984.

[PAVL83] Pavlovic, G.M., *"Using a Relational Data Base System to Store Text"*, University of California, Technical Report UCB/ERL/M83/44, Berkeley, CA, July 1983

[RICH83] Rich, E., Artificial Intelligence , McGraw-Hill, 1983.

[ROSE80] Rosenkrantz, D.J. and Hunt, H.B., *"Processing Conjunctive Predicates and Queries"*, Proceedings of the 6th International Conference on Very Large Data Bases, Montreal, October 1980.

[ROUS82a] Roussopoulos, N., *"View Indexing in Relational Databases"*, ACM Transactions on Database Systems, (**7**) 2, June 1982.

[ROUS82b] Roussopoulos, N., *"The Logical Access Path Schema of a Database"*, IEEE Transactions on Software Engineering, (**8**) 6, November 1982.

[ROUS86] Roussopoulos, N. and Kang, H., *"Preliminary Design of ADMS±: A Workstation-Mainframe Integrated Architecture for Database*

Management Systems", University of Maryland, Technical Report, College Park, MD, February 1986.

[RTI84] EQUEL/C User's Guide , Version 2.1, Relational Technology, Inc., Berkeley, CA, July 1984.

[SCHK78] Schkolnick, M., "*A Survey of Physical Database Design Techniques*", Proceedings of the 4th International Conference on Very Large Data Bases, 1978.

[SCIO84] Sciore, E. et al, "*Towards an Integrated Database-PROLOG System*", in [KERS84].

[SELI79] Selinger, P. et al, "*Access Path Selection in a Relational Data Base System*", Proceedings of the 1979 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1979.

[SELL85] Sellis, T. and Shapiro, L., "*Optimization of Extended Database Languages*", Proceedings of the 1985 ACM-SIGMOD International Conference on the Management of Data, Austin, TX, May 1985.

[SELL86] Sellis, T., "*Global Query Optimization*", Proceedings of the 1986 ACM-SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[SHIP81] Shipman, D., "*The Functional Model and the Data Language Daplex*", ACM Transactions on Database Systems, (6) 1, March 1981.

[STON75] Stonebraker, M., "*Implementation of Integrity Constraints and Views by Query Modification*", Proceedings of the 1975 ACM-SIGMOD International Conference on the Management of Data, San Jose, CA, June 1975.

[STON76] Stonebraker, M. et al, "*The Design and Implementation of INGRES*", ACM Transactions on Database Systems, (1) 3, September 1976.

[STON83] Stonebraker, M. et al, "*Document processing in a relational database system*" ACM Transactions on Office Information Systems, (1) 2, April

1983.

[STON84] Stonebraker, M. et al, *"Quel as a Data Type"*, Proceedings of the 1984 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[STON85] Stonebraker, M. et al, *"Extending a Data Base System with Procedures"*, University of California, Technical Report UCB/ERL/M85/59, Berkeley, CA, July 1985.

[STON86a] Stonebraker, M., Sellis, T. and Hanson, E., *"Rule Indexing Implementations in Database Systems"*, in [KERS86].

[STON86b] Stonebraker, M. and Rowe, L., *"The Design of POSTGRES"*, Proceedings of the 1986 ACM-SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[TSUR84] Tsur, S. and Zaniolo, C., *"An Implementation of GEM - Supporting a Semantic Data Model on a Relational Back End"*, Proceedings of the 1984 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[ULLM82] Ullman, J., Principles of Database Systems , Computer Science Press, 1982.

[ULLM85] Ullman, J., *"Implementation of Logical Query Languages for Data Bases"*, Proceedings of the 1985 ACM-SIGMOD International Conference on the Management of Data, Austin, TX, May 1985.

[WARR81] Warren, D., *"Efficient Processing of Interactive Relational Database Queries Expressed in Logic"*, Proceedings of the 7th International Conference on Very Large Data Bases, Cannes, 1981.

[WILE84] Wilensky, R., The LISP PRIMER , W. Norton, Co, New York, 1984.

[WONG76] Wong, E. and Youssefi K., *"Decomposition: A Strategy for Query Processing"*, ACM Transactions on Database Systems, (1) 3, September 1976.

[WONG85] Wong, E., *"Extended Domain Types and Specification of User Defined Operators"*, University of California, Unpublished Manuscript, Berkeley, CA, February 1985.

[YOUS78] Youssefi, K., *"Query Processing for a Relational Database System"*, PhD Thesis, University of California, Berkeley, 1978.

[ZANI83] Zaniolo, C., *"The Database Language GEM"*, Proceedings of the 1983 ACM-SIGMOD International Conference on the Management of Data, San Jose, CA, May 1983.

[ZANI84] Zaniolo, C., *"PROLOG : A Database Query Language for all Seasons"*, in [KERS84].

[ZANI85] Zaniolo, C., *"The Representation and Deductive Retrieval of Complex Objects"*, Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.

APPENDIX A

Queries Used in Experiments of Chapter 4

The set of queries used in the experiments of Chapter 4 were the following

```
Employees (name,employer,age,experience,salary,education)
Corporations (cname,location,earnings,president,business)
Schools (sname,level)

range of e is Employees
range of c is Corporations
range of c1 is Corporations
range of s is Schools

/* get all employees with more than 10 years experience */

(1). retrieve (e.all) where e.experience ≥ 10

/* get all employees less than 65 years old with more than 20 years
   experience */

(2) retrieve (e.all) where e.experience ≥ 20 and e.age ≤ 65

/* get all pairs (employee,corporation), where the employee
   has more than 10 years experience and works in a corporation with
   earnings more than 500K and located anywhere but in Kansas */

(3) retrieve (e.all,c.all)
   where e.experience ≥ 10 and e.employer=c.cname
   and c.location ≠ "KANSAS" and c.earnings > 500
```

/ get all pairs (employee,corporation), where the employee has more than 20 years experience and works in a corporation with earnings more than 300K and located anywhere but in Kansas */*

(4) retrieve (e.all,c.all)
where e.experience ≥ 20 and e.employer=c.cname
and c.location ≠ "KANSAS" and c.earnings > 300

/ get all pairs (president,corporation), where the president is less than 65 years old with more than 20 years experience and the corporation is located in NEW YORK and has earnings more than 500K */*

(5) retrieve (e.all,c.all)
where e.experience ≥ 20 and e.age ≤ 65
and e.employer=c.cname and e.name=c.president
and c.location = "NEW YORK" and c.earnings > 500

/ get all pairs (president,corporation), where the president is less than 60 years old with more than 30 years experience and the corporation is located in NEW YORK and has earnings more than 300K */*

(6) retrieve (e.all,c.all)
where e.experience ≥ 30 and e.age ≤ 60
and e.employer=c.cname and e.name=c.president
and c.location = "NEW YORK" and c.earnings > 300

/ get all triples (employee,corporation,school) where the employee is less than 65 years old, has more than 20 years experience and holds a university degree working for a corporation located in NEW YORK and with earnings more than 500K */*

(7) retrieve (e.all,c.all,s.all)

```
   where e.experience ≥ 20 and e.age ≤ 65
   and e.employer=c.cname
   and c.location = "NEW YORK" and c.earnings > 500
   and e.education = s.sname and s.level="univ"
```

```
/* get all pairs (employee,corporation), where the employee
   is less than 65 years old with more than 20 years experience and the
   corporation is located in NEW YORK and has earnings more than 300K */
```

```
(8)  retrieve (e.all,c.all)
   where e.experience ≥ 20 and e.age ≤ 65
   and e.employer=c.cname
   and c.location = "NEW YORK" and c.earnings > 300
```