PATTERN-BASED AND KNOWLEDGE-DIRECTED QUERY COMPILATION FOR RECURSIVE DATA BASES

by

Jiawei Han

Computer Sciences Technical Report #629
January 1986

PATTERN-BASED AND KNOWLEDGE-DIRECTED QUERY COMPILATION FOR RECURSIVE DATA BASES

by

JIAWEI HAN

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

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1985

ABSTRACT

PATTERN-BASED AND KNOWLEDGE-DIRECTED QUERY COMPILATION FOR RECURSIVE DATA BASES

JIAWEI HAN

Under the Supervision of Professor Larry Travis

Expert database systems (EDS's) comprise an interesting class of computer systems which represent a *confluence* of research in *artificial intelligence, logic*, and *database management systems.* They involve knowledge-directed processing of large volumes of shared information and constitute a new generation of knowledge management systems.

Our research is on the deductive augmentation of relational database systems, especially on the efficient realization of recursion. We study the compilation and processing of recursive rules in relational database systems, investigating two related approaches: *pattern-based recursive rule compilation* and *knowledge-directed recursive rule compilation and planning.*

Pattern-based recursive rule compilation is a method of compiling and processing recursive rules based on their recursion patterns. We classify recursive rules according to their processing complexity and develop three kinds of algorithms for compiling and processing different classes of recursive rules: *transitive closure algorithms, SLSR wavefront algorithms*, and *stack-directed compilation algorithms.* These algorithms, though distinct, are closely related. The more complex algorithms are generalizations of the simpler ones, and all apply the heuristics of *performing selection first* and *utilizing previous processing results (wavefronts)* in reducing query processing costs. The algorithms are formally described and verified, and important aspects of their behavior are analyzed and experimentally tested.

To further improve search efficiency, a *knowledge-directed recursive rule compilation and planning* technique is introduced. We analyze the issues raised for the compilation of recursive rules and propose to deal with them by incorporating functional definitions, domain-specific knowledge, query constants, and a planning technique. A prototype knowledge-directed relational planner, RELPLAN, which maintains a high level user view and query interface, has been designed and implemented, and experiments with the prototype are reported and illustrated.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Larry Travis, for introducing me to the area of *logic and databases* and for supervising this dissertation. His work on *logic and databases*, his teaching on *logic programming* and *expert systems*, his support, guidance and encouragement of the work, and especially his extraordinary patience while modifying and editing my thesis drafts have made this thesis a reality. I would like to thank the other members of my thesis advising committee: Professor David DeWitt and Professor Mike Carey. Professor DeWitt led me to the area of relational database systems. He taught me a great deal on introductory and advanced relational database topics, and has given me great encouragement and support since I started my research on expert database systems. Professor Carey has also greatly encouraged my working in this direction. He has spent lots of his valuable time discussing many issues with me, directing me to the important problems of this research, and reviewing my papers for conferences.

I would like to thank all the professors in the Department of Computer Sciences who have led me through the various areas of computer science in the past few years. Professor Charles Dyer has given me valuable suggestions on my research. Professor Randy Katz (now at UC-Berkeley) initiated my interest in the issues of database applications, and Professor Anthony Klug, whose tragic death in 1983 was a major loss to the Department, guided my early work on database systems.

I should also express my thanks to all of my friends in Madison. They encouraged me and helped me when I met difficulties, and made my stay in Madison memorable. In particular, Hongjun Lu has discussed many interesting issues with me and helped me on the performance tests, and Zenian Li has had many helpful discussions with me on the issues of recursive databases.

Lastly, I would like to thank my parents for their love, education and support since my childhood, my brothers and sister for their encouragement and help, and my wife Yandong Cai for her love, understanding, support and sacrifice.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

1.1. EXPERT DATABASE SYSTEMS

Expert database systems (EDS's) comprise an interesting class of computer systems which represent a **confluence** of research in **artificial intelligence, logic,** and **database management systems.** They involve knowledge-directed processing of large volumes of shared information and constitute a new generation of knowledge management systems destined to play an increasing role in scientific, governmental and business applications.

As an emerging research field, expert database system research has attracted wide interest among database, artificial intelligence, and logic programming researchers. EDS's endow database systems with deductive reasoning and planning power, and they increase the processing efficiency of data intensive expert systems by using well-developed database technology. EDS technology is essentially the merging of two technologies, database system technology and expert system technology, toward the realization of knowledge management systems with great processing power and high processing efficiency. The development of EDS technology will benefit many related fields, including deductive database systems, data intensive expert systems, decision support systems, engineering database systems, office automation systems, etc.

Among the numerous computer science research issues raised by EDS's, our study is specifically focused on the issue of augmenting relational database systems with recursive processing and planning power, which many researchers believe to be one of the most important issues in EDS research.

1.2. OUR RESEARCH

Our research is on the efficient realization of recursion in relational database systems. We study the compilation and processing of recursive rules in relational database systems, investigating two related approaches: **pattern-based recursive rule compilation** and **knowledge-directed recursive rule compilation and planning.**

Pattern-based recursive rule compilation is a method of compiling and processing recursive rules based on their recursive patterns. We classify recursive rules according to their processing complexity and develop three kinds of algorithms for compiling and processing different classes of recursive rules: **transitive closure algorithms, SLSR wavefront algorithms,** and **stack-directed compilation algorithms.** These algorithms, though distinct, are closely related. The more complex algorithms are generalizations of the simpler ones, and all apply the heuristics of **performing selection first** and **utilizing previous processing results (wavefronts)** in reducing query processing costs.

To further improve search efficiency, a **knowledge-directed recursive rule compilation and planning mechanism** is introduced. We analyze the issues raised for the compilation of recursive rules by incorporating domain-specific knowledge, functional definitions, query constants, and a planning technique. We design and implement a prototype knowledge-directed relational planner, RELPLAN, which maintains a high level user view and query interface and which demonstrates the knowledge-directed compilation and planning for relational databases.

1.3. ARCHITECTURE : THE DEDUCTIVE AUGMENTATION OF RELATIONAL DATABASE SYSTEMS

1.3.1. EDS Architectures

There are several kinds of expert system architectures: **rule-based systems, logic-based** systems and **frame-based** systems. There are also several kinds of database system architectures: **relational, hierarchical, network, entity-relationship, object-oriented**, etc. We can combine and develop the two kinds of systems in various ways to construct various new architectures for EDS's. Even in the current early stage of EDS development, there are numerous and diverse proposals [Kers 84][Smit 84][Wied 84] on EDS architectures:

(1) ES -> DS

This approach starts from an existing expert system and adds database capabilities such as indexing, clustering, database accessing, authorization, concurrency control and recovery. A representative work of this kind is by Warren [Warr 84] which expands a Prolog logic programming system to include the functions and features of relational database systems.

(2) DS -> ES

This approach adds deductive reasoning capabilities to an existing database system, in most cases, a relational database system. Some work has been done on augmenting relational database systems with a Prolog front-end [Chan 84][Jark 84], but there are still many problems in "bridging" logic programming and relational database systems [Brod 84][Zani 84]. Many researchers are interested in using general Horn clauses, not constrained by Prolog syntax, semantics and control mechanisms, to enhance relational database systems with the power of deduction. Research in this direction has been characterized as **deductive database system** research [Gall 78, 81, 84]. This is essentially the area of our research.

(3) Object-oriented approach

This approach applies the concept of object-oriented programming [Gold 83][Baro 81][Cope 84] to design for new applications an extensible database system, whose architecture consists of several levels [Care 85]. These levels contain different objects such as data model objects, query processing objects, access method objects, storage objects, etc. Operations can be associated with the classes at each level to make them generic and easy to extend. The extension to expert database systems is effected by adding new objects for storing and processing rules.

1.3.2. Our Architecture: The Deductive Augmentation of Relational Database Systems

Our EDS architecture for deductive augmentation of relational database systems starts with a core around which various kinds of high-level deductive query interfaces, natural language interfaces, and more application oriented models can be added. Our study is focused on the construction and efficient implementation of such a core architecture using **rule compilation and planning**.

The core architecture of our deductive database systems is presented in Figure 1.1, which will be explained in detail in later chapters. It is the deductive augmentation of relational systems. For the case of **non-recursive rules**, such an augmentation has been researched in the field of **logic and databases** [Gall 84] and the field of **relational database systems** [Jark 84b]. The development of **recursion mechanisms** for relational database systems has been

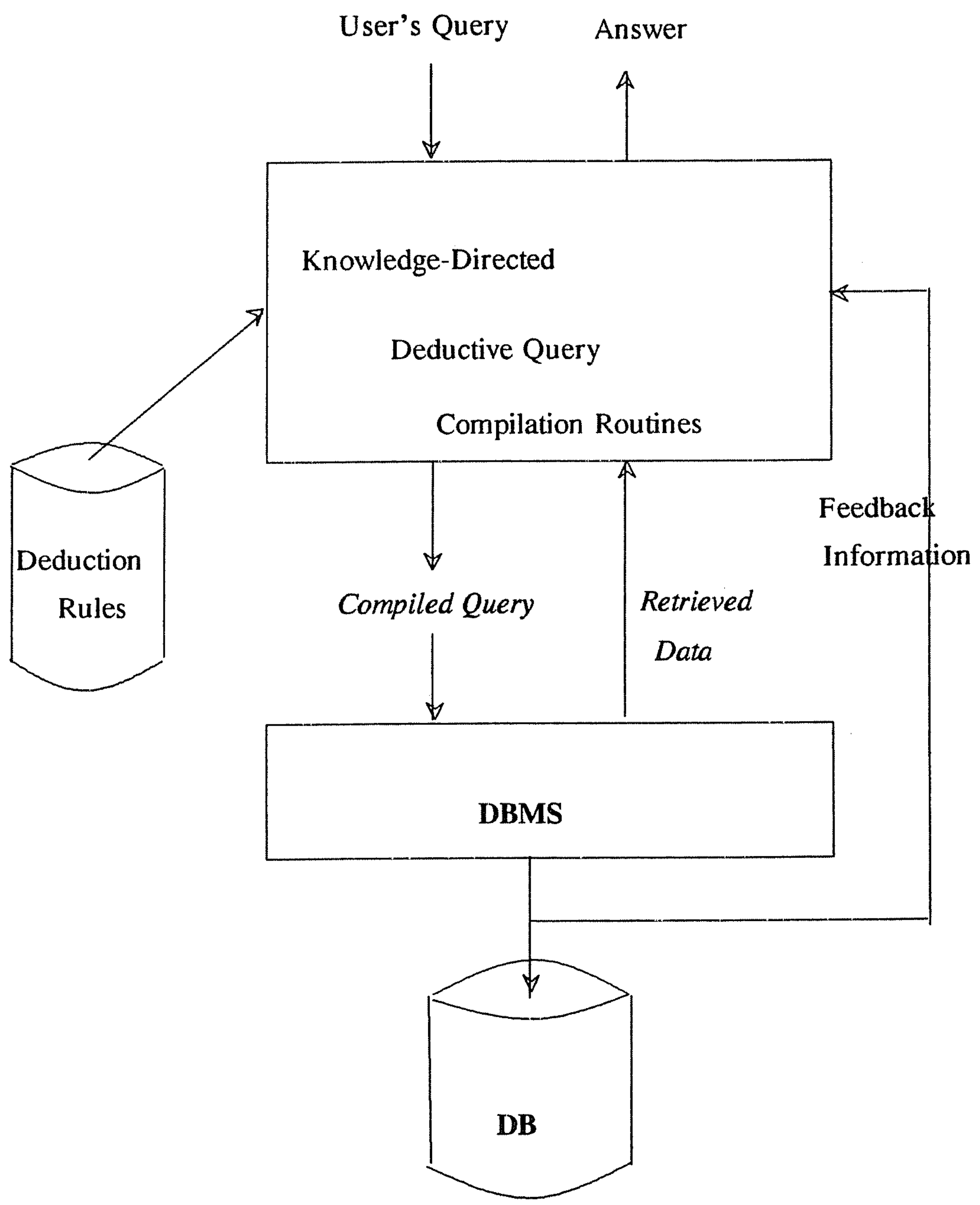


Figure 1.1. The Architecture of Our Expert Database System

an important focus of recent research [Gall 84][Hens 84][Ullm 85][Ioan 85].

1.4. ORGANIZATION OF THE DISSERTATION

The dissertation is organized into nine chapters.

Chapter 1 introduces the concept of expert database systems, their applications, and associated research issues. We outline our research and present the EDS architecture we will be investigating: **deductive augmentation of relational database systems.**

Chapter 2 is a survey of deductive query compilation techniques for non-recursive first-order databases. We contrast the compiled approach with the interpretive approach, point out the equivalence of the compilation approach and the view implementation technique using query modification, and discuss an implementation of the compiled approach for non-recursive deduction rules.

Chapter 3 is a general discussion of recursive first-order databases. We first survey the previous and current research work on processing of recursive database queries. Then, we specify some assumptions and definitions for our discussion, present a technique for transforming recursive rules into simplified forms when possible, and present a classification of recursive rules based on certain patterns, according to which their processing complexity can be determined.

From Chapter 4 through Chapter 6, we discuss compilation and processing of recursive rules according to the classification of Chapter 3.

Chapter 4 is on the compilation and processing of queries in the transitive closure class. After compiling a transitive closure rule into a **general compiled formula**, we compare several different strategies for the evaluation of the formula in databases and recommend a δ **wavefront algorithm** with a **tuple**

marking technique for efficient processing. We also discuss transitive-closure rule variations based on the δ wavefront algorithm.

Chapter 5 is on the processing of queries that involve SLSR rules, i.e. single looping rules with a single recursion point. After deriving the general compiled formula by compilation, four evaluation algorithms, **Natural Evaluation** (NE), **Single Wavefront (SW)**, **Double Wavefront (DW)**, and **Central Wavefront** (CW) are studied. The performance evaluation concludes that the **Single Wavefront Algorithm** has the best performance in most cases. **Performing selection first** and **using previous processing results (wavefronts)** are two important heuristics in the efficient processing of the compiled formulas.

Chapter 6 presents a compiled approach for more complex recursive rules. A **stack-directed compiled approach** is introduced. We perform a case study to derive a stack-directed compilation algorithm for an SLMR (single looping, multiple recursion points) rule set. We also discuss the processing of compiled formulas using **wavefront** and **potential wavefront** relations.

In Chapter 7 and 8, we develop a **knowledge-directed recursive rule compilation and planning technique**. This is motivated by EDS application requirements characterized by functional definition, high processing cost against large databases, and the availability of various kinds of domain-specific knowledge. Our analysis shows that knowledge-directed recursive rule compilation and planning can generate quite efficient processing plans.

Chapter 7 discusses new recursive rule processing problems which cannot be solved by the compilation techniques developed in previous chapters but can be solved by use of domain-specific knowledge, involving **termination constraints**, **search constraints** and **query constants**. The concept of a deductive module is developed here.

Chapter 8 introduces a **two-phase planning technique** for applying planning to recursive databases. The two phases are **selection of a planning strategy** and **generation of a retrieval program according to the selected strategy**. A relational planner, RELPLAN, for knowledge directed inference and planning in DB-oriented problem solving is specified. Analysis demonstrates significant improvement of processing efficiency by use of the proposed planning techniques.

We present our conclusions in Chapter 9, which summarizes the deductive compilation techniques that have been proposed, points out their limitations, and proposes future research which includes development of a comprehensive recursive query compiler, research on set-oriented heuristic search, and development of a plan database.

CHAPTER 2

THE DEDUCTIVE QUERY COMPILATION TECHNIQUE

In this chapter we discuss deductive query compilation techniques for non-recursive first-order databases. We contrast the compiled approach with the interpretive approach, point out the equivalence of the compiled approach for implementing virtual relations and the query modification technique for implementing views, and discuss an implementation of the compiled approach as a deductive front-end for relational databases.

2.1. FIRST-ORDER DATABASES

A deductive database is a database in which new facts may be derived from the facts that are explicitly contained in the database. A first-order database is a deductive database consisting of first-order clauses. Such databases can be classified into definite and indefinite deductive databases†, where a **definite deductive database** consists of Horn clauses which are definite assertions and definite data, and an **indefinite deductive database** may contain indefinite assertions and indefinite data. An indefinite assertion is an assertion whose consequent part consists of a disjunction of literals, and indefinite data contain facts represented by disjunctions of literals. Our research deals only with definite deductive databases. From here on, the term **first-order database** or **deductive database** refers to **definite deductive database** unless otherwise specified.

A deductive database consists of three parts: (i) an **extensional database (EDB)**, which consists of facts, represented by clauses with only one positive

† The terminology used here is from the research in **logic and databases** and can be found in [Gall 84].

literal containing no variables; (ii) an **intensional database (IDB)**, which consists of deduction rules, represented by clauses with one positive literal and one or more negative literals; and (iii) a set of **integrity constraint (IC)** rules.

For example, in a personnel database, an extensional database stores the specific facts, e.g., *person* with attributes *name*, *sex*, *birth-year*, *father*, *mother*, etc. An intensional database contains more general information, e.g., rule definitions of *brother*, *uncle*, *ancestor*, etc. Integrity constraints specify general information, such as that the *sex* of a *person*'s *father* is *male*, or that a *person*'s *birth-year* is always greater than his/her *father*'s.

Although integrity constraints, their efficient implementations, and their applications in deductive databases are important issues for research [Gall 78][Gall 81][King 81][Chak 84][Hens 84b], they are not going to be studied here. We concentrate on the issues of deduction rules. Therefore, in the following discussion, we ignore the issues of integrity constraints and treat deductive databases as databases containing only two parts: EDB and IDB.

For our discussion, we further divide first-order databases into **recursive** and **non-recursive** ones. A **recursive database** contains some recursive rules in its IDB while a **non-recursive** one does not. To distinguish a recursive rule from a non-recursive one, some notions need to be defined.

A **virtual relation** is jointly defined by deduction rules and elementary facts in deductive databases. A **deduction rule** has the form:

$$Q \text{ :- } P_1, P_2, \ldots, P_k. \tag{3-1}$$

where the **virtual relation literal** Q is called the **left side**, or the **consequent** of the rule, while $P_1, \ldots, P_k$ is called the **right side**, or the **antecedent** of the rule.

A **predecessor** of a literal is defined as follows: (i) P is the predecessor of Q if Q appears on the left side of the rule while P is on the right side; and (ii) the predecessor relation is reflexive and transitive.

A rule is **non-recursive** if the literal on the left side is not a predecessor of any literal on the right side, otherwise it is **recursive**.

For example, suppose that A, B and C represent base relations, and other letters represent virtual relations. In the following virtual relation definitions:

Q :- $P,A.$

P :- $B,C.$

R :- $A,S.$

S :- $B,C,R.$

T :- $A,T,C,T.$

The rules for P and Q are non-recursive while those for R, S and T are recursive.

2.2. DEDUCTION AND DATABASE ACCESS: COMPILATION VS. INTERPRETATION

2.2.1. Two Approaches: Compilation and Interpretation

There are two approaches to performing inferences in conjunction with database accesses: the interpretive approach and the compiled approach.

2.2.1.1. The Interpretive Approach

The interpretive approach [Mink 78] adopts a top-down search strategy which maintains a tree structure to control the search for rules and explicit facts and dynamically determines which portion (either explicit facts or implicit rules) to be searched. A general problem solver is used at the time the query is initiated. At each step the problem solver involves a selection function by applying some heuristics in determining the path to be searched and the rule to be applied in deduction. The control structure associated with the problem solver guides the search.

In the interpretive approach, a base-predicate is evaluated as soon as it is encountered. Search is interleaved with the extensional DB and intensional DB. To increase database search efficiency, a set-oriented approach has been explored [Warr 84]: at each access to DB, the search will obtain the whole set of tuples that satisfy same selection instead of one tuple at a time. Implementations of such a strategy can be found in [Warr 84].

The execution strategy in Prolog is essentially an interpretive approach. Prolog uses a simple depth-first search and backtrack control strategy to extract answers from its database, interleaving search of EDB and IDB. The Prolog database system *Solar 16* [Dahl 81] is a primitive implementation of the interpretive approach. Another Prolog database system *Chat-80* [Warr 81] applies a predicate-ordering mechanism, which is similar to the ideas of relational database access path selection [Seli 79].

2.2.1.2. The Compiled Approach

The compiled approach delays the evaluation of base predicates until all virtual predicates have been resolved into base predicates. The refutations are arranged so that all IDB literals are resolved away, leaving clauses consisting only of EDB literals. Such clauses can be handled by passing off relational expressions to a traditional relational database system, which can be processed by well-developed relational query processing techniques. Reiter showed that in the absence of recursively defined predicates, the set of all such EDB clauses can be determined before the database is accessed [Reit 78].

The compiled approach separates the theorem proving task from the database retrieval task. It reduces a problem which requires both deduction and database access to one which only requires database access, by using theorem proving techniques. In the initial step, only a theorem prover is applied. The theorem prover "sweeps through" the IDB, extracting all information relevant to a given query, without accessing the EDB. If the query contains an intensional literal, the left-hand portions of rules are matched. When a match occurs, the theorem prover replaces the intensional literal in the query with the right-hand sides of all the matching rules. This process continues until all the intensional literals are eliminated from the query. This ends the compilation phase. The set of compiled results is then passed to the relational DB system for retrieval.

The compiled approach transforms a query that requires deduction into a set of queries that do not require deduction, hence shifting inference search down to more efficient database search. The compiled approach is described in [Chan 78][Reit 78][Hens 84] and implemented in systems described in [Kell 78][Kell 81][Jark 84].

2.2.2. Processing Comparison: Compilation vs Interpretation

We illustrate with an example the processing of a non-recursive deductive database query according to the two approaches.

Example 2.1. Find Mary's tall uncles who are older than her father.

Suppose that the EDB contains only the relations: sex, age, height, father, and mother. Other relations, such as uncle, brother, and category are defined by deduction rules. Suppose the rules are as follows (using Prolog syntax),

$uncle(Name, Unc) :- parent(Name, Par), brother(Par, Unc).$ (2-1)

$parent(Name, Par) :- father(Name, Par); mother(Name, Par).$ (2-2)

$brother(Name, Bro) :-$

$sex(Bro, male),$

$((father(Name, Fa), father(Bro, Fa));$

$(mother(Name, Mo), mother(Bro, Mo))).$ (2-3)

$category(Name, tall) :-$

$height(Name, Height), age(Name, Age),$

$((Age \geq 18,$

$((sex(Name, male), Height > 6); (sex(Name, female), Height > 5.9)));$

$(Age \geq 12, Age < 18,$

$((sex(Name, male), Height > 5.8);$

$(sex(Name, female), Height > 5.7)))).$ (2-4)

$category(Name, medium) :- \ldots$ (2-5)

According to our question, the user's query is written as,

```
?-      uncle(mary, Unc),
        category(Unc, tall),
        age(U, AgeU),
        father(mary, Fa),
        age(Fa, AgeF),
        AgeF < AgeU.
```

The Interpretive Approach:

In the interpretive approach, the search can follow many possible search paths. One simple arrangement is to evaluate each predicate in the order that the query is presented. Such an arrangement is used by Prolog. Its efficiency depends on the programmer's skill and on underlying database structures. Optimal ordering strategies have been studied in [Warr 81]. We simply list the query in a reasonable order for our analysis and comparison with the compiled approach.

The interpretation of the query proceeds as follows,

(1) The search for *Unc* in *uncle*(*mary*, *Unc*) using query constant *mary* invokes deduction on the *uncle* rule. The result is new queries *parent*(*mary*, *Par*) and *brother*(*Par*, *Unc*).

(2) The search for *parent*(*mary*, *Par*) invokes the deduction on the *parent* rule. The search is then resolved to the union of *father*(*mary*, *Par*) and *mother*(*mary*, *Par*). Solving these two clauses involves the accessing of the EDB, which is done even though other literals in the formula have not yet been deductively resolved.

(3) The bindings of *Par* found by the EDB search are used to find *Unc* from the *brother* rule, which involves more deduction and EDB accessing.

(4) Then we start the evaluation of the second query literal *category*(*Unc*, *tall*) based on the set of answers *Unc* and the *category* rule. Later processing proceeds similarly and details are omitted here.

Comments on the interpretive approach:

(1) Rules can be high level constructs involving several levels of indirection and abstraction. It is a difficult task for a programmer or an optimization routine to decide the optimal processing order without resolving the rules into a single level construct containing EDB literals only. A mis-judged processing order may result in inefficient processing. For example, accessing *category*(*Unc*, *tall*) first in our query might lead to quite inefficient processing.

(2) The interactions among the definitions of different predicates are not explicit. It is difficult to perform global optimization. For example, the fact that *uncle* is male (not directly available from the definition *uncle*) makes irrelevant the half of the definition of *category* (which is about *female*). The removal of the useless half is difficult without expanding with both definitions before accessing the EDB.

(3) The definitions of different predicates may contain the same base predicates. Thus delaying of the evaluation of base predicates may avoid redundant processing. In the example, accessing *father*(*mary*,*Fa*) occurs more than once.

The Compiled Approach:

In the compiled approach, theorem proving operations, such as condensing, absorbing, factoring, and conflict removing [Chan 73][Love 78], are applied before EDB search. The result is a **compiled formula** which can be sent to relational query optimizers. In our example, the compiled result is in Figure 2.1.

Compared with the interpretive approach, the compiled approach spends more time on processing and optimizing clauses using theorem proving techniques with main memory algorithms. The volume of clauses processed at this stage is much smaller compared with the volume of data in large databases. Such clause processing and optimization result in the following advantages:

(1) Take advantage of global optimization;

The compiled approach permits global decisions to be made concerning which operations are to be performed on which tables and in which sequence, since the entire EDB search expression is available before any

```
(father(mary,Par);mother(mary,Par)),
sex(Unc,male),
((father(Par,Fa),father(Unc,Fa));
(mother(Par,Mo),mother(Unc,Mo))),
height(Unc,Height),
age(Unc,AgeU),
((AgeU≥18,Height>6);
(AgeU≥12,AgeU<18,Height>5.8)),
father(mary,F),age(F,AgeF),AgeF<AgeU.
```

Figure 2.1. The Compiled Formula for the Example Query

EDB search is done. The interpretive approach uses local decisions to guide the search process. However, locally optimal choices are often not globally optimal.

(2) No EDB accessing takes place before EDB query optimization; and

(3) Optimized EDB access expression may be compiled once and saved for recurring query types.

From the above discussion, we can see that the performance of the compiled approach in general surpasses that of the interpretive approach. However, in some cases where the problems may not require finding all solutions, an interpretive approach which uses a good heuristic search procedure (e.g. the A* algorithm [Nils 80] with good estimation for the cost function) may have a chance of performing better than the compiled approach.

2.3. DEDUCTIVE QUERY COMPILATION AND QUERY MODIFICATION

For non-recursive databases, **deductive query compilation** in the area of **logic and databases** corresponds to **query modification for view implementation** [Ston 75][Cham 75] in the area of **relational databases.** They are essentially the same technique developed in two different fields. Non-recursive deduction rules are essentially equivalent to view definitions. The deductive query compilation technique which performs resolution using deduction rules is equivalent to the query modification technique which reduces a query containing views to a query containing only base relations.

2.3.1. View Definition Is Equivalent to Definition with Non-Recursive Rules

In relational databases, a view is defined in terms of database relations or other views by a relational algebra or relational calculus expression. In non-recursive deductive databases, a virtual relation is defined in terms of extensional (base) relations or other virtual relations by Horn clauses. In non-recursive databases, a virtual relation so defined is equivalent to a view definition.

For example, the virtual relation *grand_parent* may be defined either with a view definition or with a Horn clause rule definition.

Example 2.2. Grand_parent is defined by two kinds of definitions.

View definition (in QUEL [Ullm 82]):

```
/* Assume the relation parent(ch,pa). */
range of p1 is parent
range of p2 is parent
define view grand_parent (gs = p1.ch, gp = p2.pa)
       where p1.pa = p2.ch
```

Horn clause definition:

$$grand\_parent(X,Z) \ :- \ parent(X,Y), parent(Y,Z). \tag{2-6}$$

□

If there are several deduction rules which define the same virtual relation, for example,

$$R \ :- \ A_1.$$
$$\cdots\cdots$$
$$R \ :- \ A_k.$$

these separate rules correspond to use of disjunctive combination (i.e., use of the operator *or*) in the view definition. This is shown in the following view definition.

Example 2.3. *parent* is defined by two deduction rules and a view definition with the disjunctive operator *or*.

Horn clause definition:

$$parent(X,Y) :- father(X,Y). \tag{2-7}$$

$$parent(X,Y) :- mother(X,Y). \tag{2-8}$$

View definition†:

```
/* Assume relations father(ch,fa) and mother(ch,mo). */
range of f is father
range of m is mother
define view p: parent (ch, pa)
        where
                (p.ch = f.ch and p.pa = f.fa)
        or
                (p.ch = m.ch and p.pa = m.mo)
```

□

2.3.2. Query Modification on Views Is Equivalent to 'Resolution'

In **logic and database** research, a query using deduction rules is resolved using techniques [Chan 78][Reit 81] based on Robinson's **resolution principle** [Robi 65]. In relational database technology, a query using views is transformed into relational operations on base relations by query modification techniques developed in INGRES [Ston 75] and System R [Cham 75]. The two approaches can be usefully compared with the following example.

Example 2.4. The transformation techniques: Query using deduction rules and query using views. For the query: *retrieve john's grandparent.*

In a deductive database system, the query can be written

$$?-\ grand\_parent(john, GP).$$

which is resolved using the *grand_parent* rule and the **resolution principle,** where "X" in the rule is unified with "$john$" and "GP" with "Z". The resolved query becomes

$$?-\ parent(john, Y), parent(Y, GP).$$

If *parent* is a base relation, the resolved query involves a selection on the *parent* relation, a join operation on the *parent* relations, and a projection on the last column of the joined relations to obtain GP.

On the other hand, in a relational database system, the query is written as,

range of g *is* grand_parent

† A little license with QUEL syntax has been taken, because the syntax of QUEL cannot accommodate this kind of definition nicely. This is discussed in the next subsection.

```
retrieve g.gp where g.gs = 'john'
```

Using query modification technique, the query is modified to

```
range of p1 is parent
range of p2 is parent
retrieve p2.pa
        where p1.ch = 'john' and p1.pa = p2.ch
```

which involves exactly the same selection, join, and projection operations on the same relations. □

The two techniques, although using different technologies, result in the equivalent relational expressions. This general equivalence has also been observed by [Wong 84].

However, recursive rules can be defined using Horn clauses and resolved using resolution techniques, but views cannot be defined recursively, because a view is defined by finite relational operations on base relations or *previously defined* views. If a view is to be defined recursively, it will require the modification of the view definition to include forward reference to some undefined views [Han 85a]. Such a definition is beyond the scope of conventional view definition in terms of finite relational operations, because in general cases it contains iterations with the number of iterations being data determined in the database (and thus, in general, is not determinable before the database is accessed.)

Strictly speaking, even in the case of non-recursive databases, a Horn-clause-specified virtual relation is more powerful than a view definition in two aspects:

(1) A virtual relation can be defined both by rules and facts, while a view can be defined only by rules (relational operations on data relations and other views).

For example, grand_parent may be defined using both rules and facts like,

$$grand\_parent(john, andrew). \tag{2-9}$$

$$grand\_parent(X,Z) \ :- \ parent(X,Y), parent(Y,Z). \tag{2-10}$$

This difference is essentially a naming difference. It can be accommodated by either assigning a different name to the intensional relation, or initializing two searches simultaneously: database search and deduction search.

(2) Some disjunctive relationships cannot readily be expressed in the syntax of view definitions. A virtual-relation definition containing defined attribute-values is also not easy to fit into the syntax of view definitions.

For example, in Example 2.3, *parent* is defined as *father or mother*, which can not be represented using strict QUEL syntax. However, with a minor modification of the syntax of QUEL as we illustrated, it accommodates the example. The rule definition with some attribute value specified by rules, such as the *category* rule definition in Example 2.1,

$category(Name, tall) \ :- \ \ldots$

$category(Name, medium) \ :- \ \ldots$

cannot be expressed in current view definitions (e.g. in QUEL or SQL [Ullm 82]) unless we use separate view definitions of *tall*, *medium*, etc. The modification of the syntax of a relational language to handle these cases is discussed in [Han 85a].

These two points are, however, minor differences between view and virtual-relation definitions, and they can be solved by slight modifications of the view definitions in query languages.

2.3.3. The Implementation of Non-Recursive Rules in Relational Databases

The ideas of the preceding section raise the possibility of Horn clauses functioning as a syntactic variation of relational languages as used in view definitions and queries. We have coded a transformation program which transforms simple relational view definitions and queries into Horn clauses and vice versa. The program is coded in C language [Kern 78] using the UNIX † utilities LEX and YACC.

For the realization of non-recursive deductive query compilation, a deductive query compilation program has been coded as part of our RELPLAN system (see Chapter 7 and 8). The compilation is based on the principles of query modification and variable substitution in deductive rule resolution.

The following example shows the compilation of a non-recursive deductive query. The input is a collection of deduction rules and queries, while the output is the resolved query which only contains relational operations on base relations.

Example 2.5. Find Mary's tall uncles who are older than her father.

The first part of the RELPLAN input contains the schemas and virtual relation definitions in Figure 2.2 (a); the second part of the RELPLAN input is a user deductive query shown in Figure 2.2 (b); and the output of the RELPLAN preprocessor is the resolved query in Figure 2.2 (c).

The compilation process can be divided into several steps.

(1) For each variable which references a virtual relation, e.g. variable *c*, follow the query parse tree up to find an *or_node* or a *where_root*, as the rule aug-

† UNIX is a registered trademark of Bell laboratories.

```
schema person ( name, age, sex, fa, mo, height)
range of p1, p2 ,p3 is person
define virtual relation b : brother(name = p1.name, bro = p2.name)
        where p1.fa = p2.fa and p1.mo = p2.mo and p2.sex = "male"
define virtual relation pa : parent(ch = p1.name, pr = p2.name)
        where p1.fa = p2.name or p1.mo = p2.name
define virtual relation u : uncle(name = p1.name, unc = p2.name)
        where p1.name = pa.ch and pa.pr = b.me and b.bro = p2.name
define virtual relation c : category(name , scale )
        where c.name = p1.name and ((c.scale = "tall" and
        (p1.sex = "male" and p1.height > 6) or (p1.sex = "female" and
        p1.height > 5) ) or (c.scale = "medium" and (p1.sex = "male"
        and p1.height <= 6 and p1.height > 5) or (p1.sex = "female"
        and p1.height <= 5 and p1.height > 4) ))
```

(a) Schema and Virtual Relation Definitions.

```
range of x is uncle
range of c is category
range of p1, p2 ,p3 is person
retrieve (x.name, x.unc) where
        x.name = "mary" and x.unc = c.name and
        c.scale = "tall" and x.name = p1.name and
        x.unc = p3.name and p1.fa = p2.name and p2.age < p3.age
```

(b) A User's Deductive Query

```
range of p1 is person
range of p2 is person
range of p3 is person
range of p_hf is person
retrieve ( p1.name  ,  p3.name  )
        where p1.name = "mary" and  p2.age < p3.age and
        p2.name = p1.fa and  p3.height > 6  and
        ( p1.fa = p_hf.name or  p1.mo  =  p_hf.name  )
        and p_hf.fa = p3.fa and p_hf.mo = p3.mo and  p3.sex = "male"
```

(c) The Resolved Query Program by Deductive Compilation

Figure 2.2. Compilation of Non-Recursive Rules in RELPLAN

mentation point.

(2) Substitute for the variable its rule definition, combine the query with the rule definition at the rule augmentation point to form a combined query tree, and rename the conflicting variable names that have resulted, if any, e.g. rename p1 to p_hf.

(3) Perform merging, conflict removing and collapsing on the combined query tree to simplify it. For example, the *medium* part in the *category* rule conflicts with *tall* uncle in the query and is thus collapsed. The same happens in the *female* part, which conflicts with the *brother* rule definition. Only the *male* subtree of the *tall* part in the *category* rule is augmented with the user's query. The collapsing technique is a kind of query optimization.

(4) Repeat the above process for every virtual variable in the modified query until all virtual relation references are resolved. □

Note that locating the rule augmentation point is important for rule augmentation. The augmentation point is not always at the root of the *where_clause*. This can be seen when all the references of a virtual relation are located lower than the first ancestor *or_node* in the query tree, e.g., for the query *finding some people who either have a brother John or have a sister Mary.* In QUEL, it is

```
/* Assume relations brother(name,bro) and sister(name,sis) */
range of p is person
range of b is brother
range of s is sister
retrieve p.name
        where (p.name = b.name and b.bro = "john")
                or (p.name = s.name and s.sis = "mary")
```

The rule augmentation point should be at the *or_node*: augment the *brother* rule at the left subtree of the *or* node and the *sister* rule at the right subtree.

The compilation process can be summarized as Algorithm 2.1.

Algorithm 2.1. Compilation of Non-Recursive Deductive Queries.

(1) Locating the rule augmentation point.

For each tuple variable which references a virtual relation, traverse up the query tree to the first ancestor *or_node* or to *the root of the where_clause*, whichever comes first. This node is then marked as the rule augmentation point.

(2) Rule augmentation :

Augment the rule definition by *anding* it with the referenced child of the rule augmentation point in the query tree. The combined subtree is linked to the rule augmentation point and the virtual relation reference is replaced by its definition. If there is any variable in the rule definition which is identical with the variables of the query, the variable is renamed.

(3) Merging and collapsing.

Check if there are any conflicting subtrees of the query part with the augmented rule definition, and collapse the conflicting part of the rule definition until the *or_node* is met. The *or_node* together with the conflicting part is removed. If the collapsing continues to the root of the where_clause, it means that the query contradicts the rule definition and a null answer can be returned. Also check if there are any overlapped or redundant subtrees. An overlapped part should be merged and a redundant part removed.

(4) Repeat the above process (step 1-3) for the modified query, until all intensional (virtual) relation references in the query are resolved into extensional

(base) relation references. □

The algorithm is implemented in RELPLAN and demonstrated in Example 2.5.

CHAPTER 3

A GENERAL DISCUSSION OF COMPILATION OF RECURSIVE RULES

Beginning with this chapter, we direct our attention to the compilation and processing of recursive rules in relational database systems. In this chapter, we survey previous and on-going research work on the processing of recursive database queries, specify some assumptions and definitions, develop a technique to transform recursive rules into more standard forms, and present a classification of recursive rules.

3.1. RECURSION IN RELATIONAL DATABASES : PREVIOUS AND CURRENT WORK

3.1.1. The Challenge of Recursion and the Limitation of Relational Languages

The last chapter demonstrates that non-recursive deductive queries can be processed using either **query modification techniques** developed in **relational database** systems or **deductive query compilation techniques** developed in **logic and database research**. However, when extending processing power to include **recursion**, both techniques are limited in their capabilities.

In database system research, early in 1972 Codd studied the completeness of the relational calculus [Codd 72] and pointed out the limitation of relational languages in dealing with one kind of recursive queries, least-fixed-point queries. †

† A least fixed point equation R = f(R, D) is set-monotonic if for all relations Z in D that are in the domain of f, $Z \subseteq f(Z, D)$. For database states D and all set-monotonic recursions, a unique least-fixed-point LFP(f, D) exists. This was first concluded in [TARS 55].

Aho and Ullman [Aho 79] further analyzed the least-fixed point problem, pointing out that conventional relational algebra and calculus languages cannot specify recursive rules and recursive queries. They suggested use of programming primitives such as iterative and conditional constructs like "while" and "if" to augment relational languages to be able to handle least-fixed-point problems. However, the efficient implementation of this solution to least-fixed-point problems in relational databases was not explored. Also, unlike non-procedural recursive rules, the solution is a procedural extension of relational languages.

3.1.2. The First Attempt : Reiter and Chang's Compilation Approach

Both compiled and interpretive approaches to processing recursive queries have been studied [Chan 78][Chan 81][Reit 78][Mink 78][Mink 81]. The interpretive approach often results in tuple-oriented processing and redundant patterns of database access. Moreover, this approach has difficulties in determining termination conditions, because it cannot tell when complete answers have been found [Gall 84]. The compiled approach applies the **resolution principle** to generate compiled formulas for recursive queries. Because these formulas do not contain intensional literals, set-oriented relational operations and well-developed relational query processing techniques can be applied for efficient processing. However, recursion causes two problems, termination and processing efficiency, for the compiled approach:

(1) Recursion may make the resolution chain grow indefinitely long, and

(2) It also often involves costly iterative processing on large databases.

[Reit 78] and [Chan 81] first studied recursion in relational databases using the compiled approach. [Reit 78] developed a compilation technique for databases containing recursive literals in the deduction rules. [Chan 81] developed a method which transforms a query involving a recursive statement into an iterative program.

However, Chang's method suffers from two drawbacks. First, at most one intensional literal is allowed in the antecedent of the recursive statement. Second, the termination condition for the derived program is not stated. [Reit 78] suggested that one of the literals causing the recursion should be explicitly represented in the database, but the solution undercuts one of the major benefits of having indirect intensional literals. [Mink 81] derives iterative programs, as in [Chan 81], for a particular class of recursive axioms, called singular axioms. These discussions have been restricted to the compilation of transitive closure rules. The compilation of more complex recursive rules was not discussed. The efficient processing of compiled retrieval programs in relational databases and the determination of termination conditions were not studied. Their research should be considered as an important first step in studying recursion in relational databases.

3.1.3. Shapiro and McKay's Work

In the field of AI research, Shapiro and McKay [Shap 80][Mcka 81] reported a solution for processing recursive rules, in which the entire relation corresponding to a recursive statement is generated. This method is oriented toward AI applications and would be inefficient if applied to large database systems. Further, it makes no distinction between intensional and extensional components of databases. The performance of Shapiro and McKay's approach in a large database will be compared and contrasted with Henschen and Naqvi's approach in Chapter 5.

3.1.4. Henschen and Naqvi's Compiled Approach

Henschen and Naqvi [Hens 84] applied resolution-proof techniques over connection graphs [Kowa 79] to compile recursive rules into iterative database retrieval programs that give all answers to a query and contain a well-defined termination condition. Their method applies to recursive rules with a single resolution cycle (only one cycle in the connection graph where links are built up between literals resolvable using resolution-proof techniques). There are two distinct features to their solution. The first is that their approach takes into consideration the need to use query constants at the earliest possible search stage. Doing so may considerably reduce the space to be searched and the size of intermediate relations generated. The second is that they make use of previous processing results to avoid some redundant processing.

Their approach works out a general form of **potential recursive loop (PRL)** from a connection graph, and applies the concepts of **determined variables** and **induced variables** to derive iterative query programs for recursive database queries. For example, for the following rule set

$$T(x,y):-P(w,z),S(y,w). \qquad (3\text{-}1)$$

$$S(x,z):-M(x,y),T(y,z). \qquad (3\text{-}2)$$

and

$$T(y,z):-F(y,z). \qquad (3\text{-}3)$$

the connection graph illustrated in Figure 3.1 can be built. Its potential recursive loop (PRL) can be identified, and a query on an intensional literal such as the following can be compiled into a query program.

$$?-S(?,a). \qquad (3\text{-}3)$$

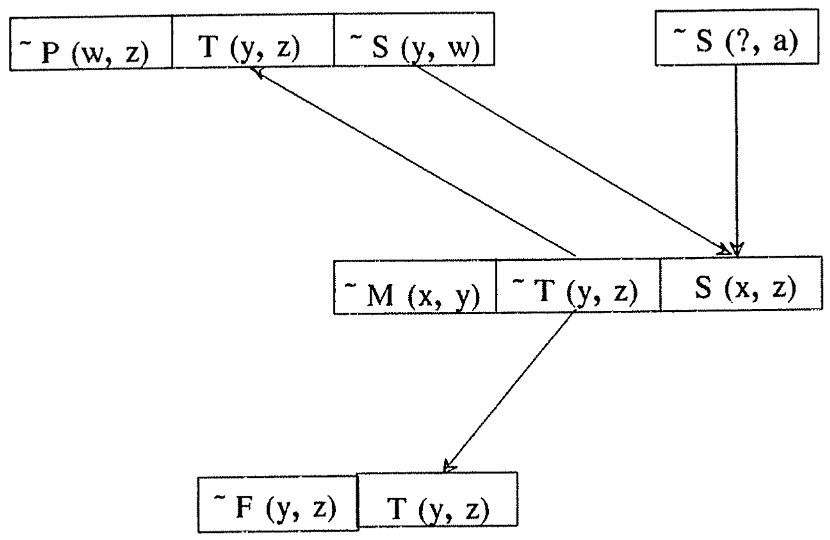


Figure 3.1.

An Example of Henschen and Naqvi's Connection Graph

However, the Henschen and Naqvi algorithm has some limitations. First, for recursive rules containing more than one resolution cycle, e.g., complex mutual recursions, they propose a set of mutually recursive processes to perform evaluation, but the proper control of the processes needs "operating systems research". Second, although their approach eliminates most of the redundancy, some still remains. Also, the derived programs contain tuple-oriented processing such as dequeue and enqueue of intermediate-result tuples, and the enqueue process may be time consuming.

3.1.5. Capture Rules: Ullman's Approach

Ullman explored query processing techniques for recursive rules using the concept of capture rules [Ullm 85]. A capture rule is a law that under certain conditions we may "capture" certain nodes of the rule/goal graph provided that certain other nodes (perhaps none) are already captured. It is information abstraction from a graph representing clauses and predicates. He defined four different kinds of capture rules: a **basic capture rule,** which corresponds to application of operators from relational algebra; a **top-down capture rule,** which corresponds to "backward" chaining; a **bottom-up rule,** corresponding to "forward chaining", to deduce all true facts in a given class; and a **side-way rule,** passing results from one goal to another. He pointed out that these rules can be applied independently, thus providing a clean interface for query evaluation systems that use different strategies in different situations.

His approach uses a technique of "smearing" detail, sometimes causing an effective evaluation strategy to be missed. For example, consider the rule,

$$A(1,x) :- B(x),\ A(2,x). \qquad (3\text{-}4)$$

This rule will not produce any recursive answers, because the head of the clause does not unify with the occurrence of A in the body. However, Ullman's basic rule cannot tell this.

Our research does not apply the concept of Ullman's capture rules. Instead, we develop our approach using the more traditional compiled approach, based on the deductive resolution principle and Kowalski's connection graphs [Kowa 79]. The effectiveness of Ullman's approach, compared with the traditional approach, remains to be seen.

3.1.6. The Recognition of Pseudo-Recursion: Ioannidis' Approach

Ioannidis has developed a special variable connection graph to detect and determine the upper bound on the number of recursions necessary to form a recursively defined virtual relation, independent of base relations [Ioan 85].

His approach distinguishes a special kind of recursion in which the upper bound of recursive calls is independent of the data stored in the database. This technique is useful for some special kinds of variable patterns, but these kinds may not occur frequently in practice.

Ioannidis graph is constructed by (1) associating with every variable a node in the graph, (2) building an undirected edge with length zero for every pair of variables in the same non-recursive literal, (3) building a directed edge $x \rightarrow y$ with length one for every variable pair x and y such that x is in the antecedent recursive literal and y is in the corresponding position of the consequent literal. The recursive rule is bounded if the graph contains no cycles of non-zero length.

For example, consider the recursive rule

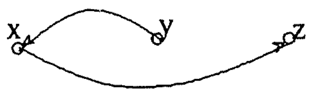
$$R(x,z) :- R(y,x),\ B(z). \qquad (3\text{-}5)$$

By construction of the Ioannidis graph in Figure 3.2(a), we find that the upper bound of the recursive application of this rule will be two (the length of the longest path in the graph), which is data independent.

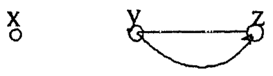
But more frequently we meet a rule like

$$R(x,z) :- R(x,y),\ B(y,z). \qquad (3\text{-}6)$$

which has the Ioannidis graph Figure 3.2(b). It clearly does not have a data independent upper bound.



(a) R(x, z) :- R(y, x), B(z).



(b) R(x, z) :- R(x, y), B(y, z).

Figure 3.2.

An Example of Ioannidis' Variable Graph

3.1.7. Other On-Going Research

There is some on-going research at MCC and other research organizations, emphasizing the efficient processing of transitive closures, the simplest form of recursion in relational databases. Such recursion is indeed important, because most practical recursion problems involving large databases appear to fall within this class.

Although our research has concentrated on the more complex forms of recursion, we do pay attention to the efficient processing of transitive closures. The difference is that we use knowledge-directed compilation and planning, which involve AI techniques.

3.1.8. Our Research: Processing Complex Recursion and Applying AI Techniques

Our research is based on deductive resolution and compilation techniques developed in logic and database research [Kowa 79][Gall 78][Gall 81] [Gall 84]. We have been strongly influenced by the work of Henschen and Naqvi

[Hens 84] on compiling recursive queries against first-order databases.

Our research can be considered as an extension of Henschen and Naqvi's work in three aspects.

(1) **Compiling Recursive Rules in the Context of Relational Database Processing.**

We have further studied the processing of recursive rules in the context of relational database processing. The **enqueue** and **dequeue** processing is replaced by database set-oriented operations (join, selection, projection, and union). Customized algorithms are developed for different recursion patterns. The termination condition for each algorithm has been studied in the context of both acyclic and cyclic databases †.

(2) **Stack-Directed Compilation of Complex Recursive Rules and Queries.**

[Hens 84] studied the compilation of recursive rules containing only one resolution cycle. We studied the compilation of recursive rules containing more than one resolution cycle. A stack-directed compilation technique has been developed for these recursive rules and queries. The method extends the compiled approach to more general recursion on relational databases.

(3) **Development of a Knowledge-Directed Compilation and Planning Mechanism.**

We have analyzed the problems in database oriented applications and developed a knowledge-directed compilation and planning technique as a solution to these problems.

† If no data item in a database derives itself in a series of joins with the relation itself, it is an acyclic database; otherwise, it is cyclic database.

3.2. ASSUMPTIONS AND DEFINITIONS

To systematically derive compilation strategies for recursive databases, the appropriate assumptions and definitions provide a formal framework for our discussion. We make the following assumptions: **the model assumption, the vector assumption, the single query literal assumption**, and **the variable pattern assumptions.** We also define some terms for classifying recursive rules and for later discussions.

3.2.1. Assumptions

Notational conventions:

(1) **Upper case letters** denote **relations** or **literals**, where R, S, and T indicate **virtual relations (intensional literals)**, while others indicate **base relations (extensional literals)**;

(2) **Lower case letters** denote **vectors** (a set of **relational attributes**), where a, b, c,... near the start of the alphabet indicate **constant vectors,** and u, v, .., z near the end of the alphabet indicate **non-constant vectors.**

(3) The **join** operation in this dissertation is denoted as $\bowtie$, or the power of a relation, for example, A^k, A^+, or A^*. However, it is interpreted differently from the natural join. For us, the join of R_1 and R_2 is the natural join of R_1 and R_2 followed by the projection on the non-joined attributes. More accurately, it should be called a **projected join**.

For example, R_1(x, y) $\bowtie$ R_2(y, z) is interpreted in the natural join as relation $R(x,y,z)$, but in our **(projected) join,** *the join attributes are projected out*, and the result is $R(x,z)$. The reason for this projection is that the join attribute is in general useless for coming iterations in the processing of recursive rules.

(4) The **union** operation is denoted with $\cup$, but it is also interpreted differently from the union operation in conventional set theory. Our union of $R_1(x,y,z,w)$ and R_2(y, z) is not the union of the tuples with different numbers of attributes, but the union of the two relations projected on the common attributes. More accurately, it should be called **projected union**.

For example, for multiple transitive closure looping rules,

$$R(x,z) :- B(x,y),R(y,z). \tag{3-7}$$

$$R(x,z) :- A_1(u,x,t,z). \tag{3-8}$$

$$R(x,z) :- A_2(x,y,z). \tag{3-9}$$

If we compile them into one formula, we may write it using the union operation,

$$B^*,(A_1 \cup A_2) \tag{3-10}$$

where $(A_1 \cup A_2)$ represents the projected union of A_1 and A_2, with unrelated attributes projected off. This is quite useful in the derivation of compiled formulas, especially for multiple exit rules.

Model assumptions: The database we discuss is a **first-order Horn database** (or **definite deductive database**) which contains two parts, (1) an **extensional database (EDB)** which consists a set of base relations, and (2) an **intensional database (IDB)** which consists of a set of Horn-clause specified deduction rules.

A Horn clause in IDB has the form

$$R :- P_1,...,P_k. \tag{3-11}$$

where the intensional literal R on the left side (the **consequent**) is a virtual relation which is defined by its right side (the **antecedent**), the conjunction of intensional or extensional literals $P_1, \ldots, P_k$. Each literal P or R may contain a list of variables, e.g.,

$$R(x_1, x_2, \ldots, x_n). \tag{3-12}$$

where x_i may be a variable or a constant. Each formula of a Horn clause must be a **safe** (or **range restricted**) formula, i.e., every variable in the literal on the left side must also appear in some literal on the right side.

The **model assumption** is the assumption which defines the problem domain of our research. It will not be relaxed, else our problem domain would be changed.

Vector assumption: An extensional literal in a rule may denote a relation obtained by applying finite relational operations to a sequence of base relations, and a variable in a relation may denote a vector of variables. For example, x may denote a vector consisting of x_1, x_2, x_3, and *A (x, y)* may denote

$$A(x,y) :- B(x,z), C(z,w), D(w,y).$$

The vector assumption enables our discussion to avoid irrelevant complexity. Extensions to full detail can be immediately and directly achieved, should they be required.

Simple query assumption: We assume that queries we discuss consist of a single query literal, with a single constant vector (called **the query constant**) and a single variable vector.

The typical recursive query studied is thus

$$?- R(a,z). \tag{3-13}$$

where a is a constant vector (the query constant), z is a variable vector, and R is a virtual relation defined recursively by some Horn clauses. A simple illustration of such a query is *"retrieve John's ancestors"*, which can be written as, *?− ancestor(john,Anc).*, in Prolog syntax [Cloc 81].

The simple query assumption is relaxed in Chapter 7 where some other forms of queries are studied.

Variable pattern assumptions: Here we make three related assumptions: **function-free assumption, constant-free assumption**, and **linear variable pattern assumption**.

(1) **Function-free assumption**: There is no evaluable function symbol among the argument terms of the recursive rules we study.

Evaluable function symbols in a recursive rule may lead to infinite relations. For example, for the rule $R(x+1) :- R(x).$, the process of generating the recursive relation may never terminate because it may determine a relation which contains all positive integers. Non-evaluable functions such as those often used to represent complex objects in logic programming, e.g.,

Employee(empno,name,age,education(school,degree,graduation_date)).

will not cause termination problems. However, complex database objects [Zani 85] are not considered in this thesis. We simply assume that there are no functions at all in our recursive rules.

Evaluable function symbols do appear in many practical problem solving applications, and the relaxation of this assumption will be discussed in Chapter 7.

(2) **Constant-free assumption:** There is no constant symbol among the argument terms of recursive rules.

If a recursive rule does contain constant symbols, we may remove them by performing selections and projections on the relations involved. The new rule, free of constant symbols, when applied to the new set of relations produced by the same operations, will generate the same results. Adopting this assumption simplifies discussion.

(3) **Linear variable pattern assumption**: We assume that the variable patterns in recursive rules are **linear variable patterns**, which have the following characteristics: (i) The antecedent relations form well-formed join expressions, i.e., the neighboring variables of adjacent relations are the same, and there are no shared variables among non-consecutive relations. (ii) Variables in the consequent literal correspond to the starting and ending variables of the antecedent, respectively. For example, the following rule manifests a linear variable pattern.

$$R(x,z) :- A(x,y),R(y,w),B(w,z). \qquad (3\text{-}14)$$

From now on, for rules and formulas manifesting a linear variable pattern, we simply do not write variables explicitly, because they are implied. For example, the above rule is written as

$$R :- A,R,B. \qquad (3\text{-}15)$$

Some variable patterns not having linear form can be transformed into linear pattern by changing the order of relations in the antecedent, changing the order of variables for each appearance of a relation of the rule, and/or projecting off variables not occurring in more than one literal. The transformation of variable patterns is discussed in the next section.

Clearly, some variable patterns can not be transformed into a linear pattern. For example, some variables may be shared among more than two literals, or not be shared among any antecedent literals at all. Our discussion is concentrated on linear patterns. Compilation for more complicated variable patterns is briefly mentioned in Chapter 6.

3.2.2. Some New Terms

Here we define terms for the classification of recursive rules.

(1) **Exit rule, looping rule,** and **indirect rule**.

A **rule** is a Horn-clause containing a non-empty consequent. An **intensional literal** is a literal defined by a rule, i.e., that occurs on the left side of a rule. An **exit rule** is a rule whose antecedent contains no intensional literals. A **looping rule** is a rule whose antecedent contains one or more intensional literals and each intensional literal is the same as the consequent literal. An **indirect rule** is a rule whose antecedent contains some intensional literal which is other than the consequent literal.

For example, in the rule set,

R :- $S,R.$ (3-16)

S :- $B,R,C.$ (3-17)

R :- $A.$ (3-18)

R :- $B,R,C,R,D.$ (3-19)

(3-19) is a looping rule, (3-18) is an exit rule, and (3-16) and (3-17) are indirect rules.

(2) **Mutual Recursion**.

Two predicates R and S are **mutually recursive** if the resolution of R involves the resolution of S, and conversely. For example, in the above rule set, R and S are mutually recursive. Many mutually recursive rules can be reduced to non-mutually recursive ones by simple transformation, which is discussed in Section 3.3.

(3) **Recursion level.**

Two recursive relations R and S are at the *same level* if the resolution of one rule relies on the resolution of the other, and conversely. R is said to be at a *lower level* than S if the resolution of S requires that of R, but the converse is not true. A **recursion level** consists of a set of mutually recursive relations $\{ R_1, R_2, ..., R_q \}$. For example, if we add more rules to the above rule set,

$$T :- R, A, S. \tag{3-20}$$

$$T :- A, T. \tag{3-21}$$

R and S are at the same recursion level, but they are at a lower level than T.

Because a call (resolution) to a lower level relation will never lead to a loop involving a higher level relation, the termination problem is essentially a one-level problem. Our discussion is therefore focused on one-level recursion.

(4) **Potential recursion point.**

A **potential recursion point** (PRP) in a rule is any intensional literal in the antecedent. It is called so because the resolution on this literal potentially contains recursions. For example, for the rule (3-20), R and S are potential recursion points. If there is only one potential recursion point in a rule, we call the rule a **single potential recursion point (SR)** rule, correspondingly for a **multiple potential recursion point (MR)** rule.

3.3. THE TRANSFORMATION OF RECURSIVE RULES

Rules may be in quite complex forms involving mutual recursion, redundancy, or irregularity. It is difficult to process rules in such complex forms to best advantage. We introduce a transformation to detect and remove redundant or conflicting rules, reduce some mutually recursive rules into rules which do not contain mutual recursion, and transform some complex variable patterns into linear patterns. With appropriate transformation, some rules in complex classes may be shifted down to simpler classes so that easier processing strategies can be applied.

Transformation is divided into three steps: (i) *indirect rule elimination*; (ii) *variable pattern transformation*; and (iii) *redundancy and conflict checking*.

3.3.1. Indirect Rule Elimination

An indirect rule in many cases can be reduced by simple resolution to non-recursive rules or looping rules. For example, the rules

$$S :- T,A. \tag{3-22}$$

$$T :- B,C. \tag{3-23}$$

can be transformed into an exit rule for literal S by simple resolution on T. The result is a rule which does not contain any recursion at all.

$$S :- B,C,A. \tag{3-24}$$

As another example, rules

R :- S,A. (3-25)

and

S :- R,B. (3-26)

can be transformed into

R :- R,B,A. (3-27)

for literal R, or

S :- S,A,B. (3-28)

for literal S, by one step resolution on S, or R, respectively. Thus we transform mutual recursion into non-mutual recursion.

Such a transformation eliminates some intensional literals though there may still be some in the remaining antecedents. Hence we call it **partial compilation**. Such partial compilation transforms indirect rules into looping rules and/or exit rules.

Mutual recursion has conventionally been considered a hard problem which can not be handled using conventional compilation techniques [Hens 84]. We feel that **mutual recursion** as such is not a good indicator of processing complexity, because some mutually recursive rules can easily be transformed into non-mutually recursive ones and be processed easily, while some non-mutually recursive rules require more complex processing than many mutually recursive ones. These more complex rules will be discussed in later chapters.

Next we develop a criterion for whether mutually recursive rules can be transformed into non-mutually recursive ones and show how to perform the transformation.

The antecedent of a mutually recursive rule contains intensional literals other than the consequent literal (i.e., it is an indirect rule). If these literals can be eliminated by partial compilation, we call it **reducible mutual recursion,** otherwise **irreducible mutual recursion.**

The following method can be used to judge whether a mutual recursion is reducible. Suppose we have two mutually recursive literals R and S. We first assume that the intensional literal R is instead a base relation and check whether the definition of S still contains the same level of recursion. If it does not, it means that S is fully defined on R, and/or some base relations, and/or some intensional relations which are defined on base relations, lower level recursions, or R only. Hence S can be eliminated by substituting for it its definition. Otherwise, if S still contains recursion, it cannot be eliminated by such substitution. This can be shown by an example.

Example 3.1. The reduction of mutual recursion to non-mutual recursion.

Suppose we have the following rule definitions.

$$R :- B. \tag{3-29}$$

$$R :- S,A,T. \tag{3-30}$$

$$S :- B,R,C. \tag{3-31}$$

$$S :- E. \tag{3-32}$$

$$T :- A,R. \tag{3-33}$$

By assuming R is not an intensional literal, the intensional literals S and T do not contain recursion, so S and T can be eliminated by partial compilation. After the partial compilation, the rule set becomes,

$$R :- B. \tag{3-34}$$

$$R :- B,R,C,A,A,R. \tag{3-35}$$

$$R :- E,A,A,R. \tag{3-36}$$

where mutual recursion has been eliminated. □

The transformation can be summarized in the following algorithm:

Algorithm 3.1. Elimination of Indirect Rules by Partial Compilation.

(1) The judgement of reducible transformation.

Under the assumption that the consequent literal on which the query is inquired is instead a base relation, we examine the intensional literals in the antecedent of the rule. If none of these intensional literals are recursively defined, they can be eliminated by partial compilation. If all the indirect rules defining this consequent literal can be eliminated, the mutual recursion is reducible.

(2) The partial compilation process.

Perform resolution on the reducible intensional literal(s) by substituting for each its definitions. The resolution continues until the consequent literal is encountered. □

This transformation transforms an indirect rule into a looping rule(s) and/or an exit rule(s). An exit rule is ready for processing using conventional relational database techniques. The processing of looping rules is the major topic of the following chapters.

3.3.2. Variable Pattern Transformation

It is hard to determine the best processing strategy for recursive rules with complex variable patterns. However, many complex variable patterns can be transformed into simpler ones, especially into linear patterns.

We may perform the following transformation on variable patterns.

(1) Eliminate those variables which are *irrelevant* to a recursion pattern. For example, in the rule

$$R(x,y) :- A(x,z,u), R(z,w), B(w,v,y). \tag{3-37}$$

the variables u and v are *irrelevant* [†], because they do not have any effect among literals in the antecedent or consequent. They should be eliminated in the recursion analysis.

(2) Swap the corresponding orders of literals and variables to form a well-formed join expression. For example, the rule

$$R(x,y) :- B(y,w), A(z,x), R(z,w). \tag{3-38}$$

can be transformed into the linear pattern,

$$R(x,y) :- A'(x,z), R(z,w), B'(w,y). \tag{3-39}$$

by (i) swapping columns of base relations $A(z,x)$ to $A'(x,z)$ and $B(y,w)$ to $B'(w,y)$; and (ii) adjusting the order of literals on the right side from B,A,R to A,R,B.

The swapping of the variables of a relation is equivalent to the swapping of the join column of a relation. For example, from A to A', the join of the first attribute of A' corresponds to the join of the second attribute of A. Notice that the swapping must be performed on all occurrences of a literal in a rule at the same time. For example, to swap z and w in $R(z,w)$, the corresponding variables x and y in $R(x,y)$ of the consequent in (3-38) must be swapped at the same time.

[†] This corresponds to anonymous variable in Prolog, represented by underscore "_".

The adjusting of the order of literals of the antecedent is based on the linkage of the variables within the consequent and the antecedent. For example, the transformation of B,A,R to A,R,B is based on the fact that (i) x in $A'(x,z)$ links to the variable x in the consequent $R(x,y)$, and z to the z in $R(z,w)$; and (ii) w in $R(z,w)$ links to the w in $B'(w,y)$, and y in $B'(w,y)$ links to y in the consequent $R(x,y)$.

3.3.3. Redundancy and Contradiction Checking

In practical applications, rules may contain redundancy and contradiction. If such redundancy and contradiction can be detected and removed, considerable processing cost can be saved, especially in cases of recursion.

(1) Elimination of redundant rules:

Some rules may have overlapped consequents and partially overlapped antecedents. This may cause redundant processing, which could be quite expensive when involving recursion. But the detection of such redundancy may not be straightforward. The transformation discussed in the last section may help the detection of such redundancy because after the transformation the relationships among rules are more explicit.

Weakest Precondition Principle : *If several rules have the same conclusion but overlapped preconditions, use the rule with the weakest precondition and eliminate the others.*

For example, for rules

$$R :- A,S,B. \qquad (3\text{-}40)$$

$$R :- A,S. \qquad (3\text{-}41)$$

(3-41) has weaker precondition than (3-40), so (3-40) should be eliminated.

The correctness of the weakest precondition principle is obvious from the principles of logic deduction and resolution [Chan 73].

(2) Dealing with contradictory rules:

Besides the redundancy check, a contradiction check is often necessary, because some new knowledge might be inconsistent with the existing rule bases and databases. There are several variations in dealing with contradiction: (i) simply reject new rules or data which contradict old; (ii) interactively communicate with a user or an expert to decide whether to abandon or modify new or old rules or data; or (iii) set up exception cases. The detailed discussion of these is another research field, viz. the field of *knowledge assimilation* [Kowa 79]. Our brief allusion to the problem here should not be taken as implying that it is simple.

Our assumption is that after such transformation processes, the rules and data in the database are non-redundant and consistent. Our later study concentrates on rules with linear variable patterns and without redundancy and inconsistency.

3.4. CLASSIFICATION OF RECURSIVE RULES ACCORDING TO THEIR PROCESSING COMPLEXITY

Based on the above assumptions, the transformed recursive rules are classified into five classes according to their processing complexity.

(1) Class 1 : **Transitive Closure Class.**

The primitive transitive closure rule set is written as follows,

$$R :- A. \qquad (3\text{-}42)$$

$$R :- A, R. \qquad (3\text{-}43)$$

The rule in the form of $R :- A,R.$ or $R :- R,A.$ is called a *transitive closure rule*, or a *TC rule*. It is a looping rule whose antecedent constitutes a *single* join expression (a one-sided join expression) which contains the join of the consequent relation literal with a vector of extensional relations.

We define the transitive closure class in a broader sense. It includes recursive rules involving only transitive closure processing strategies, which have the following cases,

(a) *a different exit expression*: The antecedent of the exit rule is different from the *induction base* (the extensional literal) of the TC rule, e.g., we have (3-44) instead of (3-42).

$$R :- B. \qquad (3\text{-}44)$$

(b) *multiple exit rules*: e.g., (3-44) and (3-45) instead of (3-42).

$$R :- C. \qquad (3\text{-}45)$$

(c) *multiple TC rules*:

$R :- A.$

$R :- B_1,R.$

... ...

$R :- B_n,R.$

$R :- R,C_1.$

... ...

$R :- R,C_k.$

The processing of multiple TC rules requires a minor modification of the primitive transitive closure algorithm.

(d) a special *SLMR (single looping rule with multiple recursion point) rule,* where the looping rule does not contain any extensional literal, e.g.,

$$R :- R,R, \cdots ,R. \tag{3-46}$$

which may be jointly defined with one or more exit rules. Such a recursive rule set can be processed with a transitive closure strategy.

The compilation and processing of the transitive closure class is discussed in Chapter 4.

(2) Class II : **SLSR** (Single Looping rule with Single potential Recursion point) **rules.**

A rule set in Class II contains a single looping rule with a single potential recursion point, and one or more exit rules. For example, (3-42) and the following rule

$$R :- B,R,C. \tag{3-47}$$

The processing of such a looping rule cannot be simply reduced to transitive closure. The compilation and processing of Class II queries are discussed in Chapter 5.

(3) Class III : **SLMR** (Single Looping rule with Multiple potential Recursion points) **rules.**

A rule set in Class III contains a single looping rule with multiple potential recursion points, and one or more exit rules. For example, (3-42) and the following rule

$$R :- B_1,R,B_2,R,B_3,R,B_4. \tag{3-48}$$

The method for compiling Class III rules is discussed in Chapter 6.

(4) Class IV : ML (Multiple Looping) **rules.**

A rule set in Class IV contains multiple looping rules. The general form of Class IV is:

R :− $A_{1.}$

. . . . . .

R :− $A_{p.}$

R :− $B_{11,}R, B_{12,}R, \ldots, B_{1i-1,}R, B_{1i.}$

R :− $B_{21,}R, B_{22,}R, \ldots, B_{2j-1,}R, B_{2j.}$

. . . . . .

R :− $B_{n1,}R, B_{n2,}R, \ldots, B_{nk-1,}R, B_{nk.}$

(5) Class V : IMR (Irreducible Mutually Recursive) **rules.**

Class V includes all irreducible mutually recursive rules, e.g.,

R :− $A_{1.}$

R :− $B_{1,}S, B_{2,}R, B_{3,}R, B_{4.}$

S :− $A_{2.}$

S :− $C_{1,}S, C_{2,}S, C_{3.}$

S :− $D_{1,}R, D_{2.}$

The methods for compiling rules in these two classes are similar to that of class III and are briefly mentioned in Chapter 6. Besides the recursive classes listed above, other forms of recursion, such as recursion involving multiple query constants and function symbols, are discussed in Chapter 7.

In our detailed discussion of various compilation algorithms in later chapters, We may observe an interesting evolution of our deductive compilation concept. For non-recursive rule sets (discussed in Chapter 2), query compilation is "pure" in the sense, that once a query is compiled, the database access plan is fully determined. For recursive rule sets of Classes I and II (discussed in Chapters 4 and 5), compilation is less "pure" in the sense that the database access plan cannot be fully determined before any database access occurs because the number of iterations (the termination point) is data dependent. For rule sets which need stack-directed compilation (discussed in Chapter 6), the compilation is even less "pure" in the sense that even the generation of compiled formulas is dependent on what is found in the extensional database. This observation may help us understand the derivation of different compilation algorithms for different recursion patterns. This is also the reason that we have called our compilation technique **pattern-based compilation**.

CHAPTER 4

TRANSITIVE CLOSURE CLASS

This chapter presents compilation and processing of queries using transitive closure rules. We study the issues of compiling transitive closure rules, evaluation strategies for general compiled formulas, and performance improvement techniques. We conclude that the δ Wavefront algorithm is the most promising of the algorithms studied. Based on this algorithm, we also discuss variations in the transitive closure class.

Transitive closure rules are in the simplest recursive rule class and are also the most frequently used in practical applications. For example, transitive relations (ancestor, friend, supervisor, part), AI inheritance relations (*is-a* and *has-a*), circular definitions, equivalences and some search problems (the path-finding, air-flight problem) involve transitive closure rules and may also involve large data relations. In fact, a scan of existing literature reveals that most recursive rules found in practical applications can be categorized into the transitive closure class or its variations.

4.1. THE DERIVATION OF A GENERAL COMPILED FORMULA FOR A TRANSITIVE CLOSURE RULE SET

Deduction rule resolution and the compilation method can be applied to derive the **general compiled formula** for a transitive closure rule. This formula is a compiled formula which represents a complete set of compilation results and guides the processing of the rule. Because the processing of a recursive rule often involves iteration, the formula usually contains closure or power operations.

Example 4.1. The compilation of a simple transitive closure rule set.

Consider the recursive rule set

$$R(x,z){:-}A(x,y),R(y,z). \tag{4-1}$$

$$R(x,z){:-}A(x,z). \tag{4-2}$$

where (4-1) is a looping rule and (4-2) is an exit rule. The recursive rule set may also be written in the clause form with positive and negative literals as,

$$R(x,z),\neg A(x,y),\neg R(y,z). \tag{4-3}$$

$$R(x,z),\neg A(x,z). \tag{4-4}$$

Because the scope of the variables is the Horn clause, (4-1) can be rewritten as:

$$R(y,z){:-}A(y,y'),R(y',z). \tag{4-5}$$

or,

$$R(y,z),\neg A(y,y'),\neg R(y',z). \tag{4-6}$$

(4-6) can be resolved with (4-3) by the **Robinson resolution principle**. We thus obtain,

$$R(x,z),\neg A(x,y),\neg A(y,y'),\neg R(y',z). \tag{4-7}$$

that is,

$$R(x,z){:-}A(x,y),A(y,y'),R(y',z). \tag{4-8}$$

The resolution process from (4-1) to (4-8) can be thought as a **call** of the looping rule. A **call** is a replacement of an intensional literal in *the calling rule* by the antecedent of *the called rule*. In the following discussion, we alternatively use the terms **call** and **resolution** to denote such a resolution process.

If we keep *calling* the non-exit rule, we generate formulas which contain a growing sequence of joins with relation A. Each formula so generated may have its intensional literals resolved by calling the corresponding exit rules. The formula thus generated contains extensional literals (base relations) only. These formulas, together with the first two formulas, the exit expressions and the call of the looping rule directly by its exit rule, determine a sequence of formulas. Because the intensional literals in the formulas are resolved by the deduction compilation technique, they are called **compiled formulas**.

In Example 4.1, the compiled formulas are

$A(x,z).$

$A(x,y),A(y,z).$

$A(x,y),A(y,y'),A(y',z).$

$A(x,y),A(y,y'),A(y',y''),A(y'',z).$

$\cdots\cdots$

According to the resolution principle, this set of compiled formulas forms the complete expansion of the recursive rule set (4-1) and (4-2), hence forms the complete set of resolution results.

The variable patterns of these formulas are in linear form. Thus the variables inside literals can be omitted, implicitly represented by the order of the literals (according to the convention in Section 3.2). Using transitive closure notation, the whole set of compiled formulas can be written in one formula:

$$A^{+}. \tag{4-9}$$

If the whole set of compilation results of a recursive rule can be simply written with a small set of compiled formulas, the formulas are called the **general compiled formula** of the recursive rule. The general compiled formula of recursive rule set (4-1) and (4-2) is A^{+}, the transitive closure of relation A.

If a recursive rule set can be compiled into a small number of general compiled formulas, the processing plan for the recursive rule set can be worked out by examining these formulas only. Next we study processing transitive closure rules using the general compiled formula A^+.

4.2. THE PROCESSING OF TRANSITIVE CLOSURES

Here we study the processing for the query

$$?-R(a,z). \qquad (4\text{-}10)$$

which can be stated as, *for query constant a, find all z's in the transitive closure relation R.* A simple example in this form is *retrieve all John's ancestors,* which can be written as, $?-Ancestor(John,z)$.

In studying this problem, both *efficient processing* and *termination* need to be taken into consideration. This chapter introduces several transitive closure algorithms: **the Basic-TC algorithm, the Wavefront algorithm, the δ wavefront algorithm,** and **the logarithmic algorithm**. Based on our analysis, we conclude that the δ wavefront algorithm with a tuple marking technique is one of the most promising techniques for processing transitive closure queries in databases.

4.2.1. The Basic-TC Algorithm

The idea of the Basic-TC algorithm is: (i) obtain the first set of answers by performing selection on the formula, i.e., *A(a, z)*, and add the set to the closure; (ii) incrementally append new answers into the closure where the new answers are derived by joining the closure with relation A on corresponding join columns; and (iii) terminate the process when the closure reaches a stable state.

The algorithm is stated as follows:

Algorithm 4.1 (Basic-TC): The Basic Transitive Closure Algorithm

(1) Data structure: two relations Old_Closure and New_Closure.

(2) Initialization:

Retrieve z into Old_Closure from $A(a,z)$ by performing selection using constant a on relation A.

(3) Iteration and Termination:

(i) Join Old_Closure with relation A to obtain New_Closure.

(ii) If New-closure is equivalent to Old_Closure, the transitive closure arrives at its stable stage and the iteration terminates.

(iii) Otherwise, union New_Closure and Old_Closure (with redundant tuples removed), and set Old_Closure as the result of the union. □

We'll prove that the algorithm terminates for both cyclic and non-cyclic data relations. We first distinguish a cyclic data relation from a non-cyclic one. If a tuple in a relation may traverse back to itself by following along its derivation path in closure derivations, we say that it is a **cyclic tuple** and the relation contains cycles. For example, both (a, b) and (b, a) are cyclic tuples in the data relation { (a, b), (b, a) }. If there is some cyclic tuple in a data relation, the relation is a **cyclic data relation**, otherwise it is a **non-cyclic data relation**.

Theorem 4.1. Algorithm Basic-TC derives complete answers and terminates for both cyclic and non-cyclic data relations.

Proof Sketch

(1) Completeness.

The algorithm is derived according to the definition of a transitive closure. At each iteration, the processing uses the accumulated closure results to derive the new one by joining the base relation with the existing closure. According to the completeness of the deductive resolution, the complete set of answers are derived from such operations. When the closure does not grow at some iteration, the further joining of the same closure with the same base relation must produce the same closure again. Hence that is the point we find the complete set of answers.

(2) Termination for both cyclic and non-cyclic data relations.

It is obvious that the processing terminates for a non-cyclic data relation when no new tuple is generated at some iteration. For a cyclic database, suppose that there is a set of cyclic tuples a_1, a_2, ..., a_k with cycle length k in the transitive closure relation A. For any two tuples a_i and a_{i+1} where a_i derives a_{i+1} by a join operation, if a_i is included in the closure, but a_{i+1} is not, the next iteration must generate the new tuple a_{i+1}. When no new tuple can be generated at some iteration, it must be that every tuple in this cycle is already in the closure. Further iteration will not generate any new tuples in this cycle. □

Algorithm Basic-TC is not a completely naive algorithm, because it performs selection first using the query constant, which reduces the size of intermediate relations, and it uses the accumulated results (Old_Closure) in deriving new results rather than starting the processing of each compiled formula from scratch.

But there is obvious redundancy in the algorithm. Suppose the first iteration generates tuples b, c, d from tuple a which is in the closure. The union of New_Closure and Old_Closure becomes a, b, c, d. According to the algorithm, a is still being used in the second iteration, but obviously no new tuples can be derived from a, and during later iterations, the process will repeatedly append b,

Iteration No.	Old_Closure	New_Closure	Unioned Closure
1	a	b, c, d	a, b, c, d
2	a, b, c, d	b, c, d, ...	a b, c, d, ...

Table 4.1. The Processing Redundancy in Algorithm Basic-TC

c, d into the closure, which is clearly unnecessary processing. This is shown in Table 4.1.

Obviously, the driver tuples used in an iteration to derive new tuples can not generate any new tuples in later iterations; hence they should not be included in future iterations. A modification to the Algorithm Basic-TC obtains a new algorithm, **TC-Wavefront.**

4.2.2. The TC-Wavefront Algorithm

If we apply a temporary **wavefront** relation *WAVE* to hold the result tuples of the last iteration and derive new tuples for the next iteration, the redundant regeneration of tuples derived in previous iterations can be avoided. TC-Wavefront uses only the results of the last iteration in deriving new results. It is stated as follows:

Algorithm 4.2. (TC-Wavefront): The Transitive Closure Wavefront Algorithm

(1) Data structures: temporary relations, WAVE, NEW-WAVE and RESULT.

(2) Initialization:

Retrieve z from relation A by performing selection on a, i.e., retrieve values of z that satisfy $A(a, z)$, and put the result tuples into RESULT and also into WAVE;

(3) Iteration:

Join A with relation WAVE, set result relation as NEW-WAVE,
append tuples in NEW-WAVE into RESULT with redundant tuples removed, and
set WAVE to NEW-WAVE;

(4) Termination:

Terminate when WAVE goes to empty. For cyclic data relations, terminate also when RESULT does not grow at some iteration. □

Theorem 4.2: Algorithm TC-Wavefront generates complete answers and terminates for both noncyclic and cyclic data relations.

Proof Sketch

(1) Completeness.

Since the join of the same set of tuples with the same data relation will generate the same set of answers, a used set of tuples cannot generate new answers when joining with the same data relation. Thus the excluding of the used tuples will not change the completeness of the solution. Because the algorithm is an execution of the compiled formulas derived from the deductive resolution in accordance with the completeness of resolution, the algorithm is complete.

(2) Termination.

The process terminates when WAVE goes to empty. This is because at each iteration we take WAVE as an operand for the join operation. If WAVE is empty, since the join of an empty relation generates an empty relation, no new RESULT tuples are generated. The closure is stable, and thus meets the termination condition.

For cyclic data relations, WAVE may not be empty, even when RESULT does not grow. At this point the tuples in WAVE are tuples previously used for generating tuples in RESULT. If each tuple in WAVE has been used previously, the joins of WAVE with relation A cannot generate any new tuples. Stability is reached and the process terminates. □

Algorithm TC-Wavefront is an improvement over Basic-TC because it reduces redundant processing by including only tuples generated in New-Closure. But there is still some redundancy in the processing. The problem can be illustrated by continuing our previous example. For Algorithm TC-Wavefront, the second iteration will use *b, c, d* but not *a* to generate new tuples. Suppose *e, f, g, a, b* are obtained at this iteration and put into the new WAVE. If tuple *b* is generated by a tuple other than *b*, it is a redundant tuple, because

Iteration No.	WAVE	New-WAVE	RESULT
1	a	b, c, d	a, b, c, d
2	b, c, d	e, f, g, a, b	a, b, c, d, e, f, g
3	e, f, g, a, b	b, c, d, . . .	a, b, c, d, e, f, g, ...

Table 4.2. Redundancy Still Exists in Algorithm TC-Wavefront

the further processing on b will not generate any new tuples. If tuple *b* is generated by the same tuple *b* , *b* is a cyclic tuple and the further processing of the cyclic tuple is also redundant. For the same reason, cyclic tuple *a* should not be processed further. This is presented in Table 4.2.

From this we can see that further improvement is needed.

4.2.3. Our Preference: The δ Wavefront Algorithm

Here we propose another algorithm, Algorithm δ Wavefront (δW). The idea is that at each iteration we check each newly generated tuple against the tuples in RESULT. If it is already in RESULT, we do not include it in NEW-WAVE. This modification further reduces redundancy, and the processing terminates when WAVE becomes empty even for cyclic data relations. The algorithm is named Algorithm δ Wavefront.

Algorithm 4.3 (δW) : The δ Wavefront Algorithm for Processing Transitive Closure

(1) Data structures: two temporary relations RESULT and WAVE.

(2) Initialization :

(i) perform selection on relation *A* using the constant vector *a* provided by the query; and

(ii) put the selected results into RESULT and also into WAVE.

(3) Iteration: (use WAVE to generate new results and new WAVEs iteratively)

(i) join WAVE and relation *A* on the corresponding join attributes; and

(ii) append to RESULT the result tuples that are **not** in RESULT, and form the new WAVE.

(4) Termination: terminate when WAVE becomes empty. □

Theorem 4.3. Algorithm δ Wavefront generates complete answers and terminates for both noncyclic and cyclic data relations.

Proof Sketch

(1) Completeness.

A tuple which has been used in previous iteration will not contribute any new tuples to the result if it is used again. Its removal does not alter the completeness of the solution. Because the algorithm modifies the previous one by removing only the tuples from WAVE that have been used previously, it still possesses the completeness of the previous algorithm.

(2) Termination.

For a non-cyclic database, because of the finiteness of the database, the infinite traversing of the same data relation along a derivation path is impossible. There must be a point along all traversing paths at which no new tuples will be generated. Thus at some iteration, there will be no tuple that can contribute to WAVE any more, and WAVE becomes empty. No new tuple can be generated by joining an empty relation. Hence the process terminates.

For cyclic databases, a cyclic tuple is deleted from WAVE if it is already in RESULT. WAVE thus goes to empty for the same reason as for the non-cyclic case. Thus the process terminates when WAVE becomes empty. □

For the same example relation, the process using Algorithm δ Wavefront is presented in Table 4.3.

Algorithm δ Wavefront has two advantages:

Iteration No.	WAVE	New-WAVE	RESULT
1	a	b, c, d	a, b, c, d
2	b, c, d	e, f, g	a, b, c, d, e, f, g
3	e, f, g,	. . .	a, b, c, d, e, f, g, ...

Table 4.3. Redundancy Eliminated in Algorithm δ Wavefront

(1) Redundancy has been eliminated. The re-processing of already processed tuples is avoided.

(2) Termination judgement is simplified for cyclic databases. Algorithm δ Wavefront terminates for cyclic databases by simply checking whether WAVE is empty. This is obviously more efficient than the method of comparing two large relations Old_Closure and New_Closure in Algorithm Basic-TC, and of checking whether every tuple in WAVE is already in RESULT in Algorithm TC-Wavefront.

4.3. THE PERFORMANCE IMPROVEMENT: IMPLEMENTATION CONSIDERATIONS

Performance in processing transitive closures can be improved using various techniques. Here are three of them: **the tuple marking technique, performing iterative joins on join indices, and the logarithmic algorithm for deriving the transitive closure of the entire relation.**

4.3.1. A Tuple Marking Technique

There are two temporary relations, RESULT and WAVE, involved in Algorithm δ Wavefront. The RESULT relation accumulates the transitive closure results which may grow quite large. The checking of each tuple generated in WAVE against RESULT and the appending of new tuples into RESULT at each iteration may be expensive. A tuple marking technique can be used to avoid the generation of the large RESULT relation.

The tuple marking technique works as follows: instead of using a temporary relation RESULT to hold result tuples, a marking bit is used for each tuple to identify whether it is in the closure. Initially, the marking bit of each tuple in the data relation is set *off.* At each iteration, some new set of answers is derived according to the algorithm. If a tuple in the database corresponding to the generated set has not been previously marked *on*, mark it *on* and put the answer into WAVE for generating new answers. That is, only the answers corresponding to the *newly* marked tuples are included in WAVE. The process terminates when no tuple can now be marked, i.e., WAVE becomes empty. The answers to the original query can be obtained from the marked tuples in the database, for example, by projection on the appropriate attributes.

Tuple marking needs an extra bit for marking but it saves the generation of large data relations. The marking bits do not need to be saved into the database, but instead we may use a bit-vector to keep them in main memory for fast access. Because each tuple takes only one bit, a page, say of 1000 16-bit words, may hold bits for 16k tuples. This works fine for medium-sized data relations. For large data relations, for example, one containing a million tuples, advanced techniques similar to the filtering mechanism used in differential files [Seve 76] need to be developed for more efficient processing.

4.3.2. Performing Iterative Joins on Join Indices

An obvious saving can be obtained by utilizing indices in iterative joins. Indices have been widely used in relational query processing. They will be more important and beneficial for processing recursive queries, because in this case join operations are often performed on the same attributes of the same relations iteratively. Index files on those join attributes can be built up once and utilized repeatedly.

There are two methods for using indices. One is to use conventional indices on the join attributes of one of the relations being joined. Another is the use of join index files. In either method, the iterative joins can be performed on smaller index files, instead of on the complete data relations.

The use of individual index files of joined relations is popular in conventional relational query processing. An index file is built for the join attribute of the iteratively joined relation. Iterative joins are performed on the index file.

The join index method was studied by Haerder [Haer 78] and Valduriez [Vald 85]. A join index is an abstraction of the join of two relations. It is a binary relation that directly represents the linkage of tuples of two relations in a database. Because the length of an index is considerably shorter than that of a tuple, performing iterative joins on join index files may considerably reduce CPU and I/O costs.

For example, the join of relation R and S on attributes R.A and S.B may be iteratively executed in deriving transitive closure. A join index file for the join attribute can be simply represented as

$$JI = (r_i, s_j) \mid r_i.A = s_j.B \qquad (4\text{-}11)$$

where r_i, s_j are surrogates which are system generated identifiers that never change. The size of the join index depends on the join selectivity factor

$$JS = |R \bowtie S| / |R| * |S| . \tag{4-12}$$

where |R| is number of tuples of the relation R, and likewise for S. If the join has high selectivity (JS is low), the join index is small. Otherwise, it can be close to the Cartesian product, making the join index quite large. Similar to other index files, a join index file may be stored using database storage structures such as B-trees.

Performing iterative joins on indices by using either index files or join index files is obviously superior to methods which perform iterative joins on whole data relations. The choice between using (individual) index files and using join index files is determined by the join selectivity (JS) of the two relations on join attributes. When JS is small, the join index file (JI) may be comparable in size to index files, and the utilization of join indices in iterative joins obviously saves processing. But when JS is large, the JI file may be quite large, and performing joins on JI files will be more expensive than performing joins on smaller index files.

4.3.3. A Logarithmic Algorithm for Deriving the Entire Transitive Closure

The generation of the entire transitive closure, if it is not impossibly large, and saving it for later selection operations might be a good alternative to deriving a partial transitive closure for each query. Also, some queries require finding the entire transitive closure anyway.

The generation of the entire transitive closure may adopt a more efficient algorithm, a logarithmic algorithm, developed by [Vald 85b]. The idea of the algorithm is as follows: instead of using a linear iteration to progressively derive a transitive closure, a logarithmic-style of processing is used.

Algorithm 4.4. (Logarithmic): Derivation of the Entire Transitive Closure Using A Logarithmic Algorithm [Vald 85b].

(1) Data structure: three temporary relations: RESULT, WAVE, and LOG_WAVE.

(2) Initialization:

Set the base relation $A(x,z)$ as LOG_WAVE and also as RESULT;

(3) Iteration:

(i) Join LOG_WAVE with LOG_WAVE to derive new LOG_WAVE,

(ii) join RESULT with LOG_WAVE to derive WAVE, and

(iii) union RESULT, WAVE and LOG_WAVE to get new RESULT;

(4) Termination:

Terminate when LOG_WAVE or WAVE is empty; and for cyclic data relations, terminate also when RESULT does not grow at some iteration. □

The process is logarithmic. This is shown by the following analysis.

(1) At the initial time LOG_WAVE is relation $A(x,z)$.

(2) At the first iteration, LOG_WAVE becomes $A \bowtie A$, written as A^2; the join of RESULT, which is A, with LOG_WAVE (A^2) derives WAVE, A^3; and the union of RESULT, WAVE and LOG_WAVE derives the new RESULT, which is $A \cup A^2 \cup A^3$.

(3) At the second iteration, LOG_WAVE becomes $A^2 \bowtie A^2$, written as A^4; the join of RESULT, which is $A \cup A^2 \cup A^3$, with LOG_WAVE (A^4) derives WAVE, $A^5 \cup A^6 \cup A^7$; and the union of RESULT, WAVE and LOG_WAVE derives the new RESULT, $A \cup A^2 \cup A^3 \cup A^4 \cup A^5 \cup A^6 \cup A^7$;

(4) At the i-th iteration, LOG_WAVE, WAVE, and RESULT becomes

$$LOG\_WAVE = A^{2^{i-1}} \bowtie A^{2^{i-1}} = A^{2^i} \tag{4-13}$$

$$WAVE = RESULT \bowtie LOG\_WAVE$$

$$= \bigcup_{k=1}^{2^i-1} A^k \bowtie A^{2^i}$$

$$= \bigcup_{k=2^i+1}^{2^{i+1}-1} A^k \tag{4-14}$$

$$RESULT = RESULT \cup WAVE \cup LOG\_WAVE$$

$$= \bigcup_{k=1}^{2^{i+1}-1} A^k \tag{4-15}$$

At the i-th iteration the algorithm generates results which need 2^{i+1} iterations in conventional (linear) join algorithms.

Using reasoning similar to the proof in Theorem 4.1, we can prove that the termination condition in the algorithm works for both acyclic and cyclic databases. The detailed proof is omitted here.

The logarithmic algorithm works for the derivation of the entire transitive closure, but it is not suitable for the derivation of partial transitive closures. The reason is that, if we start with the selected tuples and ignore the other tuples in deriving LOG_WAVE, since the progress of the WAVE may reference other tuples which are not available in LOG_WAVE, such tuples cannot be included in processing, so many desired answers will be missing.

4.4. ALGORITHMS FOR PROCESSING VARIATIONS OF THE TRANSITIVE CLOSURE CLASS

In our recursive rule classification (Section 3.4), we listed the other cases of the transitive closure class: (a) the exit rule antecedent (called **exit expression**) being different from the extensional literal in the TC rule, (b) multiple exit rules, (c) multiple TC rules, and (d) a special SLMR rule.

In the following analysis, we omit the compilation process because of its similarity to that in Section 4.1. We simply present the general compiled formulas and study the processing plans, which are minor modifications of the processing plans we have presented above. In the following discussion, Algorithm δ Wavefront is the base for the derivation of the new algorithms. The new processing algorithms lead to reasoning similar to that presented above for Algorithm δ Wavefront, so we do not present explicit proofs for them.

4.4.1. A Different Exit Expression

In a transitive closure rule set, the exit expression may be different from the extensional literal in the TC rule. For example, we may have the following rule set,

$$R(x,z) :- A(x,y), R(y,z). \tag{4-16}$$

$$R(x,z) :- B(x,z). \tag{4-17}$$

This is the case of Class I(a) (Section 3.4). Using the compilation method, the general compiled formula can be easily derived:

$$A^*B \tag{4-18}$$

Note that in this case the processing may start from two different directions: first process A^* then the exit expression B; or first process the exit expression B then A^*. The two processes are asymmetric. The former processing direction is best when the query is *?- R(a, z).*, while the latter is best when the query is ?- R(x, a). or the looping rule is defined differently, i.e., $R :- R, A$. With rules fixed, the processing direction is decided by query constant location.

Algorithm 4.5. (δWA): The Processing of A^*B (A Different Exit Expression).

(1) If the process starts from B, the algorithm is the same as δW except it is initialized by performing selection on relation B instead of on A.

(2) If the process starts from A, we modify Algorithm δW as follows:

(i) derive the partial transitive closure A^+ using Algorithm δW;

(ii) obtain solutions by unioning the results of the selection on B and the join of the transitive closure A^+ with B. □

4.4.2. Multiple Exit Rules

There are often multiple exit rules in a recursive rule set. For example, we may have (4-16), (4-17) and the following as our rule set.

$$R(x,y) :- \; C(x,y). \tag{4-19}$$

This forms the Class I(b) case (multiple exit rules) which has the general compiled formula

$$A^*(B \cup C). \tag{4-20}$$

The processing of such a formula is a minor modification of Algorithm δW.

Algorithm 4.6. (δWB): The Processing of $A^*(B \cup C)$ (Multiple Exit Rules)

(1) If the process starts from $(B \cup C)$, the algorithm is the same as Algorithm δW except at the initialization: (i) perform selection on both relations B and C, and (ii) take the union of the selected results as RESULT and WAVE;

(2) If the process starts from A, the processing should be,

(i) derive the partial transitive closure A^+ using Algorithm δW;

(ii) obtain solutions by unioning the two results (a) selection on $B \cup C$, and (b) the join of the transitive closure A^+ with $(B \cup C)$. □

The above discussion uses quite simple exit rules. In more general cases, exit rules may not be so simple and well-formed. However, they may be transformed into the simple form.

(1) Some exit expressions may be joins of more than one base relation, for example,

$$R(x,z) :- A(x,y),B(y,z). \tag{4-21}$$

This can be simply transformed into a well-formed exit expression, such as

$$R(x,z) :- E(x,z). \tag{4-22}$$

by taking E as the join of A and B

$$E(x,z) :- A(x,y),B(y,z). \tag{4-23}$$

(2) Some exit expressions may contain a different number of variables from that of the consequent literal. For example,

$$R(x,z) :- A(x,y,w),C(w,y,u,z). \tag{4-24}$$

This can be simply transformed into the standard form, by taking the join on the join attributes and projecting on the attributes which are used in the consequent literal.

(3) Some exit expressions may contain non-recursive intensional literals.

$$R(x,z) :- A(x,y),S(y,z). \tag{4-25}$$

$$S(y,z) :- B(y,w,u),C(w,z),D(u). \tag{4-26}$$

This can be reduced to case (2) by simple resolution, in this case, resolving on the literal S.

If we compare the processing for multiple exit rules (Algorithm δWB) and that for a single exit rule (Algorithm δWA), we find that the only difference is that the former one contains a union operation $B \cup C$, while the latter one does not. If we substitute $B \cup C$ for the literal B in Algorithm δWA, the algorithm is automatically switched to Algorithm δWB.

In fact, for any complex transformation, if we have more than one exit expression, we may use a union operation which simplifies discussion. For example, if we have $E1,R,E_2$, where E_1 and E_2 may be any complex expressions and R is going to be resolved with exit rules R :- B_1. and R :- B_2. The result, $E_1,(B_1 \cup B_2),E_2.$, is obviously simpler than

$$(E_1,B_1,E_2) \cup (E_1,B_2,E2).$$

In this case, the two exit rules may be better thought of as one, R :- $B_1;B_2$.

From the above discussion, we can derive a multiple exit rule principle.

Multiple Exit Rule Principle: *Multiple exit rules can be treated as a single exit rule in the derivation of a recursive rule processing plan. The exit rules are simplified by relational operations and resolutions, and unioned into one by union operations.*

The **multiple exit rule principle** is quite useful in the discussion of recursive rule compilation. In the following study, only a single exit rule is considered. Multiple exit rules can be immediately handled by applying this principle.

4.4.3. Multiple Transitive Closure Rules

The general case of a multiple transitive closure (TC) looping rule set (Class I(c))

$R :- A.$

$R :- B_1, R.$

$\dots \quad \dots$

$R :- B_n, R.$

$R :- R, C_1.$

$\dots \quad \dots$

$R :- R, C_k.$

can be compiled into

$$(\prod_{i=1}^{n} B_i^{\,*})A(\prod_{j=1}^{k} C_j^{\,*}) \tag{4-27}$$

The processing strategy can be derived from the formula, according to the discussion in this chapter. Here we discuss a special and more frequently used case, which is the rule set containing two transitive closure rules with extensional literals joining with different variables of the recursive literal.

$$R :- A. \tag{4-28}$$

$$R :- B, R. \tag{4-29}$$

$$R :- R, C. \tag{4-30}$$

This rule set can be compiled into

$$B^* A C^* . \tag{4-31}$$

The processing may proceed in either direction: from B toward A and C, or from C toward A and B. The selection of the processing direction is decided by the query constant, as we discussed before. Suppose the processing direction is from B to the right. We have the following algorithm. (The algorithm for the other direction can be stated similarly.)

Algorithm 4.7. (δWC): The Processing of B^*AC^* (Multiple Looping Rules)

(1) Derive the partial transitive closure B^+ using Algorithm δ Wavefront.

(2) Join the B^+ with relation A, and union it with the result of selection on A; and

(3) Set the result obtained in step (2) as RESULT and WAVE, and the rest of the processing is the same as finding the partial transitive closure of C. □

4.4.4. A Special Multiple Potential Recursion Point Rule

There is a special multiple potential recursion point rule set (Class I(d)), which can be handled as a transitive closure class. The rule set can be written as

$$R :- R, R, \cdots, R. \tag{4-32}$$

$$R :- A. \tag{4-33}$$

Such a rule set can be seen in practice in the definition of a "part", where

$$Part(x,z) :- Part(x,y), Part(y,z). \tag{4-34}$$

If there are two intensional literals in the antecedent, the rule set can be easily compiled into

$$A^+. \tag{4-35}$$

Its processing strategy is the same as that for the basic transitive closure types discussed already.

If there are n intensional literals in the antecedent, the rule set can be compiled into

$$A \cup A^{n+k(n-1)}. \tag{4-36}$$

where k is an integer that goes from 0 up to the termination point. The processing strategy can also be derived according to the discussion in this chapter.

4.5. SUMMARY : PROCESSING TRANSITIVE CLOSURE QUERIES

Rules in transitive closure classes can be compiled into general compiled formulas and processed iteratively using relational query processors. To improve processing efficiency, our analysis suggests use of the following heuristics:

(1) Perform selection first, using query constants;

(2) Utilize a *wavefront* of newly generated tuples;

(3) Remove from the wavefront tuples which were placed in the accumulating closure in previous iterations. This will eliminate redundant processing and facilitate processing data relations containing cyclic tuples;

(4) Perform iterative processing only on indices rather than on data tuples. We may use either join index files or individual index files, depending on the join selectivities;

(5) Apply a logarithmic algorithm to reduce the number of joins performed in deriving the entire transitive closure.

In practice, the processing of transitive closure queries involving functional definitions and multiple query constants is more frequently met than that of the cases we have discussed here. These more complex cases will be studied in Chapter 7.

CHAPTER 5

THE COMPILATION AND PROCESSING OF SLSR RULES

This chapter presents the compilation and processing of Class II: SLSR rules (rule sets with a Single Looping rule with a Single Recursion point). We first compile an SLSR rule set into a general compiled formula, then analyze four algorithms for the evaluation of the compiled formula: **Natural Evaluation (NE)**, **Single Wavefront (SW)**, **Double Wavefront (DW)**, and **Central Wavefront (CW)**. Our performance evaluation concludes that **performing selection first** and **using previous processing results (wavefronts)** are two important heuristics in the efficient processing of compiled formulas. We call the three wavefront algorithms **SLSR wavefront algorithms**.

5.1. The General Compiled Formula for an SLSR Rule Set

An SLSR rule set consists of one or more **exit rules** and one **looping rule with a single recursion point**. The typical SLSR rule set studied here is

$$R(x,z):-A(x,z). \qquad (5\text{-}1)$$

$$R(x,z):-B(x,y),R(y,w),C(w,z). \qquad (5\text{-}2)$$

where A, B and C are base relations, R is a recursively defined virtual relation, and x, y, z and w are vectors of variables.

The recursive query studied is

$$?-R(a,z). \qquad (5\text{-}3)$$

where a is a constant vector and z is a variable vector to be retrieved.

Similar to the compilation of transitive closure rules, the general compiled formula for an SLSR rule set can be derived using the deductive compilation method.

An SLSR rule set is a more general case of recursion than a transitive closure class. The looping rule of an SLSR rule set consists of two base relations, one on each side of the recursive literal. If there were one base relation missing in the looping rule, the SLSR rule set would be reduced to a transitive closure class. The first thing to try is to make SLSR rules fit into the frame of the available transitive closure algorithms. This can be done by permuting predicates R and C to group B and C together as one base relation BC, and deriving the transitive closure of BC. The formula could be written as

$$R(x,z):-BC(x,y,w,z),R(y,w). \qquad (5\text{-}4)$$

where BC is obtained by

$$BC(x,y,w,z):-B(x,y),C(w,z). \qquad (5\text{-}5)$$

Notice that the relation BC involves the cross-product of two unrestricted data relations B and C. Moreover, the further call will obtain

$$R(x,z):-BC(x,y,w,z),BC(y,y',w',w),R(y',w'). \qquad (5\text{-}6)$$

which involves joins of the cross-product relation BC, something requiring prohibitively inefficient processing. Thus the processing of an SLSR rule set must be with something other than transitive closure algorithms. The relations which share join attributes should be processed together. In the SLSR rule set (5-1) and (5-2), using step-wise recursive calls on rule (5-2), we obtain a sequence of expansions

$$R(x,z):-B(x,y_1),B(y_1,y),R(y,w),C(w,w_1),C(w_1,z). \qquad (5\text{-}7)$$

......

$$R(x,z):-B(x,y_k),\cdots,B(y_1,y),R(y,w),C(w,w_1),\ldots,C(w_k,z). \qquad (5\text{-}8)$$

where y_1 , ..., y_k and w_1 , ..., w_k are vectors of variables.

Using the exit rule (5-1) on the above sequence, the solutions are obtained by processing the following sequence of formulas and unioning the results.

$$A(a,z). \tag{5-9}$$

$$B(a,y),A(y,w),C(w,z). \tag{5-10}$$

$$B(a,y_1),B(y_1,y),A(y,w),C(w,w_1),C(w_1,z). \tag{5-11}$$

......

$$B(a,y_k),B(y_k,y_{k-1}),\ldots,B(y_1,y),A(y,w),$$
$$C(w,w_1),\ldots,C(w_{k-1},w_k),C(w_k,z). \tag{5-12}$$

The expansion sequence terminates when no new solution is found in the database. The sequence can be represented with a **general compiled formula**

$$\sigma_a B^k \bowtie A \bowtie C^k \tag{5-13}$$

where σ_a B means the selection of a on the corresponding attribute of relation B, $\bowtie$ means the join on the corresponding attributes, and B^k means the joins formed by k B relations. Join attributes are omitted in the formula for simplicity, and the range of index k is from 0 to n where n is the number of iterations up to the termination point. When no ambiguity results, we will simply write

$$\sigma B^k A C^k \tag{5-14}$$

for the general compiled formula.

5.2. THE PROCESSING OF THE SLSR COMPILED FORMULA IN RELATIONAL DATABASES

The general compiled formula $\sigma B^k A C^k$ for SLSR rule set can be evaluated using different ordering and grouping strategies. We develop and compare performance of four evaluation algorithms: **NE (Natural Evaluation), SW (Single**

Wavefront), CW (Central Wavefront), and DW (Double Wavefront).

The simplest algorithm we can think of is a naive evaluation algorithm which evaluates the compiled formula from scratch at each iteration. No inter-query grouping and sharing are performed. The execution plan can be written in a sequence as (5-15), and the flow of processing is shown in Figure 5.1.

$$\sigma B \bowtie B \cdots \bowtie B \bowtie A \bowtie C \bowtie \cdots C \bowtie C. \qquad (5\text{-}15)$$

At the k-th iteration, the formula contains k B's to the left of A and k C's to the right of A. The processing is: (1) perform selection on relation A to get σA and on relation B to get σB; (2) starting with the selection result σB, join corresponding attributes on relation B k-1 times to get $\sigma\ B^k$; (3) join with relation A once to get $\sigma\ B^k$ A; and (4) join with relation C k times to obtain the final result $\sigma B^k A C^k$. The total number of joins for all iterations is

$$\sum_{n=0}^{k} (2n) = k^2 + k.$$

Because the same relations B, A and C are used for joins at each iteration, it will be beneficial to build up indices on the join columns of these relations or

$$\begin{array}{lr}
n = 0 & \sigma A \\
n = 1 & \sigma B \bowtie A \bowtie C \\
n = 2 & \sigma B \bowtie B \bowtie A \bowtie C \bowtie C \\
n = 3 & \sigma B \bowtie B \bowtie B \bowtie A \bowtie C \bowtie C \bowtie C \\
\ldots & \cdots\cdots \\
n = k & \sigma B \bowtie \ldots \bowtie B \bowtie A \bowtie C \bowtie \ldots \bowtie C
\end{array}$$

Figure 5.1. The Processing Flow of Algorithm NE (Natural Evaluation)

join indices on them.

The NE algorithm does not save intermediate results for later iterations. The only heuristic used is to perform selection first when possible.

5.2.1. The Single Wavefront Algorithm

If the intermediate result, $\sigma\ B^{k-1}$, is saved for the coming k-th iteration, some redundancy can be avoided. The saved result behaves like a **wavefront** to derive a new wavefront $\sigma\ B^k$. Such saving gives us Algorithm **Single Wavefront**. It preserves one redeeming feature of Algorithm NE: performing selection first to reduce the size of the relation to be joined. The processing plan is illustrated in Figure 5.2 and can be simply illustrated as follows, where the parentheses show the grouping of the operations.

$$(\sigma B^{k-1} \bowtie B) \bowtie A \bowtie C \bowtie \cdots C \bowtie C. \tag{5-16}$$

At the k-th iteration, using the result $\sigma\ B^{k-1}$ which was saved as the old wavefront, perform join with relation B one more time to get $\sigma\ B^k$, which is in turn

n = 0	σA
n = 1	(σB) $\bowtie$ A $\bowtie$ C
n = 2	(σB $\bowtie$ B) $\bowtie$ A $\bowtie$ C $\bowtie$ C
n = 3	(σB^2 $\bowtie$ B) $\bowtie$ A $\bowtie$ C $\bowtie$ C $\bowtie$ C
...	. . . . . .
n = k	(σB^{k-1} $\bowtie$ B) $\bowtie$ A $\bowtie$ C $\bowtie$ . . . $\bowtie$ C

Figure 5.2. The Processing Flow of Algorithm SW (Single Wavefront)

saved for the next iteration. Then perform one A-join and k C-joins. The total number of joins for all iterations is

$$\sum_{n=2}^{k} (n+2)+2 = (k^2+5k)/2-1$$

which gives fewer joins than required for the NE algorithm.

To speed up the processing, it is beneficial to build up index files on join columns for relations B, A and C, or build up join index files for them when necessary. Because of its saving of previous processing, Algorithm SW should be expected to perform better than Algorithm NE.

Algorithm 5.1 (SW) : The Single Wavefront Algorithm for Processing $\sigma B^k AC^k$

(1) Data structures :

(i) an iteration number IterNum to register the number of iterations performed;

(ii) a wavefront relation WAVE; and

(iii) a buffer relation BUFF which contains two attributes, driver tuple identifier and derived result identifier, and which is used in determining the termination for cyclic databases.

(2) Initialization:

(i) perform selection on A to obtain the first set of results;

(ii) perform selection on B using the query constant to obtain the first wavefront WAVE;

(iii) perform join of WAVE with A and then C to obtain another set of results; and

(iv) set the iteration number IterNum to 1.

(3) Iteration:

(i) increment IterNum by 1;

(ii) perform join of B with WAVE to derive a new WAVE; and

(iii) use the new WAVE to perform join with A, and then consecutively perform joins with C for IterNum times to derive another set of results.

(4) Termination:

Terminate when WAVE is empty.

If B is cyclic and WAVE is not empty, terminate when (i) WAVE generates no new tuples (only cyclic tuples), and (ii) all the cyclic tuples ever generated by WAVE are marked *dead*. (See below for the explanation of *dead*). □

The determination of the termination condition for cyclic data can be implemented as follows.

(1) Modify the implementation of the join operation for tracing the tuples generated by each tuple.

(2) Observe the cyclic driver tuples † to see if they can produce any new results. This is done as follows. At some iteration when WAVE derives a tuple t, enter it into BUFF, and examine the result generated from tuple t. If any new tuple is generated, add it to BUFF; if t generates no tuple or all the tuples generated are already in BUFF, mark t as **dead**. The dead tuple will not be used to generate new tuples in future iterations. The process terminates when (i) WAVE is empty or (ii) it generates no new tuples and all the driver tuples ever generated by WAVE are marked dead.

† A cyclic tuple can be detected by checking whether a newly derived tuple is already in BUFF.

The processing done by SW is based on the soundness and completeness of deductive compilation. All we need to verify are the necessary and sufficient conditions for termination.

Theorem 5.1. The termination condition in Algorithm SW is a sufficient condition for both acyclic and cyclic databases.

Proof Sketch

Each new solution is derived by using WAVE to perform more operations. When WAVE is empty, there is no hope of generating any new tuples or new WAVE's. Thus WAVE being empty is sufficient for the process to terminate.

For cyclic databases, when WAVE is not empty, the process terminates when (i) WAVE generates no new tuples, and (ii) all the cyclic tuples ever generated by WAVE are marked dead. The reasoning behind this follows.

When WAVE still contains some new tuples, we cannot conclude termination because a new tuple may derive some new results driving the second half of the processing.

We now argue that our tuple marking method is correct for marking **dead** tuples. A tuple is marked dead if it cannot derive any new tuples. When a tuple t cannot derive any new tuples in BUFF at the k-th iteration, it means there will be no new tuples derived later for tuple t, because tuples from the k-th iteration are the base for deriving tuples at the k+1 th iteration. If each tuple has appeared before, it means that it has already been used at some previous iteration. Using it again can only generate what is already in BUFF. So the tuple t should be marked dead.

When WAVE generates no new tuples (it contains cyclic tuples only), all the tuples generated by WAVE later will be from among the tuples it has previously generated. But if all the cyclic tuples ever generated by WAVE are marked dead, it means that WAVE will henceforth generate dead tuples only. So the process should terminate at this point. □

Interestingly, the termination condition in Algorithm SW is, though sufficient, not a necessary condition. There is some chance that the result of the join sequence $A \bowtie C \bowtie \cdots \bowtie C$ becomes empty at some iteration before WAVE reaches empty or before it reaches the termination state for cyclic data. In this case, further iteration with WAVE will be fruitless. This could happen, for example, when the C relation has a data distribution similar to *ancestor* where the database registers only several generations but no more.

The reasons we do not include within our algorithm making the termination condition both sufficient and necessary are: (i) it is quite expensive to compute the iterative joins of the entire relations $A \bowtie C \bowtie \cdots \bowtie C$; and (ii) we expect that for practical cases it will be rare that the later halves of our compiled formulas, which contain joins of whole relations, will reach empty before the earlier halves, which are started with a selection. Testing only the sufficient condition would thus appear to cost less than testing both conditions.

When there is no case for some given relations where the later half becomes empty *before* the termination condition is met, the termination condition in Algorithm SW is a necessary condition for cyclic databases. This is demonstrated in the following example.

Example 5.1. The termination condition in Algorithm SW is a necessary condition for the following database.

Suppose the recursive rule set is the same as (5-1) and (5-2), and the data relations A, B and C contain the following tuples. Note that some are cyclic tuples.

B : { (1, 2), (1, 6), (2, 1), (2, 7) }
A : { (1, 2), (2, 5), (3, 4) }
C : { (1, 2), (2, 3), (3, 1), (4, 5), (5, 4) }

The cycle length for B is 2 and that for C is 3. The tuples in the relation form directed graphs as in Figure 5.3.

For a query *?- R(1, z).*, the solutions are z = { *1, 2, 3, 4* }. The execution sequence is as in Table 5.1, where * means that the tuple in σB^k (WAVE) is marked **dead**. Even when WAVE contains only dead or cyclic tuples (for example, at the third iteration, WAVE contains just 2 and 6), we still do not terminate because the cyclic tuple 1 which was generated in WAVE has not died yet. That is why the algorithm requires that the process terminates when WAVE generates no new tuples *and* all the cyclic tuples ever generated by WAVE are marked dead. □

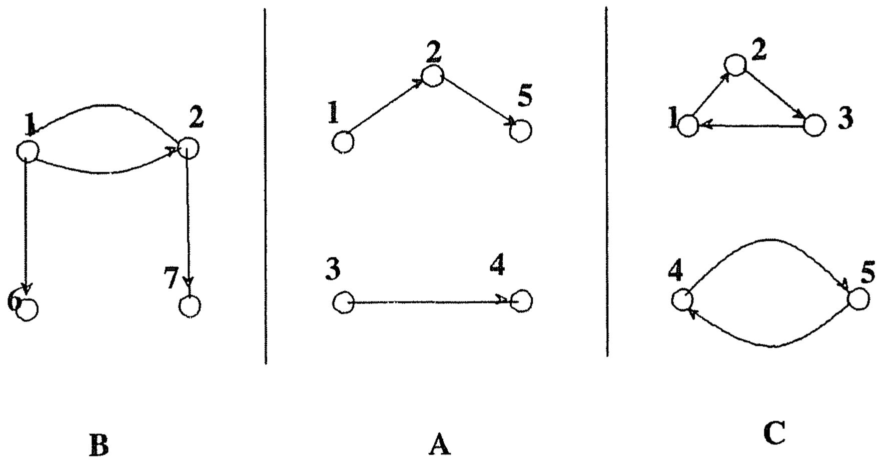


Figure 5.3 A Directed Graph for Cyclic Tuples

k	σB^k	·A	C^k	Buff { driver : [derived]] }	Solution
0	/	2	/	{}	[2]
1	2, 6*	5	4	{2:[4]}	[2, 4]
2	1, 7*	2	1	{2:[4]} {1:[1]}	[1, 2, 4]
3	2*, 6*	5	4	{2:[4]} {1:[1]}	[1, 2, 4]
4	1, 7*	2	3	{2:[4]} {1:[1, 3]}	[1, 2, 3, 4]
5	2*, 6*	5	4	{2:[4]} {1:[1, 3]}	[1, 2, 3, 4]
6	1*, 7*	2	1	{2:[4]} {1:[1, 3]}	[1, 2, 3, 4]

Table 5.1. Recursion and Termination for Cyclic Data

Algorithm SW has two important features: (i) it performs selection first to reduce the size of relations to be joined later, and (ii) it takes advantage of previous processing by using a wavefront relation. These two heuristics are the main features of Henschen and Naqvi's algorithm [Hens 84]. Essentially, SW is the same as Henschen and Naqvi's algorithm, except that we apply relational operations instead of their enqueue and dequeue operations on the queue of tuples. Based on the performance results presented later, SW performs the best in most cases among the four algorithms we consider.

5.2.2. The Central Wavefront Algorithm

If we want to reduce the number of joins performed in the processing, we may let the processing start from the center to join relations B and C at each iteration, and a **central wavefront** is saved for the next iteration. Then each set of solutions is obtained by performing *selection* on the new wavefront. The result is the union of all such solution sets.

The processing flow is illustrated in Figure 5.4 and the formula is simply

$$\sigma(B \bowtie B^{k-1}AC^{k-1} \bowtie C) \tag{5-17}$$

At each iteration, the last *central wavefront* $B^{k-1}AC^{k-1}$ is used to join relation B which results in an intermediate relation, then we perform a join with relation C to get new *central wavefront* $B^{k}AC^{k}$. The set of answers for this iteration is then obtained by selection on this new wavefront. The total number of joins for all iterations is

$$\sum_{n=0}^{k} 2 = 2k.$$

As with the algorithms NE and SW, it is beneficial to build index files on corresponding join columns or to build a join index file for relations B and C.

Algorithm 5.2 (CW) : The Central Wavefront Algorithm

(1) Data structure : the wavefront relation WAVE.

(2) Initialization : take relation A as the initial WAVE.

n = 0	σ A
n = 1	σ (B * A * C)
n = 2	σ (B * BAC * C)
n = 3	σ (B * B^2AC^2 * C)
...	. . . . . .
n = k	σ (B * $B^{k-1}AC^{k-1}$ * C)

Figure 5.4. The Processing Flow of Algorithm CW (Central Wavefront)

(3) Iteration :

(i) perform selection on WAVE to derive one set of results.

(ii) join relation B with WAVE, and then join the result with with C to derive the new WAVE.

(4) Termination : The process terminates when WAVE goes to empty or does not change. □

The idea of Algorithm CW is to work out the full joins for the entire relations A, B, and C at each iteration and then perform selection based on the specific query constant. The evaluation plan is derived from the general compiled formula which is sound and complete. Now we verify that the termination condition is both necessary and sufficient.

Theorem 5.2. The termination condition in Algorithm CW is both necessary and sufficient.

Proof Sketch

The termination condition is sufficient for both cyclic and acyclic databases because, when WAVE is empty or does not change from joining B and C, there will be no more change from joining them yet one more time. The selection will be on the same WAVE relation for all later iterations, hence no new result will be obtained.

The termination condition is necessary for both cyclic and acyclic databases because, if WAVE is not empty and it changes for joining B and C, there is the possibility that the selection on the given query constant will obtain some new result. It cannot terminate at this point. So the termination condition is necessary. □

Algorithm CW is essentially the evaluation strategy proposed by Shapiro and McKay [Shap 80][McKa 81]. It has a quite small number of joins and a quite simple termination condition, but at each iteration it involves two joins on three unrestricted relations. Thus it tends to generate very large intermediate relations and could easily run into combinatorial explosion. The performance evaluation results presented later show that in most cases it is less efficient than Algorithm SW.

5.2.3. The Double Wavefront Algorithm

Algorithm CW does not use the query constant in its earlier processing and thus results in joining unrestricted relations. A possible modification is to make earlier use of the constant while keeping the number of joins small.

This leads to Algorithm DW which has **double wavefronts** (Figure 5.5). We form one *wavefront* by starting from relation B, performing selection first, and then performing successive joins with relation B to achieve σB^k from σB^{k-1}; and we form another *wavefront* by starting from relation A, performing

$$\begin{array}{ll}
n = 0 & \sigma \, A \\
n = 1 & (\sigma \, B) * (A * C) \\
n = 2 & (\sigma B * B) * (AC * C) \\
n = 3 & (\sigma B^2 * B) * (AC^2 * C) \\
\ldots & \ldots\ldots \\
n = k & (\sigma \, B^{k-1} * B) * (AC^{k-1} * C)
\end{array}$$

Figure 5.5. The Processing Flow of Algorithm DW (Double Wavefronts)

successive joins with relation C to achieve AC^k from AC^{k-1}. This algorithm is illustrated with the formula,

$$(\sigma B^{k-1} * B) * (AC^{k-1} * C) \tag{5-18}$$

At the k-th iteration, using the two saved wavefronts σB^{k-1} and AC^{k-1}, we perform one join on relation B (to derive the new wavefront σB^k), a second join on relation C (to derive the other new wavefront AC^k), and a third join on the two wavefronts σB^k and AC^k to get the k-th result $\sigma B^k AC^k$. The total number of joins for all iterations is

$$\sum_{n=0}^{k} 3 = 3k.$$

It is again beneficial to build index files on joined columns for relation B and C, or join index files. There will be no index files available for the two changing wavefronts σB^k and AC^k. Different join methods, such as hashing may be used to facilitate the join process.

Algorithm DW performs selection during early processing and reduces the number of joins to 3k, but the second wavefront AC^k involves joining two large unrestricted relations, which renders the algorithm less attractive.

Algorithm 5.3 (DW) : The Double Wavefront Algorithm

(1) Data structure :

(i) two wavefronts WAVE1 and WAVE2; and

(ii) a buffer relation BUFF used in the determination of termination for cyclic databases.

(2) Initialization :

(i) perform selection on relation A to get the first set of results;

(ii) perform selection on relation B to obtain the initial WAVE1; and

(iii) join relation A with C to obtain the initial WAVE2.

(3) Iteration :

(i) join WAVE1 with WAVE2 to derive a set of results.

(ii) join B with WAVE1 to derive the new WAVE1; and

(iii) join WAVE2 with C to derive the new WAVE2.

(4) Termination:

When either WAVE1 or WAVE2 becomes empty, or when both wavefronts do not change, terminate. □

A further termination judgement for cyclic B may be obtained using the same method as for algorithm SW, which is not presented in detail here.

Algorithm DW follows the ideas underlying algorithms SW and CW. It takes advantage of early selection to reduce the size of relations to be joined for WAVE1, and it needs only three joins at each iteration. But WAVE2 involves joining two large unrestricted relations. The join of two temporary relations, WAVE1 and WAVE2, may cause some extra burdens. Based on the performance analysis in the next section, Algorithm DW performs less well than Algorithm SW in most cases.

5.3. PERFORMANCE COMPARISON OF THE SLSR ALGORITHMS

Obviously, processing cost can not be simply related to the number of joins, as the size of the joined relations contributes a significant factor to total cost. In order to compare the algorithms provided in this chapter, both analytical modelling and experimental tests were used to determine I/O cost, CPU cost, and maximum storage space needed.

Our performance experiment is based on research on query processing and query optimization for conventional relational database systems [Jark 84b]. In recursive processing, because the same join attributes (columns) of the joined relations B, A, and C are used repeatedly, indices on join attributes should be built to facilitate the repeated accessing of the same columns. The results of Blasgen and Eswaran [Blas 77] indicate that two join methods, the indexed nested-loop join and the sort-merge join, perform well for both small and large relations. In our experiment, the indexed nested-loop join method using B-tree indices was used for all four algorithms, and non-clustered index files on the corresponding join attributes of B, C, and A were built to facilitate iterative joins. In fact, some other access structures and join methods could also be applied, such as a hashed indicies or a sort merge join algorithm, to improve the performance of the algorithms. Our experiment emphasizes the comparison of the algorithms under similar structures and join methods. Further improvement of accessing mechanisms should be explored in future research.

5.3.1. Analytical Model

We first built an analytical model to calculate the costs of the four algorithms. The calculation of cost is based on the following parameters: (1) join selectivities on relations B and B: J_{bb}, on relations B and A: J_{ba}, on relations A and C: J_{ac}, and on relations C and C: J_{cc}; (2) the selectivity of selections on relations A and B; and (3) the sizes of relations B, C, and A.

The analytical formulas are extracted from the analysis of the query access paths in relational databases [Seli 79] and from the experience obtained in experimental performance testing in relational database systems [Lu 85].

For the indexed nested-loop join, the processing is performed with a sequential scan of the outer relation and an indexed scan of the inner relation, i.e., the outer relation is scanned once and, for each outer relation tuple, the index on the inner relation is used to probe for matching tuples. If a matching tuple is found, the data page is fetched for accessing. Hence we have the formula for I/O cost

$$IO\_Cost_{join} = (NumPage(Outer) + NumOfTuples(Outer) * (w_1 * depth(InnerIndex) + w_2 * J_j * NumOfTuples(Inner))) * Page\_IO\_cost \quad (5\text{-}19)$$

where J_j is the join selectivity of two relations Inner and Outer, depth is a function representing the estimated depth of the B-tree, w_1 is the probability of the inner relation index pages not being in the buffer and w_2 is the probability of data page fetching.

The I/O cost for the selection operation is

$$IO\_Cost_{selection} = s * NumOfTuples(Rel) * (w_1 * depth(Index\_of\_Rel) + w_2) * Page\_IO\_cost \quad (5\text{-}20)$$

where s is the selectivity of the relation, w_1 is the probability of the index page not being in the buffer and w_2 is the probability of data page fetching.

The CPU cost can be calculated with

$$CPU\_Cost_{join} = CPU\_Cost_{outer} + CPU\_Cost_{inner} + CPU\_Cost_{merge} \quad (5\text{-}21)$$

where CPU_Cost_{outer} is the CPU time needed to fetch the tuples of the outer relation. It can be expressed as

$$CPU\_Cost_{outer} = Tuple\_CPUCost_{outer} * NumOfTuples(Outer) \quad (5\text{-}22)$$

CPU_Cost_{inner} is the CPU time needed to fetch the tuples of the inner relation given outer relation tuples

$$CPU\_Cost_{inner} = Tuple\_CPUCost_{inner} * NumOfTuples(Outer) * depth(Inner) * FanOut \quad (5\text{-}23)$$

where *FanOut* is the average fan-out of a B-tree node that should be searched in finding a matching tuple in a B-tree index node. Finally, CPU_Cost_{merge} represents the cost for merging the result tuples from two relations

$$\begin{aligned} CPU\_Cost_{merge} = {} & Tuple\_CPUCost_{merge} * J_j * \\ & NumOfTuples(Outer) * NumOfTuples(Inner) \end{aligned} \tag{5-24}$$

Also, we have

$$\begin{aligned} CPU\_Cost_{selection} = {} & s * NumOfTuples(Rel) * \\ & Tuple\_CPUCost_{selection} * depth(Rel) * FanOut \end{aligned} \tag{5-25}$$

The total processing cost is calculated by summing up the I/O and CPU costs

$$Total\_Cost = IO\_Cost + CPU\_Cost \tag{5-26}$$

The maximum space needed for storing intermediate relations during the iterative processing is also presented in the test. It is the maximum number of pages used in the processing. It is calculated by counting the number of tuples newly generated in the iterative joins and then calculating the maximum storage needed to hold these new tuples. These formulas are used in our analytical testing, with the parameters related to specific databases adopted to be the same as in the experimental testing. Other parameters are estimated based on the experience obtained in [Lu 85] and adjusted by the experimental tests.

5.3.2. Experimental Method

The performances of the four algorithms were also evaluated by experimental tests. This is important since (1) the formulas presented in the last section enable only a rough estimation of processing costs, and they need to be validated; and (2) the formulas represent estimation of processing costs given *general database characteristics*, and they do not reflect the data dependent aspects of different databases and implementation details.

The tests were conducted on a synthetic database, Wisconsin Benchmarking Database [Bitt 83], built on WISS (WIsconsin Storage System) [Chou 83] running on a VAX11/750 machine (Wisc-devo) in a single user environment. There are three data relations B, A, and C in the database. Each relation contained 1000 tuples, and each tuple was 182 bytes long. The join columns of the three relations were generated using a random number generator. We use the built-in random number generator function *rand* of UNIX. The ranges of the values of join columns are restricted to control the join selectivity between relations. We assume that the outer relation was unordered, so we built non-clustered B-tree indexes on the join columns.

5.3.3. Performance Testing

The results of analytical modeling and experimental testing are presented in Figures 5.6 to 5.10. With both approaches, we compare the four algorithms NE, SW, CW and DW presented in this chapter. The results of the analytical and experimental evaluation are quite similar. Because Algorithm SW is a direct improvement of Algorithm NE by saving the previous processing results, which obviously saves processing power, the performance of NE is only presented for the analytical model and is not explicitly shown for the experimental tests. Both approaches are applied to computing the processing time (in seconds) versus selectivity on relation B, and versus join selectivities J_{bb}, J_{ba} and J_{ac} or J_{cc} [†], under certain different situations.

[†] J_{bb} represents the join selectivity of joining relation B with B. Other notations for join selectivities are similar.

With the analytical model we also show the result when both join selectivities of J_{bb} and J_{cc} become larger (Figure 5.9). We also tested the maximum disk storage needed vs. selectivity in our analytical tests (Figure 5.10), because the recursive processing of databases consumes large amounts of intermediate storage space. (Sometimes we ran out of the disk space during the testing!).

For simplicity, we have $J_{bb} = J_{ba}$ and $J_{ac} = J_{cc}$ in both experimental and analytical tests. To distinguish the two different situations in iterative joins, we call join selectivity *low (high)* when the joined result relation is smaller (larger) than the original one. The iteration number is set to 5 for both experimental and analytical tests.

The analytical and experimental results are presented as follows.

(1) Figure 5.6: Processing time vs. selectivity with join selectivities J_{bb} and J_{cc} fixed.

(2) Figure 5.7: Processing time vs. join selectivity J_{bb}, where (a) (b) are the tests using high selectivity s ((a) is the analytical test, and (b) is the experiemental test), and (c) (d) are the tests using low selectivity s ((c) is the analytical test, and (d) is the experiemental test).

(3) Figure 5.8: Processing time vs. join selectivity J_{cc}, where (a) (b) are the tests using high selectivity s ((a) is the analytical test, and (b) is the experiemental test), and (c) (d) are the tests using low selectivity s ((c) is the analytical test, and (d) is the experiemental test).

(4) Figure 5.9: The analytical test on processing time vs. join selectivity J_{bb} with medium selectivity s and high join selectivity J_{cc}; and

(5) Figure 5.10: The analytical test on storage required vs. selectivity s with medium join selectivities J_{bb} and J_{cc}.

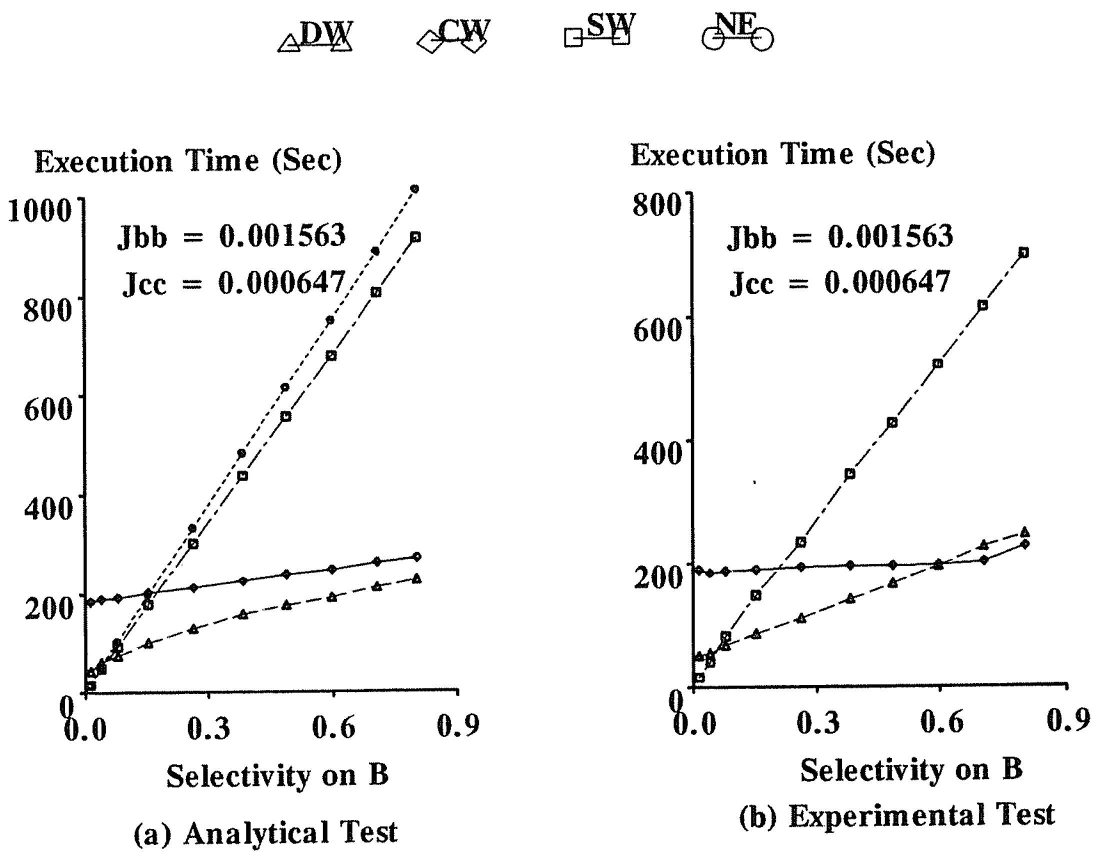


Figure 5.6

Processing Cost vs. Selectivity on Relation B

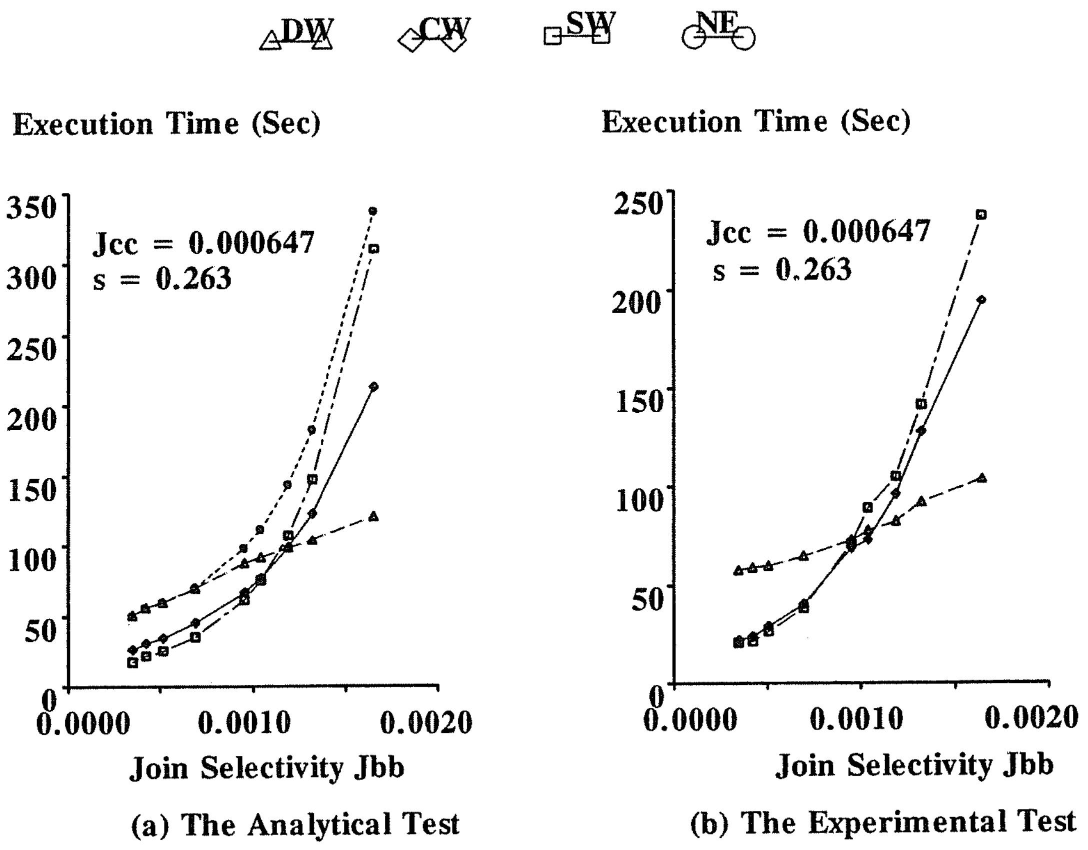
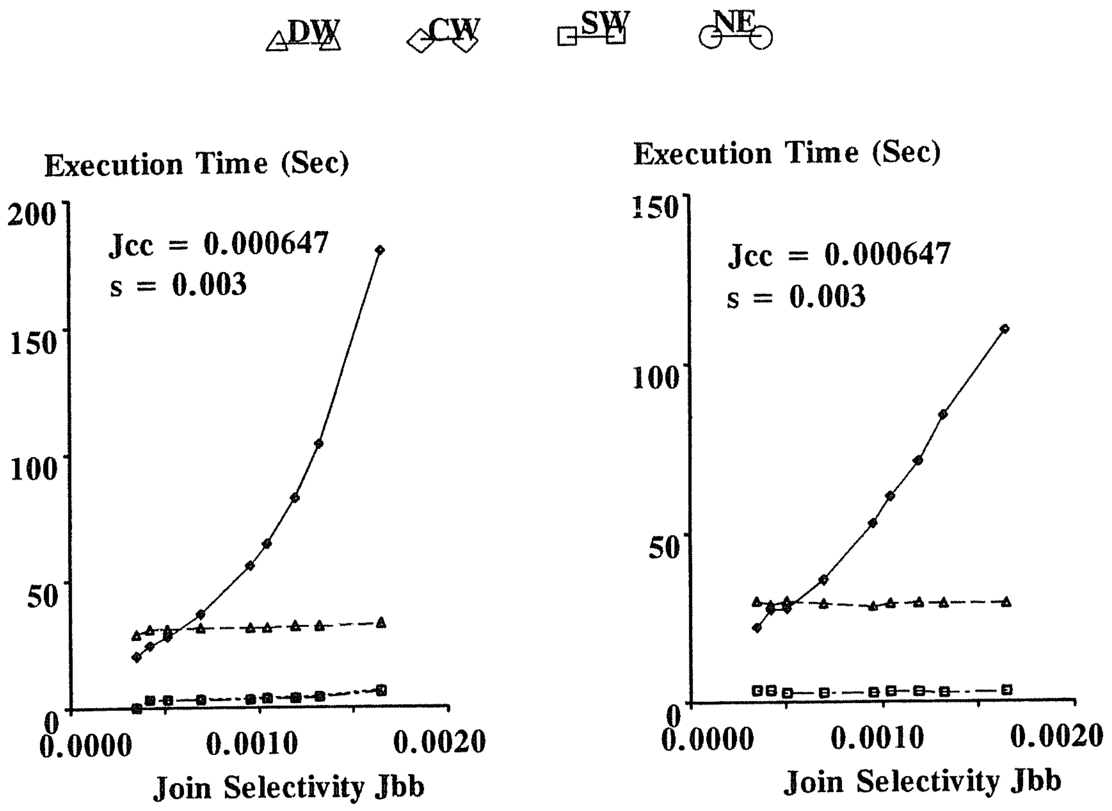


Figure 5.7 (a) (b)

Processing Cost vs. Join Selectivity Jbb

(With High selectivity)



(c) The Analytical Test (d) The Experimental Test

Figure 5.7 (c) (d)

Processing Cost vs. Join selectivity Jbb

(With Low Selectivity)

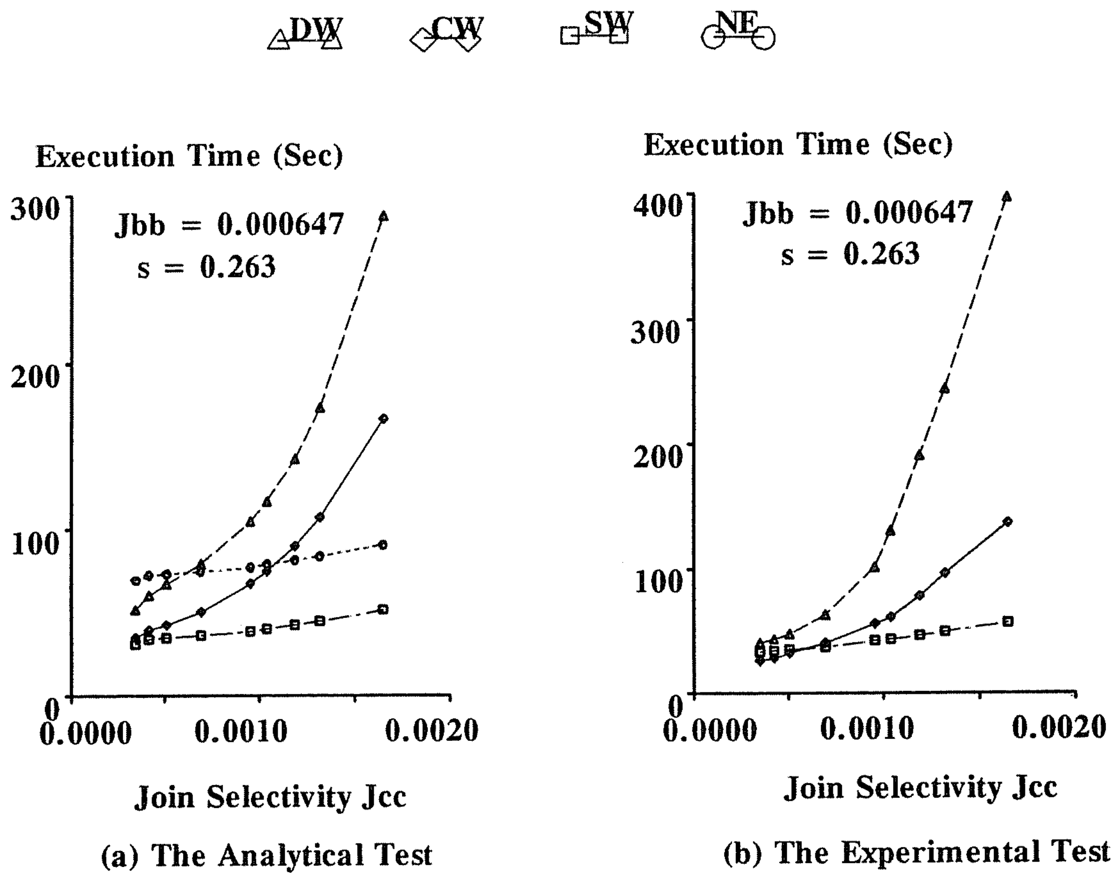


Figure 5.8. (a) (b)

Processing Cost vs. Join Selectivity Jcc

(With High Selectivity)

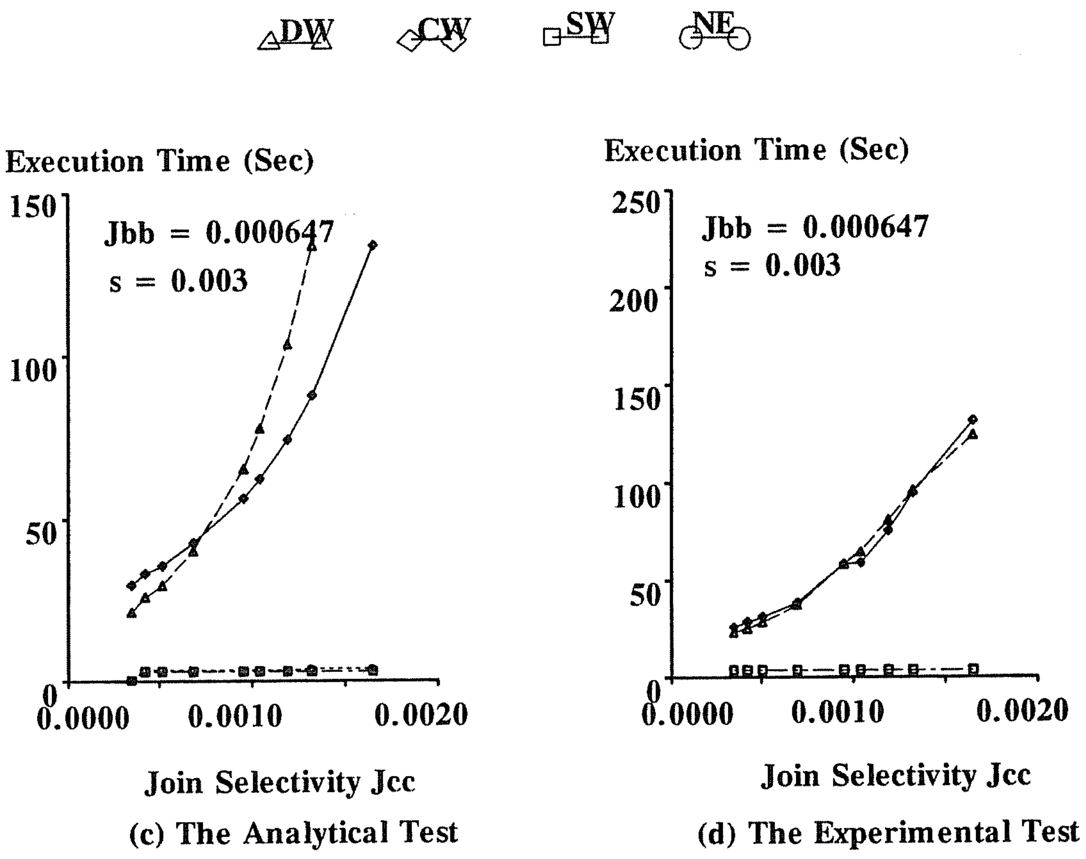
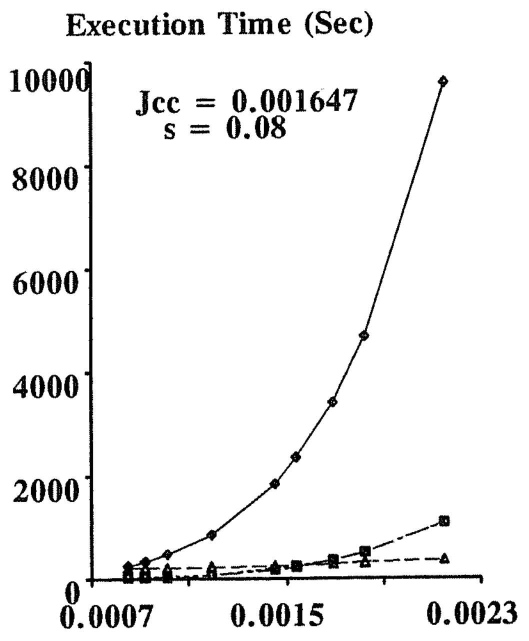


Figure 5.8. (c) (d)

Processing Cost vs. Join Selectivity Jcc

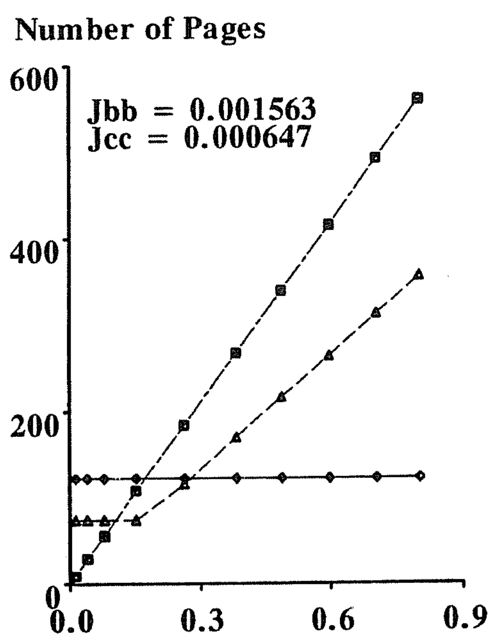
(With Low Selectivity)

△ DW ◇ CW □ SW ○ NE



Join Selectivity Jbb

Figure 5.9
Execution Cost
vs.
Join Selectivity



Selectivity on B

Figure 5.10
Storage Expanded
vs.
Selectivity

5.3.4. Summary of Testing Results

The experimental tests and analysis present some interesting results. A first observation is that processing recursive queries using compiled formulas and relational operations is practical for relational databases, provided (i) the join selectivities of the data relations are not large, (ii) the selectivity s is low, and (iii) the appropriate algorithm is adopted. A second observation is that different algorithms have diverse performances under different situations. Choice of the best algorithm is definitely database and query dependent. Nevertheless, there are some important heuristics for the selection of an appropriate algorithm.

(1) **Perform selection first.**

When a query provides highly selective information (s is low), Algorithm SW, *performing selection first* and avoiding joining the whole (non-restricted) data relations, has much better performance than the other two algorithms. Since most practical queries provide such information, e.g. *retrieving John's ancestors* rather than *retrieving everybody's ancestors*, Algorithm SW should be selected for most recursive query executions.

In those cases that the query does not provide highly selective information (i.e. when s is quite high, our test would indicate in the range $s > 0.1$), *performing selection first* may not win, because the number of joins will be an important factor for total processing cost. In these cases, Algorithm CW and DW may perform better.

(2) **Save previous processing results to avoid redundant processing.**

The utilization of previous intermediate processing results (*wavefront algorithms*) avoids some redundant processing, thus reducing total processing cost. The reduction is demonstrated in our comparison of Algorithms NE and SW. The comparison shows that the wavefront idea should be incorporated in all situations.

(3) **Group processing of joins with diverse join selectivities, which may generate smaller intermediate relations.**

Diverse join selectivities means that in a sequence of joins the join selectivities of some joins are quite high while the join selectivities of others are quite low. The joins with diverse join selectivities may be grouped for processing together in order to generate smaller-sized intermediate relations to offset the huge intermediate relation sizes which would be generated from iteratively joining relations with higher join selectivity.

For example, when the join between relation B with itself has high join selectivity while the join of relation with C has low join selectivity, Algorithm CW and DW will generate smaller intermediate relations than iteratively joining B's only (Figure 5.10). This results in lowered processing cost (Figures 5.7 (a) and (b)).

(4) **Reduce the size of relations to be iteratively joined.**

From the figures we can see that the join selectivities are essential in contributing to processing cost. High join selectivity tends to run into combinatorial explosion, while low join selectivity generally results in quite efficient processing. To reduce the high cost of recursive processing, we must try to reduce the size of relations to be iteratively joined.

One approach is to apply better indexing techniques to reduce the size of tuples to be iteratively joined, for example, performing joins on index files only instead of on entire data tuples, or building join index files and performing joins on join indices only [Vald 85], where a join index is a binary relation that consists of the joined attributes of two relations. Because the length of an index is shorter than that of a tuple, the size of the data to be iteratively joined will be reduced considerably.

CHAPTER 6

STACK-DIRECTED COMPILATION OF COMPLEX RECURSIVE RULES

This chapter introduces a stack-directed compilation method for processing recursive rules more complex than those discussed in the previous two chapters. A recursive rule set which requires stack-directed compilation is one whose resolution graph contains more than one cycle. Such rule sets are in Class III, IV or V of Chapter 3. In this chapter, we develop a stack-directed compilation algorithm for SLMR (Single Looping rule with Multiple Recursion points) rule sets (Class III), and we study the optimization of accessing the database for such rule sets using wavefront and potential wavefront relations. This approach can be applied to other forms of recursion with appropriate modification of the basic algorithm.

In the compilation of transitive closures and SLSR rules, complete general compiled formulas can be derived before any accessing of the database needs occur. However, there are no such general compiled formulas that can be derived for recursion with more than one resolution cycle. The stack-directed processing should be applied for such cases. The stack-directed processing is not **pure** compilation, because the stack operation is determined by results returned from accessing data relations. But it has more flavor of compilation than of interpretation, because it delays evaluation of base predicates until the deductive resolution for a formula has been finished, and the execution of compiled formulas can be subjected to relational set-oriented operations.

Since no general compiled formula can be derived for recursion involving more than one resolution cycle, a blind test-and-try method is generally incomplete or redundant. Our approach is to generate a processing plan which is complete (i.e., no possible solution will be missed), which is non-redundant (i.e., the same formula will not be derived more than once), which is easy to optimize, and for which termination conditions can easily be determined.

6.1. A STACK-DIRECTED COMPILATION ALGORITHM

6.1.1. A Case Study

We first perform a case study for an SLMR rule set which consists of a single looping rule with double recursion points and a single exit rule. We assume that data relations contain no cyclic data.

Example 6.1. Derive a processing plan for query *?- R(a, z).* where *R* is defined with the SLMR rule set

$$R(x,z){:-}A(x,z). \qquad (6\text{-}1)$$

and

$$R(x,z){:-}B(x,y),R(y,w),C(w,u),R(u,v),D(v,z). \qquad (6\text{-}2)$$

In order to achieve ordered expansion, a **compilation stack** is used to hold intermediate formula expansions for generating compiled formulas. Each stack element is a clause consisting of a list of query literals or their expansions generated by calling non-exit rules. Thus a stack element consists of a list of literals which may be extensional or intensional. An intensional literal within a stack element is called a **potential recursion point (PRP)**.

We identify three states for each PRP in a compilation stack: **active, potentially active**, and **exhausted**. **Active** is the state where the PRP is to be expanded immediately by non-exit calls; **potentially active** is the state where the PRP is not to be expanded immediately but will become active later; and **exhausted** is the state where the PRP has already been exhaustively expanded and can not be further expanded by non-exit calls.

The following analysis of the example illustrates the stack-directed compilation process.

(1) Step 1: The query literal R is pushed onto the compilation stack. The first PRP of the top stack element, R, is set to *active*. The first compiled formula is generated by substituting for R using the exit rule. We have

Stack Top: **R**
Compiled Formula Generated: σ A.

Our conventions are that when an intensional literal on the stack is shown in boldface, it is **active**, when it is italicized, it is **potentially active**, and when it is in Roman, it is **exhausted**.

(2) Step 2: Because there is on top of the stack an active PRP, which could be replaced by its non-exit definition, we set the action to PUSH. PUSH is performed by the following steps: (i) A new stack element is formed by substituting for the active PRP its non-exit definition, with the states of newly generated PRP's initialized to *potentially active*, the states of the rest of the literals on top of the stack remaining unchanged, and the first *potentially active* PRP on top of the stack set to *active*; (ii) the new stack element is pushed onto the stack; and (iii) a set of compiled formulas is generated by substituting for all PRP's of the new stack element their exit definitions.

In our case, because R is active, PUSH is performed as follows: (i) R is replaced by its non-exit definition, forming a new stack element: B, **R**, C, *R*, D.; (ii) the new element is pushed onto the top of the stack; and (iii) the compiled formula: σ B, A, C, A, D. is generated by substituting for all R's in the new element according to the exit rule R :− A. We now have

Stack Top: B, R, C, *R*, D
Compiled Formula Generated: σ B, A, C, A, D.

(3) Step 3: Suppose the expression in front of the active PRP is not exhausted, i.e., σB is not empty [†]. The stack is pushed again. Then we have

Stack Top: B, B, R, C, *R*, D, C, *R*, D
Compiled Formula Generated: σ B, B, A, C, A, D, C, A, D.

(4) Step 4: If PUSH is the action for one more step, the stack becomes

Stack Top: B, B, B, R, C, *R*, D, C, *R*, D, C, *R*, D
Compiled Formula Generated:
σ B, B, B, A, C, A, D, C, A, D, C, A, D.

Suppose the expression in front of the active PRP is exhausted due to empty σBBB. The generated compiled formula produces no answer so it is useless to expand on this PRP further. The stack is then *popped.* POP is performed by the following steps: (i) decrement the stack pointer by 1; (ii) change the state of the active PRP to *exhausted*; and (iii) if there is a next non-exhausted PRP (the first *potentially active*) PRP in the top stack element, set it to *active*, otherwise pop the stack again. The popping may continue until the stack is empty.

In our case, when stack element No. 4 is popped, the active state is shifted to the non-exhausted PRP of stack element No. 3.

(5) Step 5 : After the active state is shifted, PUSH is called again and the situation becomes

Stack Top: B, B, R, C, B, R, C, *R*, D, D, C, *R*, D.
Compiled Formula Generated:

[†] Here we study the acyclic database and simply apply the termination condition discussed in Algorithm SW.

$$\sigma \text{ B, B, A, C, B, A, C, A, D, D, C, A, D.}$$

The snapshot of the stack at this point in the processing is presented in Table 6.1.

The compilation process is directed by stack expansion, which mirrors the execution of a recursive function. The difference between the two (stack-directed compilation and the execution of a recursive function) is that the former is controlling the generation of database access plans before accessing a database, while the latter uses a stack to create local execution environments and to pass execution results.

The stack-controlled generation of compiled formulas facilitates the grouping of similar processing patterns and the use of previous processing results. For example, to process the formula generated at Step 4

$$\sigma B,B,B,A,C,A,D,C,A,D,C,A,D.$$

we may use the Step 3 processing result, σB^2, in deriving σB^3, and we may group operations together, e.g., $(C,A,D)^3$.

No.	Stack Element
4	B, B, R, C, B, R, C, *R*, D, D, C, *R*, D.
3	B, B, R, C, R, D, C, *R*, D.
2	B, R, C, *R*, D.
1	R.

Table 6.1. Snapshot of the Compilation Stack (at Step 5)

This is reminiscent of our wavefront algorithms. The saving of σB^2 for deriving σB^3 is essentially the Algorithm SW. The grouping of (C,A,D) and using previous results $(C,A,D)^2$ for deriving $(C,A,D)^3$ is essentially the *second wavefront* of Algorithm DW.

Moreover, the saving of previous results is not confined to the *active* PRP. A saving connected with a *potentially active* PRP may benefit further processing after popping. For example, the saving of σBBAC at step 3 may benefit the processing for the compiled formula generated at step 5: σBBAC, BACADDCAD. We call this kind of saving *potential wavefront.*

Based on the performance analysis of the previous chapter, we predict that the grouping and saving of $(C,A,D)^2$ for deriving $(C,A,D)^3$ may not gain much because it involves the join of unrestricted relations. The benefit of performing selection first and then using wavefronts is obvious. That is why we explore two kinds of wavefront relations in stack-directed processing: *WAVE,* which is the saving of processing results up to but excluding the current active PRP, and *Potential WAVE's (P_WAVE's)* , which are the savings of the processing results up to but excluding potentially active PRP's. Once the stack is popped, the first Potential WAVE is used to derive the new WAVE. There is only one WAVE; the number of Potential WAVE's is equal to the number of potentially active PRP's on the stack. □

6.1.2. The Essential Stack-Directed Compilation Algorithm

Based on the discussion of Example 6.1, we present an stack-directed compilation algorithm, Algorithm ES. The algorithm works only for simple SLMR rule sets on acyclic databases. However, for more complex rule sets and cyclic databases, it can be extended by appropriate modifications. It does realize the *essen-*

tial ideas of our stack-directed compilation approach. Although we have worked through the details, there does not appear to be anything of major conceptual interest in extending the Algorithm ES to handle greater complexity so we do not present the details in this dissertation.

Similar to the algorithms in previous chapters, the processing start point is decided by query constants. The algorithm is presented by assuming that the start point is at the left. A different query may require starting at the right, but the logic would be symmetric to what is presented here.

Algorithm 6.1 (ES) : Stack-Directed Compilation for an SLMR Rule Set

(1) Data structures:

(i) a *compilation stack* where each stack element consists of a clause, i.e., a list of literals;

(ii) a *state marker* which indicates the state of each PRP: *active*, *potentially active*, or *exhausted*; and

(iii) an ACTION *flag* to indicate the action to be performed: PUSH, POP, or SHIFT.

(2) Initialization: Initially, the compilation stack is empty.

(i) the query literals are first pushed onto the stack, and the leftmost PRP is set to *active* and the remaining PRP's to *potentially active*;

(ii) a compiled formula is generated by substituting for each PRP on top of the stack its exit definition; and

(iii) if the active PRP on top of the stack is not *exhausted*[†] , set ACTION to PUSH, otherwise set it to POP.

[†] For acyclic databases, an active PRP becomes *exhausted* when WAVE (the result of evaluating the sub-expression in front of the PRP) becomes empty, because at this point further pushing of the stack cannot generate any new answer.

(3) Iteration:

If ACTION is PUSH, then

(i) increment the stack pointer by 1;

(ii) substitute for the active PRP of the current top stack element its non-exit rule definition, to form a new top stack element;

(iii) initialize the states of the new PRP's of the new top stack element *potentially active*, while the states of the PRP's that already existed in the parent stack element are as they were there (i.e., *potentially active* or *exhausted.*);

(iv) set the state of the first *potentially active* PRP on top of the stack to *active*;

(v) generate a compiled formula by substituting for the PRP's of the top stack element their exit definitions; and

(vi) if the active PRP on top of the stack is not *exhausted*, set ACTION to PUSH, otherwise set it to POP.

If ACTION is POP, then pop the top stack element, decrement the stack pointer by 1, and set ACTION to SHIFT.

If ACTION is SHIFT, then

(i) if there is no potentially active PRP in the top stack element to be activated, reset ACTION to POP;

(ii) otherwise, set the current active PRP to *exhausted*, set the first *potentially active* PRP to *active*, and set ACTION to PUSH.

(4) Termination: The process terminates when the stack becomes empty. □

Proof of the completeness and non-redundancy of the algorithm is in the next section.

6.2. DISCUSSION OF STACK-DIRECTED COMPILATION

6.2.1. Compilation vs. Interpretation

Should we consider the stack-directed processing **compilation** or **interpretation**? Obviously, it is not **pure** compilation, because the stack operation, such as PUSH or POP, is determined by the results that are returned from accessing data relations. This is unlike the compilation of transitive closures and SLSR rules, where complete general compiled formulas can be derived before any accessing of the database need occur. However, it has more flavor of compilation than of interpretation. The interpretation approach evaluates base predicates as soon as they are encountered, while the method being discussed here delays evaluation of base predicates until the deductive resolution for a formula has been finished, and the execution of compiled formulas can be subjected to relational query optimization routines. In this sense, it is still a compilation algorithm.

We may observe an interesting evolution of our deductive compilation concept. For non-recursive rule sets, query compilation is "pure" in the sense that, once a query is compiled, the database access plan is fully determined. For recursive rule sets of Classes I and II, compilation is less "pure" in the sense that the database access plan cannot be fully determined before any database access occurs because the number of iterations (the termination point) is data dependent. For rule sets which need stack-directed compilation, the compilation is even less "pure" in the sense that even the generation of compiled formulas is dependent on what is found in the extensional database. This is the reason that we need a **feedback information** line in our architecture design (Figure 1.1).

Theorem 6.1. The compiled formulas generated by Algorithm 6.1 are necessary, complete and non-redundant.

Proof Sketch

(1) Necessity

In the algorithm, a new stack element is obtained by "calling" the non-exit rule definition of the active PRP, when the top stack element is not "exhausted"; and a compiled formula is generated by "calling" the exit rule definitions for all PRP's in a new stack element, when the new element is pushed onto the stack. Both calls are based on the deductive resolution process. According to the compilation method discussed in previous chapters, each generated formula represents a set of solutions, which is a subset of all possible solutions. Thus the generation and processing of these formulas are necessary.

(2) Completeness

Suppose a stack element contains two recursion points as follows

$$\ldots,R_i,\ldots,R_j,\ldots \qquad (6\text{-}3)$$

According to the algorithm of deduction rule resolution, there are three possible non-exit calls on rule definitions: calling on R_i with R_j fixed, calling on R_j with R_i fixed, and calling on R_i and R_j at the same time. The first call is realized in the algorithm when R_i is active and R_j potentially active. The second call is realized in the algorithm when R_i is exhausted and R_j active. The third call is realized in the algorithm when R_i was first active and called with its definition pushed on to the stack to form a new stack element

6.2.2. Depth-First Search vs. Breadth-First Search

Interestingly, the algorithm reflects a combination of two search paradigms: **depth-first search** and **breadth-first search**.

If we observe only stack manipulation, we may conclude that we have adopted a **depth-first search and backtracking** mechanism to find all possible expansions. Each expansion is focused on one possible active point at a time. When this active point is exhausted, we backtrack (pop the stack) and resume expansion of the deepest unexhausted point. This imitates the depth-first search mechanism of many AI algorithms and of Prolog.

However, if we observe the execution of each compiled formula, we may conclude that we have adopted a breadth-first search mechanism for finding all possible solutions using database set-oriented operations. The formula is evaluated against the whole database instead of a tuple at a time.

The combination of the two search paradigms not only takes advantage of the execution efficiency of relational set-oriented operations and makes the solution complete, but it also benefits the inter-query optimization, i.e., optimization across the successively produced compiled formulas. Recurring sub-expressions in the compiled formulas can be extracted and evaluated only once, rather than at each occurrence.

6.2.3. The Algorithm is Complete and Non-Redundant

Stack-directed compilation produces a sequence of compiled formulas, and their processing clearly demands a considerable amount of processing power. It is important to make sure that the compiled formulas produced are necessary, complete and easily optimized. Necessity and completeness are analyzed in this section, and optimization is discussed in the next.

$$\ldots, R_{i_l}, \ldots R_{i_m}, \ldots, R_j, \ldots \tag{6-4}$$

and R_j becomes active in this new element. This kind of reasoning can be extended to any number of recursion points, and so every possible recursive call will be realized at some stage. The solution is complete.

(3) Non-Redundancy

Considering the same formula (6-3), there are only three possible non-exit calls expanding the two recursion points of the compilation formula on the stack. For these three calls, the first can be realized only when R_i is active and R_j potentially active, the second only when R_i is exhausted and R_j active, and the third only when R_i was first active and called with its definition pushed on to the stack to form a new stack element as in (6-4) and R_j becomes active. There is no chance that one of these particular expansions could be caused by other calls of some kind, and each compiled formula is generated by calling the exit definitions of the recursion points in a newly generated stack element. Because there is no redundancy in the recursive rule set (by our assumption in Chapter 3), there is thus no chance for a compiled formula to be generated more than once by the algorithm. Hence, the compiled formulas generated are non-redundant. □

6.2.4. Improving Processing Efficiency of the Generated Formulas

To improve the processing of the generated compiled formulas, several techniques can be considered.

An important one is to start from the most selective point and perform selection first. This reduces the size of the intermediate relations iteratively processed, hence saves processing cost. The value of *performing selection first* has been emphasized by Henschen and Naqvi's algorithm, verified by our performance testing reported in the previous chapter, and should be part of Algorithm ES. Moreover, if there are several query constants available, the processing should start at the most selective point determined by these constants.

A second important technique is the use of wavefront relations. Observing the generated compiled formulas, we find that there are many common subexpressions among the formulas, and thus it is beneficial to save processing results for the processing of other compiled formulas. Besides the wavefront ideas explored in previous chapters, we develop a new kind of wavefront: **Potential WAVE.**

The idea behind using wavefront relations is to prevent redundant processing by re-using intermediate processing results. In the processing of formulas generated by stack-directed compilation, not only should the intermediate results for the initial sub-expression preceding the *active* PRP (*wave*) be saved, but also those for the initial subexpression preceding *potentially active* PRP's (*potential waves*). The former is useful when the active point is not yet exhausted, while the latter are useful when the stack is popped and the active point shifted.

In Example 6.1, we apply these two kinds of wavefronts.

(1) The wavefront relation.

The function of the wavefront relation WAVE is similar to that for the algorithms studied in previous chapters. WAVE saves the intermediate processing results for the initial sub-expression up to but not including the active recursion point. When WAVE goes to empty, which indicates that the active point is exhausted, the stack is popped and the active point shifted.

(2) Potential wavefront relations.

In Example 6.1, we suppose that the wavefront relation σBBB is empty.

No.	The Compilation Stack	WAVE
4	B, B, B, R, C, R, D, C, R, D, C, R, D	σBBB
3	B, B, R, C, R, D, C, R, D	σBB
2	B, R, C, R, D	σB
1	R	

(a) Wavefronts for processing formulas generated at pushing

	The Compilation Stack	WAVE	1st P_WAVE
4	B, B, R, C, B, R, C, *R*, D, D, C, *R*, D	σBBACB	σBBACBAC
3	B, B, R, C, R, D, C, *R*, D	σBB	σBBAC
2	B, R, C, *R*, D	σB	σBAC
1	R		

(b) Potential wavefronts for processing formulas generated at popping

Table 6.2. The Benefits of Wavefronts and Potential Wavefronts

According to the algorithm, the stack is popped, and the active recursion point is shifted to the first potential recursion point. The top stack element becomes

B, B, R, C, R, D, C, *R*, D.

When the new active point is called, it generates the new stack element

B, B, R, C, B, R, C, *R*, D, D, C, *R*, D.

The new compiled formula generated is

σB, B, A, C, B, A, C, A, D, D, C, A, D.

The potential wavefront σBBAC is a common sub-expression of compiled formulas generated by stack elements No.3 and No.4 (See Table 6.2). The saving of σBBAC from the processing for the formula from No.3 speeds up the processing for the formula from No.4. The application of wavefronts and potential wavefronts can be summarized in the following algorithm.

Algorithm 6.2. The Optimization of Algorithm ES Using Wavefront and Potential Wavefront Relations.

(1) In the processing of a compiled formula, the processing result for the portion of the formula up to but excluding the active recursion point is saved as WAVE. The processing of the succeeding formula starts from WAVE instead of from the beginning.

(2) The processing result for the portion up to but excluding each potentially active recursion point of the compiled formula is saved as a **Potential WAVE**. When WAVE becomes exhausted, the stack is popped, and WAVE is replaced by use of the first Potential WAVE for the newly designated active recursion point. The processing of the next compiled formula starts from the new WAVE instead of from the beginning. □

Interestingly, from Algorithm ES we can rederive Algorithm SW. Because there is only one recursion point in an SLSR rule set, there is no *potential wavefront* to be saved, and once the stack starts popping there will be termination because the popping will continue until the stack becomes empty. Algorithm ES is in effect a generalization of Algorithm SW.

6.3. SUMMARY: STACK-DIRECTED COMPILATION OF COMPLEX RECURSIVE RULES

This chapter has introduced a stack-directed compilation method for compiling complex recursive rule sets. We presented a compilation algorithm for an SLMR rule set in acyclic databases. We have studied the required modifications of Algorithm ES for the cases of cyclic databases, multiple looping rules (Class IV), mutually recursive rules (Class V), multiple recursion levels. But we do not present those modifications here. The interested reader may find the details in [Han 85e].

In summary, we may conclude the following:

(1) Stack-directed compilation is a method for compiling complex recursive rule sets containing more than one resolution cycle. The method generates a non-redundant sequence of compiled formulas that will together produce all possible solutions.

(2) The manipulation of the compilation stack is dependent on execution of the compiled formulas. Hence the method is not "pure" compilation. It combines depth-first stack manipulation with set-oriented relational operations and thus makes the search more efficient than either alone would be.

(3) The two heuristics advocated in previous chapters apply to stack-directed processing as well. These are: (i) **perform selection first and start from the best selective point** and (ii) **make use of previous processing results.** The second heuristic is realized by using **wavefront** and **potential wavefront** relations.

Nevertheless, recursive processing in databases generally involves costly processing, and more efficient processing methods need to be explored. One solution to this problem is the **knowledge-directed compilation and planning** to be introduced and evaluated in the next two chapters.

CHAPTER 7

EXPERT KNOWLEDGE-DIRECTED RECURSIVE RULE COMPILATION

In previous chapters, we studied the compilation of function-free recursive rules without special augmentation of domain-specific expert knowledge. The techniques developed there are interesting for both theoretical and practical reasons. However, if we rely only on those techniques, we cannot adequately solve many practical application problems, because those problems often (i) contain function symbols in their recursive rule definitions, and/or (ii) involve costly iterative processing of large databases.

Such problems are often categorized as **database-oriented problem solving** problems. In artificial intelligence research, **problem solving** is the process of developing a structure of (in the simplest case, a sequence of) actions to achieve a goal, whose processing often involves application of search and planning techniques. **Database-oriented problem solving** involves recursion in relational databases and also demands application of such techniques.

In this chapter, we study the problems of and mechanisms for compiling recursive rules with the aid of domain-specific knowledge, and we demonstrate that *expert knowledge can play an important role in the correct and efficient processing of recursive database problems.* We confine our discussion to the transitive closure class (Class I queries) only because (i) most recursive rules in practical domains appear to fall in this class, and (ii) the study of this simple case can provide a base for investigation of more complex recursion patterns.

7.1. NEW PROBLEMS OF RECURSION IN DATABASE ORIENTED PROBLEM SOLVING

Recursion arises naturally in many applications that have to do with scheduling, planning, routing, and path-finding [Aho 79][Kung 84][Smit 84]. Practical applications pose new problems that cannot be solved with the conventional compiled approach. With an example we examine the issues raised in database-oriented problem solving. The air-flight reservation problem has served as a paradigmatic problem for several artificial intelligence (AI) research efforts. See, for example, [Wood 68].

Example 7.1. An air-flight reservation problem: Suppose there are two relations *flight* and *airport* in the database:

> *flight (fno, dpt, arr, dpttime, arrtime, fare)*
> *airport(port, size, lat, long)*

To schedule a flight from one airport to another *distant* airport, one-step simple retrieval is inadequate, and we must connect flights. This involves recursion on the *flight* relation. The recursive relation *new_flight* can be defined as in Figure 7.1 (in Prolog).

Besides a recursive rule definition, expert knowledge is usually available. For example, an air-flight administrator may have some regulations, such as that the lay-over time (the interval between the respective arrival and departure of two connected flights) must be within some range. A travel agent may use some "common sense" knowledge in scheduling flights, e.g., each flight should be in the same general direction as the direction from the initial departure to the final destination. These can be coded into knowledge rules to assist in problem solv-

```
new_flight (Flight_No, Departure, Arrival, Dpt_Time, Arr_Time, Fare) :-
        flight (Flight_No, Departure, Arrival, Dpt_Time, Arr_Time, Fare).

new_flight (Flight_No, Departure, Arrival, Dpt_Time, Arr_Time, Fare) :-
        new_flight (Flight_No sub 1 , Departure, Intermediate,
                                        Dpt_Time, Arr_Time, Fare1),
        flight (Flight_No sub 2 , Intermediate, Arrival,
                                        Dpt_Time, Arrival_Time, Fare2),
        Fare is Fare1 + Fare2.
        Flight_No is Flight_No1 & Flight_No2.†
```

Figure 7.1. Recursive Rules for Air-Flight Reservation

ing.

Moreover, a database user may impose many different query requirements interacting with recursive rules and knowledge rules. For example, suppose a customer wants to find inexpensive flights from Madison to Shanghai where the departure time is between 8:00 am and 11:00 am, the total flight time is under 30 hours, and the fare is less that $1000. In QUEL

```
range of  x is  new_flight
retrieve (x.dpttime, x.arrtime, x.fare)
where x.dpt = "Madison" and x.arr = "Shanghai" and
        x.dpttime > 8 and x.dpttime < 11 and
        x.arrtime - x.dpttime < 30 and x.fare < 1000
```

It is an interesting problem to compile such a query using domain-specific knowledge. □

† & is a concatenation operator which forms a new *Flight_No* by concatenating *Flight_No*$_1$ and *Flight_No*$_2$.

The compilation methods studied in previous chapters are not adequate for solving this problem, because it generates new problems including *functional definition and its termination problems, use of search constraints, interaction among query constants, finding only some solutions vs. finding all solutions*, and *the preserving of high level views and high level query interfaces.*

7.1.1. Evaluable Functions in Recursive Rule Definitions

We assumed that there were no evaluable functions in the recursive rule definitions studied earlier (see Chapter 3). The primary reason for such an assumption was to avoid termination problems. However, in practice, evaluable functions do appear in many database-oriented recursive problems (Table 7.1 which is based on [Daya 85]). In the *new_flight* rules of Example 7.1, there are two evaluable functions: summation "+" and concatenation "&".

Application Problem	Property	Function	
		Aggregate	Concatenate
shortest path	length/time	min	+
critical (longest) path	length/time	max	+
maximum capacity path	capacity	max	min
most reliable path	reliability	max	*
Bill of materials	item count	+	*
List all paths	edge name	∪	&
List any paths	edge name	choose_any	&

Table 7.1. Evaluable Functions in Recursion Problems

Fare is $Fare_1 + Fare_2$.
*Flight_No is Flight_No*1 *& Flight_No*2.

A deductive database with such recursive rules is essentially a database with new tuples iteratively generated using recursive rules. *New_flight* defines new data which are not stored in the database. If there is no constraint or bound, the generic data relation *new_flight* may grow indefinitely and never terminate, because one could fly around the globe indefinitely many times or fly back and forth indefinitely between two intermediate points before settling down and going on to the destination.

7.1.2. Search Constraints in Recursive Rule Compilation

Recursion in large databases usually requires considerable processing power. However, if we do not further reduce the search space, processing may still be too expensive for practical applications.

This can be seen from Example 7.1. Suppose (i) there are 10^5 tuples in the relation *flight*; (ii) the maximum acceptable propagation cycle (the number of edges in a flight path) from Madison to Shanghai is 20; (iii) at the beginning there are 10 tuples selected; and (iv) the average join selectivity is 10^{-3}, i.e., the second iteration will generate $10*10^5*10^{-3} = 1000$ tuples. The total number of tuples processed in 20 iterations will be : $10 + 1000 + \dots + 100^{20} = 10 * (100^{20} - 1)/(100 - 1) \simeq 10^{39}$, a number too huge to be processed with a reasonable amount of processing power in a reasonable time.

Recursion on large databases tends to run into combinatorial explosion. Combinatorial explosion is a major challenge in both AI and database-oriented problem solving. But a DB problem solver usually searches an even larger search space than an AI problem solver, due to the breadth-first flavor of DB operations that arises from exploring all possible solutions. Most AI problem solvers use various heuristics to reduce the large search space. Obviously, use of search constraints is critical in DB-oriented problem solving as well.

7.1.3. Query Constants in Recursive Rule Compilation

A recursive query processor should be ready to deal with various kinds of query constants, which may be related to the initial condition (e.g., the departure port and time), the final condition (e.g., the arrival port), and the relationship between the two (e.g., the total flight time and fare). For example, in the query of Example 7.1, we have several query constants,

x.dpt = "Madison" and x.arr = "Shanghai" and x.dpttime > 8
and x.dpttime < 11 and x.arrtime - x.dpttime < 30
and x.fare < 1000

The use of the query constants in recursive processing is a non-trivial problem. Query constants should be used as early as possible to reduce the search space, but they can not be indiscriminately used at the beginning of iteration. For example, the minimum fare *x.fare > 800*, if user poses such restriction in the query, can not be used until the *final* stage, otherwise most possible answers will be cut off at early iterations; while the maximum fare *x.fare < 1000* must be used at each iteration to terminate those flight paths with accumulated fares exceeding $1000.

In the following sections we show how a recursive rule compiler can use query constants at the right time and correctly.

7.1.4. Finding All Solutions vs. Finding Some Solutions

In the previous chapters we have developed algorithms which exhaustively search for all solutions for a recursive query. However, many practical application problems, such as those listed in Table 7.1, require finding just a small portion of all solutions, and it is certainly *not* the case that the best way to find one or a few solutions is always to find all possible ones and then to select from among them. When addressing the problem of finding only some from among all possible solutions, one can often use various search constraints, planning techniques, and other kinds of heuristics that expedite finding solutions at the cost of ruling out some possible solutions.

For example, in the query of Example 7.1, *all* the flights from Madison to Shanghai include flights via tiny intermediate ports, flying back-and-forth, etc. Most users are not interested in seeing such solutions although they would definitely be included in the set of all possible solutions.

7.1.5. User Transparency

An important advantage of a relational interface over network or hierarchical database interfaces is a high level view of databases and a high level query interface. Considering high level relational query interface, recursive extension is preferable to the procedural extension of relational languages [Aho 79] in processing of least-fixed-point queries. When applying domain-specific knowledge, we will encounter the problem of specifying and using diverse kinds of knowledge while preserving a high level query interface.

7.2. A SOLUTION: KNOWLEDGE-DIRECTED RECURSIVE RULE COMPILATION

Our solution to these problems is **knowledge directed recursive rule compilation.** In this section, we study compilation using **termination constraints, search constraints,** and **query constants.** A modularized knowledge-directed recursive rule compiler is presented in the next section.

7.2.1. Termination for Monotonic Functions

The termination problem for recursion containing evaluable functions can frequently be solved by using bound information and characteristics of a monotonic function. Such information is available in many application problems.

7.2.1.1. Monotonic Functions and Termination Constraints

In a function definition, if the value of the function increases or decreases monotonically when the value of the function argument increases, it is a **monotonic function**; otherwise, it is a **non-monotonic function**. For example, $f(x) = 2 * x + a$ and $f(x) = 1 - x^3$ are monotonic functions, while $f(x) = random(x)$ is a non-monotonic function. Even for some nonnumeric functions, there may still exist some monotonic characteristics of the function. For example, concatenation of lists makes the length of the resulting list monotonically grow, so we consider concatenation a monotonic function.

In Example 7.1, *fare* is defined by a monotonic function,

$$Fare = Fare_1 + Fare_2$$

Suppose a user specifies an upper bound in the query, such as

$$x.fare < 1000$$

Because iterative computation makes *fare* monotonically increase, a generated tuple with fare beyond $1000 should be dropped since further exploration will never meet the constraint $x.fare < 1000$. Based on monotonicity characteristics, the more iteration, the more generated tuples will be dropped from the *newflight* buffer relation. There must be a time when all tuples have been dropped from the buffer and it becomes empty. That is the termination point. Bound information is useful at the termination of a monotonically increasing or decreasing function. Thus we call such bound information a **termination constraint**.

Theorem 7.1. If a recursive rule contains an evaluable function definition on an attribute, and the function value increases or decreases monotonically when the attribute value increases, the search in the database using the recursive rule terminates when the attribute value of every driver tuple increases or decreases beyond a termination constraint.

Proof Sketch

(1) According to the recursive rule compilation discussed in previous chapters, the compilation process transforms recursion into iterative execution. Suppose the attribute value monotonically increases in iteration. At some iteration, the attribute value of a driver tuple surpasses the upper bound. Then, future iterations can only generate values greater than the upper bound. That is, further exploration from this tuple cannot generate any new value which meets the query requirements.

(2) According to the monotonic behavior of the attribute functional definition, iteration makes every driver tuple grow. Eventually, all tuples will grow beyond the upper bound. If all driver tuples are beyond the bound, there will be no new tuple generated by further iteration. The process terminates at this point. □

However, there is no such termination point for non-monotonic functions. For example, for the function *f(x)* = *random(x)* each call will randomly generate a number. No *termination constraint* can be applied to eliminate further exploration.

7.2.1.2. The Specification of Termination Constraints

There are several ways to provide termination constraints for monotonic functions.

(1) User's query implied: e.g., *x.fare < 1000* in Example 7.1. It is important to incorporate query constants in termination judgement, if at all possible.

(2) Expert specified: A termination constraint may be specified by a domain expert.

(3) System exception: A termination constraint may be determined by a specific system exception. For example, *maxint* (the maximum integer) can be used by a system exception handler to terminate iteration.

A termination constraint may be discoverable from databases with monotonic data distribution. For example, the *birth_year* of a *person* is always greater than that of his or her *parents*, thus forming a monotonic data distribution in the *person* relation. When a query asks for all John's ancestors born in the 18th century, search following any tuple with *birth_year* value less than 1700 should terminate. Without utilizing a termination constraint, search still terminates when all paths are explored, but it would be less efficient than with the application of the termination constraint.

7.2.1.3. Determination of Monotonicity in a Database

There are two methods for determining monotonic characteristics in a database: **derivation** and **specification**.

(1) Derivation: Derive monotonicity of a function definition or data distribution by deduction from integrity constraints and recursive rule definitions.

For example, the monotonicity of the *birth_year* attribute can be determined from the integrity constraint,

```
range of p1 is person
range of p2 is person
define integrity p1.birth_year > p2.birth_year
        where p1.fa = p2.name or p1.mo = p2.name
```

For the virtual relation *new_flight*, using the integrity constraint

> *range of* f *is* flight
> *define integrity* f.fare > 0

and the new_flight definition where $Fare = Fare_1 + Fare_2$, we can conclude that $Fare > Fare_1$ and $Fare > Fare_2$, which means that the summation of *Fare* is monotonic.

(2) Specification: Explicitly specify monotonicity of a function. For example, the keywords **upper bound** and **lower bound** specify that a function is monotonically increasing or decreasing. The specification *upper bound f.fare* specifies that *f.fare* is a monotonically increasing function. This method is adopted in our language for RELPLAN.

7.2.2. Use of Search Constraint Rules

Search constraint rules are constraints applied in iterative search. They represent restrictive information which, if used appropriately, may considerably reduce search space.

In Example 7.1, we have two search constraint rules: (1) *same-direction constraint*: the new flight path should have the same general direction as from the start location to the final destination; and (2) *lay-over time constraint*: the transfer time between two consecutive flights should be within some range.

In general, a constraint rule does not preserve the possibility of finding all possible solutions derivable from other rules above, because it modifies the rules by restricting the set of possible solutions. In database-oriented problem solving, there are usually many constraint rules. The use of a sequence of constraint rules is equivalent to incremental modification of recursive rules. The more constraint rules applied, the more precise knowledge provided, and the smaller search space to be explored. An important task is to find constraint rules and selectively use them to keep database search within a reasonable cost and solution range.

The general mechanism of constraint use in recursive rule compilation is similar to the ADDQUAL mechanism described in [Ston 84], but here we discuss some techniques for optimal processing.

When a knowledge rule is added to a recursive rule set, it often generates what in effect is a loop invariant. Similar to the loop optimization technique in compiler construction, a loop invariant should be moved outside of the loop instead of repeatedly processed inside the loop. For example, for the looping rule

$$R(x,y,t):-A(x,z,t),R(z,y,t). \tag{7-1}$$

if a query has the form

$$?-R(x,y,a). \tag{7-2}$$

a selection should be first performed on the base relation A to reduce the size of the relation to be iteratively processed. After performing the selection, the looping rule becomes

$$R(x,y):-A'(x,z),R(z,y). \tag{7-3}$$

where A' results from performing selection on A, i.e.,

$$A'(x,z):-A(x,z,a). \tag{7-4}$$

and is usually much smaller than A.

As another example, if we have a constraint C(z, b) added to (7-1), the augmented rule becomes

$$R(x,y,t) :- A(x,z,t), R(z,y,t), C(z,b). \tag{7-5}$$

The constraint should be computed before entering the loop to reduce the size of the relations to be iteratively joined. Thus we have

$$R(x,y,t) :- A'(x,z,t), R(z,y,t). \tag{7-6}$$

where

$$A'(x,z,t) :- A(x,z,t), C(z,b). \tag{7-7}$$

A practical example of such optimization is that in the air-flight reservation example, if the constraint is flying via big-ports only, the *big-port* restriction should be performed on the relation *flight* before starting iteration, to reduce the size of the relation to be iteratively processed.

$$\begin{aligned} &BigFlight(fno, dpt, arr, dpttime, arrtime, fare) :- \\ &\quad Flight(fno, dpt, arr, dpttime, arrtime, fare), \\ &\quad Airport(arr, port\_size, lat, long), \\ &\quad port\_size > 10. \end{aligned} \tag{7-8}$$

7.2.3. Use of Query Constants in Recursive Database Search

Query constants play an important role in reducing iterative search in databases. Query constants can be applied appropriately at different points: **start, iteration,** and **termination.**

In the query of Example 7.1, query constants

x.dpt = "Madison" *and* x.dpttime > 8 *and* x.dpttime < 11

should be used at the start point to restrict the number of tuples to start with, while

x.fare < 1000 *and* x.arrtime - x.dpttime < 30

should be applied in the iteration to cut off tuples out of bounds, and x.arr = 'Shanghai'

should be applied at the end of iteration to select the qualified answers.

In general, suppose we have a rule

$$R(x,y,f,t):-$$
$$A(x,y',f',t),R(y',y,f'',t),F(f,f',f''). \tag{7-9}$$

where F is a function definition which defines f as dependent on the values f' and f''.

We divide the variables of a recursive rule into five different classes: *B-var, F-var, L-var, R-var,* and *T-var.* We call the extensional literal (A) in the antecedent **the extensional literal,** and the intensional literal (R) **in the antecedent the recursive literal.**

(1) A **B-var** is an **induction base variable** contained both in the consequent literal and the extensional literal, e.g., x.

(2) An **F-var** is a **function variable** contained in the evaluable function definition, e.g., f, f' and f''.

(3) An **L-var** is a **link variable** which is not in the consequent literal but links the extensional literal and the recursive literal in the antecedent, e.g., y' .

(4) An **R-var** is a **recursive variable** contained in both the consequent literal and the recursive literal, e.g., y.

(5) A **T-var** is a **through variable** which passes through the consequent, extensional and recursive literals in a recursive rule definition, e.g., t.

Observe an iterative expansion of the above rule

$$
\begin{aligned}
&R(x,y,f,t) :- \\
&\quad A(x,y',f',t), \\
&\quad A(y',y'',ff,t), \\
&\quad R(y'',y,f'',t), \\
&\quad F(fi,ff,f''), \\
&\quad F(f,f',fi).
\end{aligned}
\qquad (7\text{-}10)
$$

We see that during the expansion: (i) B-vars always remain in the first extensional literal; (ii) R-vars always remain in the recursive literal (if the intensional literal is replaced according to an exit rule, the R-vars will be in the exit literal); (iii) F-vars form a sequence of applications of the function definition; (iv) L-vars do not appear in the consequent literal, hence will not interact with query constants; and (v) T-vars always remain in every expanded literal.

For efficient processing, we may start from query constants in the position of either B-vars or R-vars, whichever is the most selective point, and terminate the iterative search using F-vars. The query constants in the position of R-vars or B-vars that were not used for starting selection will be used for final selection. For a T-var position, if it does not contain a query constant, and it is not contained in the query results, it is useless and should be projected off before iteration; otherwise, if it contains a query constant, we should use the query constant to reduce the size of the base relation to be iteratively processed. The above discussion is summarized in the following algorithm.

Algorithm 7.1. Use of Query Constants in Compiling Transitive Closure Rules.

(1) Classify the variables in the consequent literal into four sets: B, F, R, and T.

(2) If a query constant is in the T set, perform selection on the base relation thus using the constant before iteration. Project off T-var positions before iteration starts, if it is not contained in the final query result.

(3) Choose the best selective point by comparing the selectivity on data relations of query predicates involving constants in the B set and in the R set. Set the winner as **start set** and the loser as the **end set**. To start the iteration, use the query constants in the **start set**.

(4) At each iteration, use the termination predicate (determined by the F set). If there is no query predicate involving constants in the F set, use expert-specified termination constraints. Collect solutions which satisfy the termination predicate into the result buffer.

(5) After termination, the answers for the recursive query are obtained by using the query constants in the **end set** to select from the result buffer those tuples which satisfy the query. □

The algorithm is based on the discussion presented in this section, and it implements the knowledge-directed compilation of RELPLAN presented in the next section.

7.3. MODULARIZATION OF DEDUCTIVE PROCESSES

In this section we present a knowledge-directed recursive rule compiler, which is a primitive implementation of the principles discussed in the last section and which preserves a high level relational query interface with a syntax similar to that of QUEL. It constitutes the knowledge-directed compilation part of the RELPLAN project.

The design philosophy of the recursive query compiler has focused on the modularization of the specification of recursive rules, search constraints, termination constraints, and other aspects of recursive rule compilation. **Deductive modules** serve to classify diverse expert knowledge, realize search and termination constraints, isolate the interaction of different rules, and preserve high-level view and high-level query interfaces.

7.3.1. The Architecture of the Relational Planner RELPLAN

The architecture of RELPLAN is presented in Figure 7.2. The motivation for developing such a relational planner has been to experiment with knowledge-directed compilation and planning mechanisms for compiling recursive database queries in relational DB systems.

RELPLAN uses expert knowledge to transform users' deductive queries into non-deductive query programs. Expert knowledge is coded in the form of rules and entered into a rule base consisting of global rules and local rules. The global rules are available for all scopes of deductive queries, while local rules are confined to deductive and plan modules to be used only by queries that explicitly reference these modules. The transformation of a deductive query into a non-deductive query program is based on the resolution principle. The output query program of RELPLAN can be sent to query optimization routines to generate database access plans.

RELPLAN is implemented in C, using YACC running under UNIX. The RELPLAN grammar, using extended BNF, is specified in Appendix I.

Like other database languages, RELPLAN contains a data definition part and a data manipulation (query) part. To ensure a high level query interface, the RELPLAN query language is defined to be the same as the relational query language QUEL, except that the entities that queries reference may also be

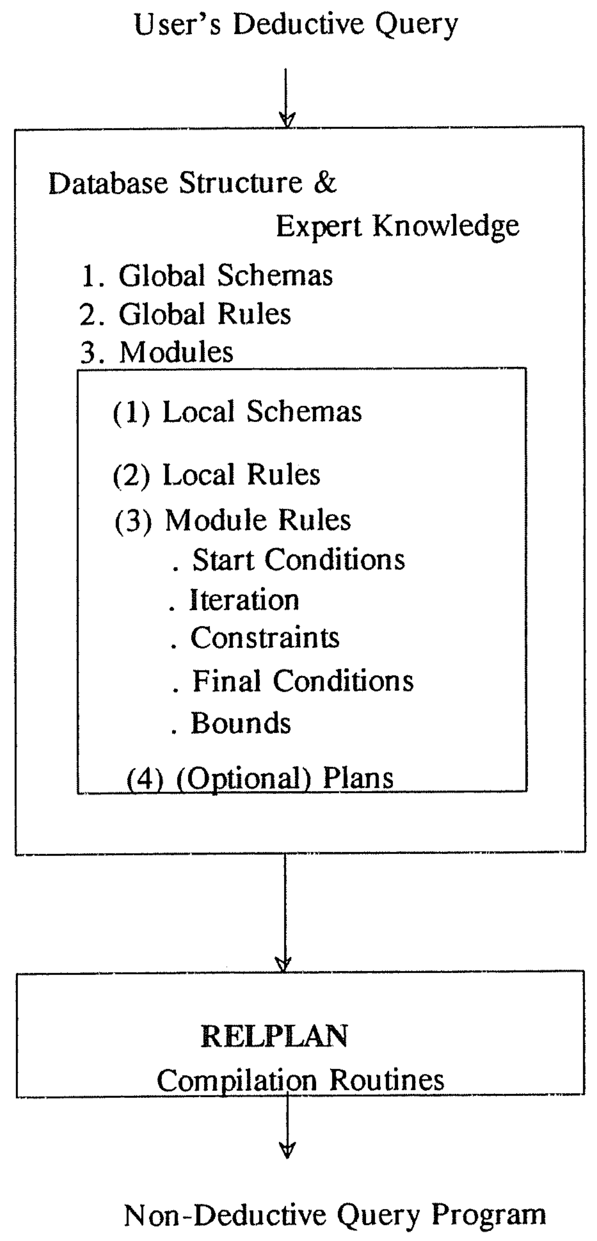


Figure 7.2

The Architecture of the Relational Planner RELPLAN

deductive modules. The data definition part, where rules and modules are specified, is the major enhancement over other relational languages. Rules are specified as virtual relation definitions, search constraints and other module rules (such as *start, iteration, bound, etc.*) in deductive modules. A deductive module contains (i) the specification of local schemas (for temporary generic relations to be used during the problem solving process) and local rules, (ii) module rules, and (iii) an optional planning section which contains local specification and planning steps. Each planning step contains planning rules which append, delete or modify the corresponding module rules in the deductive module.

When processing a query which references a deductive module, RELPLAN uses information provided in the query along with rules in the deductive module to decide which planning strategy should be adopted and what constraints should be used during the problem solving process. The query is then resolved by using the knowledge provided in the rule base and/or in the deductive module. A query program is generated with all virtual relations resolved and ready for further processing in a relational database system.

7.3.2. Development of Deductive Modules

A deductive module is a module that consists of a group of rules, schemas and built-in queries which form a problem solving package. It modularizes the rule system and focuses the problem solving process on a small group of rules, thus reducing search effort devoted to rule invocation as well as minimizing interaction among different rules and goals.

A deductive module can be viewed by a database user as a virtual relation. The module defines the procedures according to which instances of the virtual relation are obtained.

RELPLAN follows scope rules similar to the scope rules of conventional block-structured programming languages, such as Modular. Rules, schemas and built-in queries defined inside a module can be referenced and executed only by the rules and queries inside the same module. User queries which reference the module relation treat the module as an "abstract" virtual relation. No individual rule or relation inside the module can be referenced by the user's query. Rules outside the module cannot reference rules inside. Rules inside the module can reference rules in the global rule space but not those in other modules. There can be declared and used inside the module a variable that references the entire virtual (*module*) relation.

To facilitate iterative rule processing, which is a major motivation in the design of deductive modules, a sequence of module rules can be defined within a deductive module. The sequence specifies *start condition, iteration, final condition, search constraints* and *upper* and/or *lower bounds.*

Example 7.2. Specification of the deductive module *flight* for the air-flight reservation problem.

Constraints and heuristic rules proposed by database administrators or experts are considered as query independent knowledge which should be specified inside the module †, while user's queries are considered as dynamic requirements to be specified outside. The deductive module is presented in Figure 7.3.

The module *flight* contains several components: (1) definition of a local generic relation *new_flight*; (2) local rules, e.g., the constraint rule

† Rules inside a module can be modified, added or deleted by experts using primitives similar to those for plan rules discusssed in the next chapter. This feature is not hard to add but is not included in our prototype implementation.

```
schema flight( fno, dpt, arr, dpttime, arrtime, fare)
        airport(port, lat, long, size) /* lat : latitude, long : longitude */

module flight

schema new_flight( fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)

range of mf is module flight
range of f  is flight
range of n  is new_flight
range of p0, p1, p2, p3 is airport
define constraint s : same_direction(dpt1, arr1, dpt2, arr2) is
                (p0.lat - p1.lat) * (p2.lat - p3.lat) > 0 and
                (p0.long - p1.long) * (p2.long - p3.long) > 0
        where s.dpt1 = p0.port and s.arr1 = p1.port and
                s.dpt2 = p2.port and s.arr2 = p3.port

start - >       retrieve into new_flight (f.fno, 0, f.fno, f.dpt, f.arr,
                        f.dpttime, f.arrtime, f.fare) where f.dpt = mf.dpt

iteration - > retrieve into new_flight (n.fno & f.fno, n.fno,
                f.fno, n.dpt, f.arr, n.dpttime, f.arrtime, n.fare + f.fare)

constraint - > n.arrtime + 3 > f.dpttime and n.arrtime + 1 < f.dpttime

constraint for iteration - > same_direction(f.dpt, f.arr, mf.dpt, mf.arr)

upper bound - >     (1) mf.fare     (2) mf.arrtime

end module
```

Figure 7.3. Example of A Deductive Module

same_direction; and (3) a sequence of module rules to specify initialization, iteration, final states, constraints and bounds.

The generic relation *new_flight* is used to iteratively generate flight paths during the problem solving process. The local rules such as *same_direction* are used for performing inference inside the module. The module rules include: (1) general constraints, e.g., the lay-over time between two flights should be between 1 to 3 hours *(n.arrtime + 3 > f.dpttime and n.arrtime + 1 < f.dpttime)*, and constraints for iteratively connecting the consecutive flights, e.g., flying in the same general direction as the initial departure and the final arrival posed in the query, i.e., *same_direction(f.dpt, f.arr, mf.dpt, mf.arr)*; (2) initial state: the portion of the base relation *flight* which has the same initial departure as the user's query; (3) iteration rule: iteratively connecting base relation flights to obtain *new_flight*, where the departure of the tuples in *flight* is the same as the arrival of the tuples in the generic relation *new_flight*, and (4) bound rules, used for terminating the iteration and for implicit control of constraint use. In this example, we specify that there must be an upper bound rule for fare or arrival time. □

7.3.3. Compiling Deductive Queries Using Deductive Modules

Using a deductive module, a user query can be compiled into an iterative query program with the rules in the module appropriately used. The rules inside the module use the query constants to perform deduction and other query modification. The output is the modified query program, with rules resolved, query constants used, and termination conditions determined. Such a compilation process is called **knowledge-directed compilation**. Here we illustrate the compilation process with the air-flight example.

Example 7.3. Compilation of an air-flight reservation query using the *flight* module.

Suppose a user wants to book a ticket from Madison to Shanghai. The fare requested is less than $1000, and the total travel time is required to be less than 30 hours. The database query is formulated in QUEL and shown in Figure 7.4. The resolved query program produced by RELPLAN is shown in Figure 7.5.

The compilation process proceeds as follows.

(1) Selection of the deductive module. The user's query

```
range of x is flight
retrieve (x.dpttime, ...)
     where ...
```

references the relation *flight* which is a deductive module, hence that module is selected.

(2) Initialization: First we retrieve the data in the database which immediately satisfy the query,

```
retrieve ( x.dpttime , x.arrtime , x.fare )
     where x.dpt = "Madison" and x.arr = "Shanghai"
     and x.fare < 1000 and x.arrtime - x.dpttime < 30
```

Then we apply the start rule to initialize the iteration, with the start variables *x.dpt = "Madison"* and termination constraints *x.fare < 1000 and*

```
range of x is flight
retrieve (x.dpttime, x.arrtime, x.fare)
where x.dpt = "Madison" and x.arr = "Shanghai"
     and x.fare < 1000 and x.arrtime - x.dpttime < 30
```

Figure 7.4. RELPLAN Input: A User's Deductive Query

```
range of x is flight
range of n is new_flight

retrieve ( x.dpttime , x.arrtime , x.fare )
      where x.dpt = "Madison" and x.arr = "Shanghai"
      and x.fare < 1000 and x.arrtime - x.dpttime < 30

retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)
      where n_.fno = x.fno and n_.f1no = 0 and n_.f2no = x.fno
            and n_.dpt = x.dpt and n_.arr = x.arr and
            n_.dpttime = x.dpttime and n_.arrtime = x.arrtime and
            n_.fare = x.fare and x.dpt = "Madison" and
            n_.fare < 1000 and n_.arrtime - n_.dpttime < 30

loop

range of p0 is airport
range of p1 is airport
range of p2 is airport
range of p3 is airport
retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)
      where n_.fno = n.fno * 1000 + x.fno and n_.f1no = n.fno
      and n_.f2no = x.fno and n_.dpt = n.dpt and
      n_.arr = x.arr and n_.dpttime = n.dpttime and
      n_.arrtime = x.arrtime and n_.fare = n.fare + x.fare and
      n.arrtime + 3 > x.dpttime and n.arrtime + 1 < x.dpttime
      and p0.port = x.dpt and p1.port = x.arr and
      p2.port = x.dpt and p3.port = x.arr and n_.fare < 1000
      and n_.arrtime - n_.dpttime < 30
      and ( p0.lat - p1.lat ) * ( p2.lat - p3.lat ) > 0
      and ( p0.long - p1.long ) * ( p2.long - p3.long ) > 0

retrieve ( n_.dpttime , n_.arrtime , n_.fare ) and delete n_ : new_flight
      where n_.dpt = "Madison" and n_.arr = "Shanghai"
      and n_.fare < 1000 and n_.arrtime - n_.dpttime < 30

exit when new_flight is empty

end loop
```

Figure 7.5. RELPLAN Output: The Resolved Query Program

x.arrtime − x.dpttime $<$ *30* used.

(3) The iteration part is enclosed inside a *loop ... (loop body) ... end loop* statement and it is divided into three parts: (i) use the iteration rule of the module with search constraints and termination constraints (the bound part of the user's query) added to generate new tuples, which are placed into a generic relation; (ii) retrieve and delete from the generic relation the tuples which satisfy user query requirements; and (iii) terminate when the generic relation becomes empty.

The algorithm is summarized as follows,

Algorithm 7.2. Compilation of a Deductive Query Using a Deductive Module.

(1) The selection of a deductive module.

A deductive module, viewed by a database user as a virtual relation, is selected when the query references the module relation.

(2) Initialization:

(i) retrieve the data in the database which immediately satisfy the query.

(ii) initialize a generic relation by modifying the start rules as following: (a) take the start rule as the central rule, (b) add query constants that match the referenced variable of the start rule, (c) add query constants that match the termination constraints, and (d) add search constraints if they are specified to be applied to the start rule; and then execute the modified rules to derive the relation.

(3) Iteration:

If there is no iteration part in the deductive module because it has been deleted (see Section 8.3), the module is non-iterative and there is nothing generated for the iteration part.

The iteration part will be enclosed in a *loop .. end loop* statement which contains the following:

(i) retrieve data into the generic relation by modifying the iteration rules as following: (a) take the iteration rule as the central rule, (b) add the query constant that matches the referenced variable of the iteration rule, (c) add the query constant that matches the termination constraints, and (d) add search constraints if they are specified to be applied to the iteration rule. The modified iteration part becomes a query with the format *retrieve into temporary relation*, which will be iteratively executed.

(ii) Tuples in the generic relation satisfying the query requirements are retrieved as portions of the answer and removed from the generic relation.

(iii) Iteration terminates when the generic relation becomes empty. □

7.4. SUMMARY: KNOWLEDGE-DIRECTED RECURSIVE RULE COMPILATION

The knowledge-directed compilation technique we have studied can be summarized as follows:

(1) Knowledge-directed compilation incorporates domain-specific knowledge in recursive rule compilation to reduce the cost of database search.

(2) Functional symbols which cause termination problems in recursion can be dealt with by using termination constraints and the monotonicity of functions;

(3) Search constraints can be used to add more qualification requirements in modifying recursive rules, hence serve to focus search.

(4) Query constants are useful in performing selection first by using the most selective information, determining termination, and reducing size of relations to be iterative executed.

(5) A modularized knowledge-directed compiler has been realized in REL-PLAN to perform knowledge-directed compilation while maintaining a high-level query interface.

CHAPTER 8

PLANNING IN DATABASE ORIENTED PROBLEM SOLVING

In this chapter, we further investigate knowledge-directed recursive rule compilation by developing a planning technique for compiling recursive queries in relational databases. Planning is a hierarchical augmentation of recursive rules with knowledge. We develop a two-phase planning technique in our relational planner: the first phase is **determination of an appropriate plan strategy**, and the second phase is **generation of the query program according to the selected strategy**. The two phases (**plan selection** and **plan generation**) constitute the *planning section* of a modularized **plan module** in RELPLAN. The function of planning is demonstrated in an example, and analysis shows that planning may drastically reduce the cost of iterative accessing of a relational database.

8.1. PLANNING: A NECESSARY TECHNIQUE FOR DATABASE-ORIENTED PROBLEM SOLVING

By planning we mean the development of a representation of a course of actions before acting to solve a problem. Planning is widely used in AI problem solvers [Sace 77][Nils 80]. For complex problem solving in expert database systems, planning will also be a necessity. This is illustrated by the air-flight reservation problem.

In the air-flight reservation problem, if a traveller wants to fly from a small port to another small port a long distance away, experience suggests that we schedule the flight path as follows: first fly from the departure airport to a neighboring big airport, then fly to a big airport near the destination *via big airports* *only* and in the same direction as from the start to the final destination. The final flight should be the flight from this adjacent airport directly to the final destination. Because most small airports are ignored in such scheduling, the search space is reduced considerably.

This scheduling technique results from a quite useful planning strategy called **means-ends analysis** [Newe 72], which compares the goal with the currently achieved state, extracts a difference between them, and then selects a relevant operator to reduce the difference. The small-big-big-small flight plan is a hierarchical representation that divides the search space into different search space, uses different search constraints at different search space, and avoids consideration of most small ports in scheduling long distance travel.

There are at least two alternative approaches to developing a planning process: a **top-down approach** and a **bottom-up approach**. In the top-down approach, we first find a path of consecutive flights from a big airport near the initial departure airport to a big airport near the final destination, then find local flights connecting to these big airports. In the bottom-up approach, we first find a flight from the initial departure to a neighboring big airport, then find a path of flights from this airport to a big airport near the final destination, etc. We demonstrate our planning mechanism using the bottom-up approach.

8.1.1. Different Kinds of Knowledge: Constraints, Plans, and Heuristics

A planning process is guided by planning rules: rules that capture planning knowledge. Similar to AI problem solving, database oriented problem solving needs various kinds of knowledge rules. In the last chapter, we studied the use of constraint rules in recursive rule compilation. In this chapter we study the specification and application of planning rules. Heuristic rules will be briefly discussed as raising future research issues.

Planning rules and heuristic rules are different from constraint rules. A **constraint rule** is a uniform application of expert knowledge across the entire scope of the recursive rules. In effect, it serves to confine and thus reduce the total search space.

A **planning rule** is hierarchical application of expert knowledge. A planning process partitions the search space for a recursive rule into a hierarchy. Each level in the hierarchy constitutes a separate search space and may involve different constraints and heuristics. For example, planning may divide the flight from Madison to Shanghai into two stages: first flying from Madison to a big nearby airport such as Chicago, or Minneapolis, then flying from that port to Shanghai via big ports only. In the different stages, database search strategies may apply quite different search constraints.

A **heuristic rule** represents knowledge to be applied for controlling heuristic search. It is neither uniform across the entire scope of the recursive rules, nor hierarchical in its effect upon the shaping of search. For example, if a user asks for an inexpensive flight, the problem solver may dynamically decide to explore some promising initial paths and prune away others that appear unpromising, based on cost estimates (i.e., estimates of final cost made before a path is completed and cost can be computed exactly). Other heuristic criteria on which pruning is based might include search depth, path length, accumulated search cost, the number of tuples generated for the next search step, etc. The application of heuristic rules to recursive database search is an interesting topic for future research.

8.1.2. Planning in Database-Oriented Problem Solving: A Hierarchical Search Mechanism

The partitioning of the search space into a structure of hierarchical, staged search spaces is called **planning** in the AI literature. In solving recursion problems for database systems, we need planning in just this sense, i.e., for partitioning the search space into a structure of hierarchical, staged spaces and then applying the appropriate constraints and heuristics for each partition.

Suppose we want to fly from Madison to Suzhou (a small city near Shanghai). The small-big-big-small planning strategy should be adopted, thus partitioning the original problem into three different stages: *small-big, big-big*, and *big-small*. The two local flights, *small-big* and *big-small* involve only one flight step each. The major search cost will be for finding *big-big* flight paths. To reduce this cost various constraints should be applied. If the problem is partitioned and the search for *big-big* flight paths is distinguished from the local searches, appropriate constraints can be applied at exactly those points where they will do the most good. There are at least two other advantages in proceeding this way:

(1) Smaller data relations are iteratively processed.

The data relation *flight* is confined to airports containing big ports only. A restriction like

$$(f.arr = p.port \text{ or } f.dpt = p.port) \text{ and } p.size > 10$$

should be executed before entering the loop, thus drastically reducing the size of the data relation to be iteratively joined.

(2) Fewer iterations.

Because each big-big flight flies a longer distance and consumes more time and money, fewer iterative joins need be performed to meet the given bound constraints.

8.2. PLANNING PHASES

In the air-flight reservation problem, the *small-big-big-small* strategy is only one of several possible strategies. The user may pose different queries that require different strategies. The best strategy is query and data dependent, and its selection is prerequisite to correct application of the planning mechanism. Thus we need first *select the appropriate planning strategy*, and then *generate the actual plan according to the selected strategy*. These constitute two separate phases in the planning process.

If a user wants only a local flight, say, from Madison to Chicago, the planner doesn't need to consider any hierarchical partitioning. If a user wants a flight from New York to Tokyo, the planner should construct flight paths via big ports only. If a user wants a cheap flight from Los Angeles to any city in northern England, the planner must consider arriving at both big and small ports in northern England. Different queries clearly require different planning strategies.

A flexible planning mechanism must be able to select different strategies for different situations. Similar to advice typically given for building expert systems, our advice is that the various plan strategies should be specified by problem domain experts (though we do not exclude the possibility that in some future development, the strategies may be discovered or learned by more intelligent systems). Our emphasis is on providing the experts with a formalism for specifying the strategies and on developing a mechanism for automatically selecting plan strategies appropriate for particular user queries and for the data stored in the database.

The selection of an appropriate strategy is not hard for a human expert. A travel agent can easily choose the best strategy according to the different requirements of his customers — because he has good knowledge of geography and of the air transportation system.

If a system is designed to automatically select the best strategy, it must possess the same kind of knowledge. Such knowledge is usually stored in the database or rule base, e.g., airport and air-flight information. We must retrieve and examine such information to determine an appropriate strategy. The retrieval and examination constitute the first distinct phase in planning. The generation of a query program for further database retrieval according to the selected strategy can only be done after a strategy is selected and constitutes a second distinct phase.

8.2.1. The First Phase: Selection of a Planning Strategy

Because a planning strategy is usually query and data dependent, the selection of an appropriate strategy should take into consideration both query constants and data in the database. In the air-flight reservation problem, there may be several planning strategies as in Table 8.1.

size_of_dpt	size_of_arr	distance	plan_strategy
any	any	<= 100 miles	local
> 10	> 10	> 100 miles	big_big
> 10	<= 10	> 100 miles	big_big_small
<= 10	> 10	> 100 miles	small_big_big
<= 10	<= 10	> 100 miles	small_big_big_small

Table 8.1. Different Planning Strategies for Different Queries

In the strategy selection phase, we still use set-oriented relational operations in order to facilitate implementation of the planner in relational database systems. We specify several tiny relations as **planning relations** using local schemas, with one relation for each planning strategy. For the air-flight example, there will be a separate such planning relation for each row in Table 8.1. During plan strategy selection, data is retrieved from global databases into the planning relations, based on query constants. This retrieval will generate one or several tuples for some planning relations. Such tiny relations can be stored in buffer space for immediate and efficient accessing. Strategy selection is based on which planning relation is non-empty.

If there is some planning relation which is non-empty, the corresponding strategy is selected and the planner uses the information contained in the non-empty planning relation to generate a detailed plan. If there is more than one non-empty planning relation, more than one planning strategy will be selected, and several plans will be generated according to the different selected strategies. This would be the case, for example, if a user wants to book some flights from Madison to any port in northern England. Shared processing can be explored for the coordination of several similar plans, but we do not pursue this possibility in this thesis.

For our air-flight example, we implement the first phase as follows: (1) specify the planning relations, one relation for each planning strategy; (2) based on the information provided in the query, retrieve airport information from the database into the planning relations; and (3) use the non-empty planning relations to determine the selection of planning strategies. For the example, we specify five *planning relations*:

(1) *Local (dpt, arr)*, which represents that the departure port is quite close to the arrival port and *only local scheduling is needed.* The other planning relations are for non-local flight schedules.

(2) *Big_Big (dpt, arr)*, which represents that both the departure and the arrival ports are big ports, and *scheduling flights via big ports only* is the correct strategy.

(3) *Big_Small (dpt, arr)*, which represents that the departure port is a big port but the arrival port a small one. The planner will suggest *flying to a big port near the destination via big ports only and then flying directly to the destination.*

(4) *Small_Big (dpt, arr)*, which represents that the departure is a small port and the arrival is a big one. The appropriate planning strategy is worked out accordingly.

(5) *Small_Small (dpt, arr)*, which represents the departure port is a small port and the arrival port a distant small one. The planning strategy for this case will be described here in more detail.

In the strategy selection phase, we use user query information to retrieve airport information into the five planning relations — *Local, Big_Big, Big_Small, Small_Big and Small_Small.* The retrieving queries are stored in the *planning section* of the *plan module.* One example query is as follows.

Example 8.1. A query specified in the plan module *flight* for plan selection.

Here we demonstrate a query in Figure 8.1 for plan selection which retrieves information into the planning relation, *Small_Small,* based on the user's query and data stored in the database.

```
range of p1, p2 is airport
retrieve into s : Small_Small(dpt = p1.port, arr = p2.port)
        where
                p1.port = mf.dpt and p2.port = mf.arr
                /* p1 and p2 are both small ports */
                and p1.size < 10 and p2.size < 10
                /* Two ports are located beyond local distance */
                and p1.lat − p2.lat > 5 and p1.lat − p2.lat > −5
                and p1.long − p2.long > 5 and p1.long − p2.long > −5
```

Figure 8.1. A Built-In Query for Plan Selection

In most cases, the retrieval in plan selection will result in only a small number of tuples being pulled into one of several planning relations. The query asking for flights from Madison to Suzhou will result in only one tuple in the *Small_Small* relation. □

8.2.2. The Second Phase: Plan Generation

The second phase of our planning is the generation of a query program based on the strategy selected and on the tuples in the planning relation. The non-empty planning relation indicates what strategy is to be selected, while the tuples in the planning relation provide more information based on query constants for generating the final plan.

For the query asking for flights from Madison to Suzhou, the planning relation *Small_Small* will end up containing a single tuple with two attributes *s.dpt = "Madison" and s.arr = "Suzhou"*. From this, (1) the strategy *Small_Small* should be selected, and (2) two query constants are provided for the generation of the final plan.

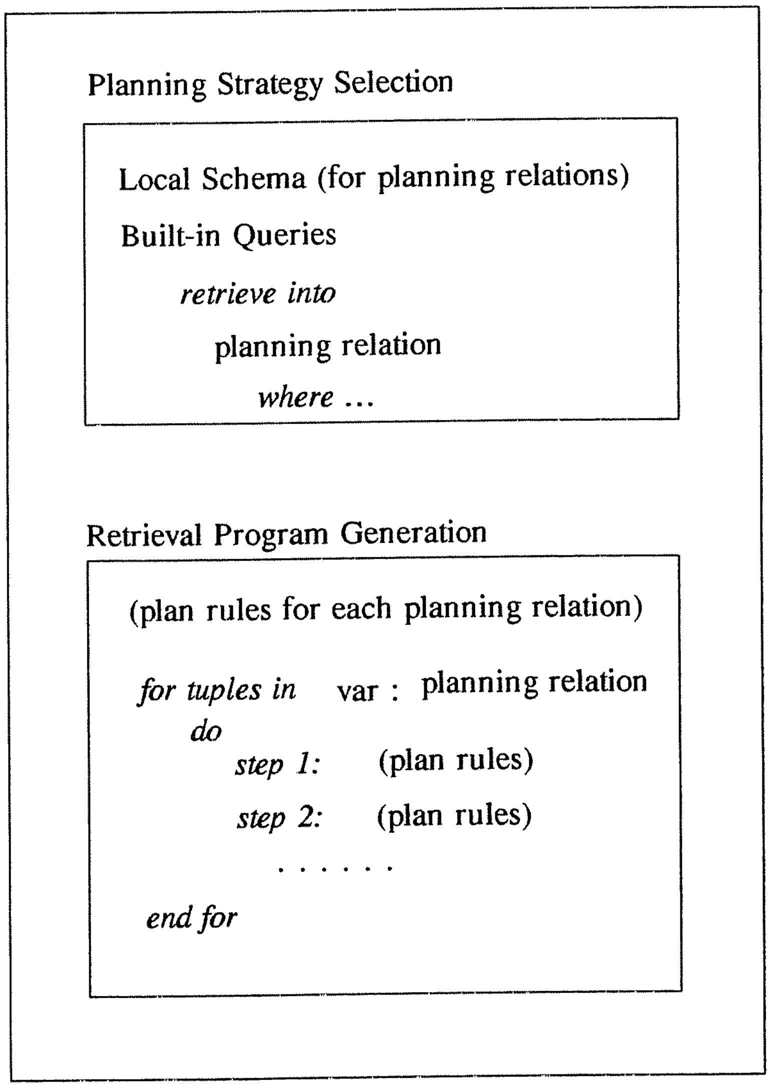
In RELPLAN, we use the following syntax to specify use of the selected strategy,

```
for tuples in variable: planning_relation_name do
        query program generation
end for
```

In the query program generation part, we specify several steps which correspond to the plan hierarchy. Each step defines specific constraints and heuristics for its search space. Each step consists of a set of rules which specify modification of rules stored in the non-plan part of the deductive module. The modification consists of adding constraints and changing start condition and/or final conditions. After the plan is selected, the modification rule for each hierarchy level determined by the selected plan is executed. The execution generates a compiled query program based on modified rules and query constants.

8.2.3. The Function of the Plan Module in Plan Generation

In RELPLAN, planning is prescribed with a plan module. A **plan module** is a deductive module augmented with a planning section at the end of the module. The planning section is divided into two parts for the two-phase planning. The first part consists of a set of query statements which retrieve information for the selection of the planning strategy. The second part consists of one or more planning steps for each possible planning strategy. Each such step serves to modify some module rules of the deductive module that precedes the planning section. The modification instructions are coded using syntax similar to that of QUEL. The syntax for the planning section is also specified in the extended BNF in Appendix 1. Figure 8.2 is the structure of the planning section of RELPLAN.



```
Planning Strategy Selection

    Local Schema (for planning relations)
    Built-in Queries
        retrieve into
            planning relation
                where ...

Retrieval Program Generation

    (plan rules for each planning relation)

    for tuples in    var :  planning relation
        do
            step 1:    (plan rules)
            step 2:    (plan rules)
                 . . . . . .
    end for
```

Figure 8.2
The Structure of the Planning Section of RELPLAN

Example 8.2. The specification of one step of the plan generation part for the planning strategy *Small_Small.*

In the planning strategy Small_Small, the second step is to fly from a big airport iteratively to a big airport which is near the destination. The modification rules in this step are as in Figure 8.3.

The first rule *append ...* restricts the arrival port to being a big airport, which reduces the size of the data relation to be iteratively searched. The second rule *replace ...* changes the final state of the iteration (the arrival port) from the final destination port to a port near it. (The final destination port is stored as *s.arr* = ... in the small planning relation.) □

A plan module is generally much longer than a deductive module without a planning section, because it contains plan selection and plan generation operations for each possible planning strategy. But the generated iterative query program will not be as complex as one might conclude from looking at the plan module. For example, for the planning strategy *Local*, the iteration part in the plan generation part is deleted, and the query program generated is just a one-

```
step 2: /* flying from a big port to a big port which is close to the
              destination by via big ports only. */

        append constraint for iteration  - > f.arr = p1.port and p1.size > 10

        replace  final - > n.arr = p1.port and s.arr = p2.port and
              p1.lat - p2.lat < 5 and p1.lat - p2.lat > -5
              and p1.long - p2.long < 5 and p1.long - p2.long > -5
```

Figure 8.3. Plan Modification Rules in One Plan Step of Plan Generation

step simple retrieval. And, even for a complex planning strategy such as Small_Small, the generated program will be much more efficient than would be a program generated without the tailoring done by the planning section. This will be demonstrated in Section 8. 4.

Example 8.3. The structure and planning rules of the plan module *flight.*

The planning section of the plan module *flight* in RELPLAN is shown in Figure 8.4. To simplify our discussion, only the case *Small_Small* is demonstrated. The other four cases, *Local, Big_Big, Big_Small and Small_Big*, are not included in the figure. □

8.3. COMPILATION OF RECURSIVE RULES ACCORDING TO PLANNING STRATEGIES

A query program is generated by the execution of the modified deductive module. The generation is similar to what is described in Section 7.3 with the difference being that the generation of a query program using planning techniques uses modified rules, while the one without planning directly uses the rules stored in the non-plan section of the deductive module in RELPLAN.

Algorithm 8.1. Compilation of Deductive Queries Using Planning Techniques.

(1) Phase 1: The selection of planning strategies.

(i) The plan selection is performed based on the rules and queries specified in the plan selection part of the planning section;

(ii) The queries specified in the plan selection part are executed against the user's query and data stored in the database;

(iii) The execution of the queries possibly results in some non-empty planning relations. Each non-empty planning relation triggers selection of a

```
module flight
                ......
plan ->

schemaSmall_Small(dpt, arr)
                ......

retrieve into Small_Small(p1.port, p2.port)
        where . . .
                ......

for tuples in s : Small_Small do

step 1: /* First fly to a big port in one step. */
        append constraint for start -> f.arr = p1.port and p1.size > 10
        delete iteration

step 2: /* Then fly via big ports only to a big port
                which is close to the destination. */
        append constraint for iteration -> f.arr = p1.port and p1.size > 10
        replace final -> n.arr = p1.port and s.arr = p2.port
                and p1.lat - p2.lat < 5 and p1.lat - p2.lat > -5
                and p1.long - p2.long < 5 and p1.long - p2.long > -5

step 3: /* Finally fly from that port directly to the destination. */
        delete  iteration

end for

end module
```

Figure 8.4. The Planning Part of a Plan Module in RELPLAN

particular planning strategy.

(2) Phase 2: Plan generation.

A plan-based retrieval program is generated based on each plan strategy selected and the user's query. It has the following steps:

For each plan step, do [†]

(i) modify the module rules in the deductive module as instructed by the planning rules. Instructions will be to *append, delete* or *replace* the module rules originally in the deductive module with rules specified in planning section;

(ii) generate the query program based on the modified module. The query program generation is based on the algorithm (Algorithm 7.1) for the generation of query programs from deductive modules containing no planning sections. □

The algorithm is demonstrated by the following example.

Example 8.4. Apply the planning technique to generate a query program for the query *"retrieve flights from Madison to Suzhou."*

User's query :

```
range of x is flight
retrieve (x.dpttime, x.arrtime, x.fare)
where x.dpt = "Madison" and x.arr = "Suzhou"
        and x.fare < 1000 and x.arrtime - x.dpttime < 30
```

Figure 8.5. A User's Deductive Query for Planning

[†] If the planning relation is empty, the *for* statement is skipped. This serves as the test for which planning strategy has been selected.

The plan-based query program that results from RELPLAN:

```
/* Only the second phase, plan generation, is demonstrated here. */

range of x is flight
retrieve (x.dpttime, x.arrtime, x.fare)
where x.dpt = "Madison" and x.arr = "Suzhou"
        and x.fare < 1000 and x.arrtime - x.dpttime < 30

range of p1 is airport
retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr,
        dpttime, arrtime, fare)
        where n_.fno = x.fno and n_.f1no = 0  and  n_.f2no = x.fno
        and  n_.dpt = x.dpt  and  n_.arr = x.arr  and
        n_.dpttime  =  x.dpttime and  n_.arrtime = x.arrtime  and
        n_.fare = x.fare  and  x.dpt = "Madison" and  x.arr = p1.port
        and  p1.size  >  10  and  n_.fare  <  1000
         and  n_.arrtime  -  n_.dpttime  <  30

loop

range of p1 is airport
range of p2 is airport
/* Collect the new_flights whose arrival are close to the destination */
retrieve into tmp_relation and delete new_flight
        where
         n.arr  =  p1.port and  s.arr  =  p2.port
         and p1.lat  -  p2.lat  <  5 and  p1.lat  -  p2.lat  >  -5
         and p1.long  -  p2.long  < 5 and  p1.long  -  p2.long  >  -5

range of n is new_flight
range of p0 is airport
range of p3 is airport
/* Obtain new_flights by connecting the old new_flights
        with the flights which meet the constraints. */
retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr,
        dpttime, arrtime, fare)
        where
        n_.fno = n.fno & x.fno and n_.f1no = n.fno and n_.f2no = x.fno
        and n_.dpt = n.dpt and n_.arr = x.arr and n_.dpttime = n.dpttime
        and n_.arrtime = x.arrtime and n_.fare = n.fare + x.fare
        and n.arrtime + 3 > x.dpttime and n.arrtime + 1 < x.dpttime
        and p0.port = x.dpt and p_mg.port = x.arr and
        p2.port = x.dpt and p3.port = x.arr and x.arr = p1.port
        and p1.size > 10 and n_.fare < 1000 and
        n_.arrtime - n_.dpttime < 30 and
        ( p0.lat - p_mg.lat ) * ( p2.lat - p3.lat ) > 0
        and ( p0.long - p_mg.long ) * ( p2.long - p3.long ) > 0

exit when new_flight is empty

end loop

range of n is tmp_relation

/* Flying from the port which is close to the destination directly to
the final destination. */
retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr,
        dpttime, arrtime, fare)
        where
        n_.fno = n.fno & f.fno and n_.f1no = n.fno and
        n_.f2no = f.fno and n_.dpt = n.dpt and
        n_.arr = f.arr and n_.dpttime = n.dpttime
        and n_.arrtime = f.arrtime and n_.fare = n.fare + f.fare
        and n.arrtime + 3 > f.dpttime and n.arrtime + 1 < f.dpttime
        and n_.fare < 1000 and n_.arrtime - n_.dpttime < 30

retrieve ( n_.dpttime , n_.arrtime , n_.fare )
        where
        n_.dpt = "Madison" and n_.arr = "Suzhou" and
        n_.fare < 1000 and n_.arrtime - n_.dpttime < 30
```

Figure 8.6. The Generated Query Program that Results from Planning

□

The generated query program can of course be sent to a relational query optimizer for further optimization.

8.4. THE PERFORMANCE GAIN FROM PLANNING IN DATABASE ORIENTED PROBLEM SOLVING

The planning process in Example 8.4 generates a longer query program than one not based on planning. People may wonder whether the program will result in more efficient processing. Next we examine the processing efficiency measured by the number of tuples generated during the iterative searches of each program.

Suppose there are 100 k tuples in the database, each requiring 100 bytes, in the relation *flight* and 5 k tuples, each requiring 100 bytes, in the relation *port*. The total database of 10.5 megabytes cannot be processed by main memory algorithms, so database processing is a necessity. Suppose that the average cost of each local flight is $50.00 and of each flight from one big port to another is $150.00. Then the average number of flight connections (iterations) for a maximum $1000 fare is twenty, if via both big and small ports, and seven, if via big ports only. We divide the discussion into several cases:

(1) Bare iterative search without any restriction and with the user's query processed at the end.

The process will never terminate because without restriction iteration will generate an infinitely large number of tuples in the generic relation *new_flight*.

(2) Iterative search with user's bound information used during iteration.

With bound information (e.g., maximum fare $1000) used, the iteration will terminate. Suppose the average join selectivity is 10^{-3}, and the initial start state has 10 tuples selected. The second iteration will generate $10 * 10^{-3} * 10^5 = 1000$ tuples. The total number of tuples processed in 20 times would be : $10 + 1000 + \ldots + 10 * 100^{19} = 10 * (100^{20} - 1)/(100 - 1) \simeq 10^{39}$, a number too huge to be processed in a reasonable amount of computing time.

(3) Iterative search with (2) and the *same-direction* constraint used.

With the *same-direction* constraint used, the join selectivity will be improved by about four times, and the total number of tuples processed would be: $10 + 10 * 25 + \ldots + 10 * 25^{19} \simeq 10^{27}$, a significant reduction but still too huge to be processed.

(4) Iterative search with (3) and the *lay-over* time constraint used.

With the *lay-over* time constraint used, assume that the selectivity will be increased by about a factor of 10, so the total number of tuples processed would be: $2.5 + 2.5^2 + \ldots + 2.5^{20} \simeq 1.6*10^8$, another significant reduction, which may or may not require reasonable processing power.

(5) Iterative search with (4) and planning technique used:

With our planning technique used, the search on small ports is limited to the end of the search. The average number of iteration will be significantly reduced. Suppose the number of iterations is reduced to eight in our example. The total number of tuples processed would then be: $2.5 + 2.5^2 + \ldots + 2.5^8 \simeq 2500$. The planning technique contributes significantly because it reduces the average number of iterations from 20 to 8. Note here we have just considered the effect of fewer iterations. If we were also to consider the reduction in the size of the relations to be iteratively joined at each iteration (restricted to big ports only), we obtain an even better cost reduction.

Method	Search Strategy	Tuples Generated	Comparison
1	Bare iteration	Infinity	Infinity
2	Use of bounds	10^{39}	$4*10^{34}$
3	2 + one-direction	10^{27}	$4*10^{22}$
4	3 + lay-over time	$1.6*10^{8}$	64000
5	4 + planning	25000	1

Table. 8.2. Search Efficiency Using Knowledge and Planning

Taking the processing cost for the case using planning to be unity, we compare costs in Table 8.2. We use the number of tuples generated as a rough cost measurement. The table shows that the more knowledge is applied, the more focused and efficient the search is. Planning clearly can play an important role in achieving efficient search in recursive database problems.

Although our coarse estimation only estimates order of magnitude differences in the number of tuples processed, it is reasonable to expect that more detailed simulation and performance testing would give comparable results in favor of knowledge-based constraints and planning.

8.5. SUMMARY: PLANNING IN DATABASE ORIENTED PROBLEM SOLVING

The planning techniques we have studied can be summarized as follows:

(1) Planning is hierarchical space shaping with expert knowledge. It is to be distinguished from search reduction with constraints and with heuristic rules.

(2) Planning should enable flexible use of different planning strategies for different queries and different kinds of data. A planning process should have two-phases: **the selection of a planning strategy** and **the generation of a plan-based retrieval program according to the selected strategy**.

(3) The strategy selection phase uses built-in queries to decide an appropriate strategy. The selection is based on information from the user query and retrieved from the database.

(4) The retrieval program generation phase is performed by modification of rules in a deductive module, according to selected planning strategy, using the user query and the the modified rules to generate a plan-based retrieval program.

(5) A plan module in RELPLAN implements this two-phase planning technique at the same time preserving a high-level query interface. The planning mechanism is transparent to database users.

(6) By adopting different search constraints and heuristics for different hierarchy levels, planning may considerably reduce search cost in database oriented problem solving.

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH ISSUES

9.1. CONCLUSIONS

In this thesis, we have studied two related approaches for compiling and processing relational queries involving recursive rules: **pattern-based recursive rule compilation** and **knowledge-directed recursive rule compilation and planning.**

9.1.1. Pattern-Based Recursive Rule Compilation

Recursive rules are transformed and classified into several classes according to their recursion patterns. We have studied three kinds of compilation and processing algorithms for different classes of recursive rules: **transitive closure algorithms, SLSR wavefront algorithms,** and **stack-directed compilation algorithms.**

For processing rule sets in the transitive closure class, we studied several algorithms and found the δ **Wavefront algorithm** to be preferable. For rule sets in the SLSR (Single Looping rule with a Single Recursion point) class, we compared four different evaluation algorithms using analytical models and performance tests, and we concluded that the **Single Wavefront algorithm** performs the best in most cases. **Stack-directed compilation algorithms** were introduced for recursive rule sets with more complex forms of recursion, and we studied an SLMR rule set with acyclic databases. For shared processing, we proposed the use of two kinds of wavefront relations: **wavefront** and **potential wavefront**. **Performing selection first** and **making use of previous processing results (wavefronts)** are two important heuristics to reduce query processing cost, and they are common heuristics for all these algorithms.

By comparing different kinds of recursive rules and their respective compilation techniques, we have observed an interesting evolution of the compilation techniques. The more complex the recursion, the less "pure" the compilation. That is, for more complex recursion, expansion and termination of "compiled" formulas cannot be completely isolated from the deductive resolution and the entire process takes on aspects of interpretation. The ordered generation of compiled formulas is a depth-first process while the processing of compiled formulas using set-oriented relational operations can be a breadth-first process. The deductive compilation and processing of recursive queries in relational databases is thus essentially an integration of breadth-first and depth-first search paradigms.

9.1.2. Knowledge-Directed Recursive Rule Compilation and Planning

For solving database-oriented application problems, we have studied the application of AI techniques in compilation and evaluation processes, and we introduced **knowledge-directed recursive rule compilation and planning.** Our approach incorporates functional definitions, domain-specific knowledge, query constants, and a planning technique into the compilation of recursive queries.

Knowledge-directed compilation incorporates termination constraints, search constraints, and query constants into the compilation of recursive rules. **Planning** partitions the search space hierarchically and enables utilization of different constraints for different levels. We proposed a two-phase planning technique: **selection of an appropriate planning strategy** and **generation of a database accessing plan according to the selected strategy.**

We have designed and implemented a knowledge-directed relational planner, RELPLAN, to experiment with knowledge-directed compilation and planning while maintaining a high level query interface. We have shown that knowledge-directed recursive rule compilation and planning can result in significant performance improvements.

The two approaches, **pattern-based recursive rule compilation** and **knowledge-directed compilation and planning**, are complementary in many ways. The former is the basic approach for compiling recursive rules, while the latter adds ideas from AI research for reducing the search space in processing. The former emphasizes the techniques of deductive query compilation and relational query processing, while the latter applies AI search and planning techniques. Integration of the two approaches reflects the contemporary confluence of research in AI, logic, and database systems.

9.2. LIMITATIONS OF OUR RESEARCH

The algorithms we studied need to be implemented and validated in real systems. More efficient algorithms need to be designed and researched in depth. We have observed several limitations of our research results. Some are explicitly presented here, and others are implied in our discussion of future research issues.

(1) Customization of algorithms for efficient processing.

Our stack-directed compilation algorithms, though general, may involve costly processing. More customized algorithms and processing techniques for specialized recursion patterns should be explored in depth.

(2) The procedural flavor of RELPLAN's deductive and plan modules.

RELPLAN's deductive and plan modules, though preserving a high-level user view and query interface, have a procedural flavor. In RELPLAN, a looping rule is specified in iterative form rather than recursive form, and other module rules are represented in rigid, stereotyped forms. Such specification discourages more flexible control of the compilation process. For example, the *start rule* dictates the iteration start point and excludes any other start points (such as backward search).

(3) Automatic construction of planning strategies.

In RELPLAN, planning strategies are constructed by human experts. The proposed system certainly cannot automatically construct a planning strategy. More intelligent systems should be capable of discovering planning rules from analysis, from experience (learning), or by deduction from other rules. This is a most interesting topic for AI research.

(4) Knowledge-directed compilation and planning for complex recursion.

Our research on knowledge-directed compilation and planning focused on transitive closure recursion. For compiling more complex recursion, the knowledge-directed compilation and planning techniques we have investigated will almost certainly have to be significantly extended.

9.3. PROPOSALS FOR FUTURE RESEARCH

We propose for future research: (1) development of a comprehensive recursive query compiler, (2) research on breadth-first flavored heuristic search, and (3) development of a plan database.

9.3.1. Development of a Comprehensive Recursive Query Compiler

An immediate software development project, which applies the theories and algorithms studied in this dissertation, is to develop a comprehensive knowledge-directed recursive rule compiler for relational database systems.

Such a compiler would function as a deductive front-end similar to our relational planner RELPLAN, but it would be different from RELPLAN in two aspects: (1) it would deal with rules in declarative recursive forms instead of procedural iterative form, which would radically change the means of user control; and (2) it would deal with more complex recursive rules, in particular, those requiring SLSR wavefront algorithms and stack-directed compilation algorithms.

The proposed compiler would contain the following components: (1) recursive rule transformation routines (based on the discussion in Chapter 3), (2) knowledge use and planning (based on Chapters 7 and 8), and (3) pattern-based recursive rule compilation (based on Chapters 4 through 6).

The construction of such a knowledge-directed recursive query compiler is not just a software development project. It would involve some interesting research issues, e.g., those associated with the development of a comprehensive high-level recursive query interface.

9.3.2. Breadth-First Flavored Heuristic Search

The proposal for studying breadth-first flavored heuristic search is based on the following observations:

(1) Different search philosophies for expert systems and database systems.

Different search philosophies are adopted by expert systems and by database systems. The former usually adopt depth-first flavored heuristic search and follow along one search path (likely a promising one according to some heuristic information or expert judgement), and then they backtrack if the path ends in failure. The latter usually apply breadth-first flavored search which explores all possible solutions using set-oriented relational operations. The search in an EDS should somehow combine the best features of these two approaches.

(2) The merit of set-oriented search.

In finding multiple solutions, set-oriented search generally costs less than tuple-oriented search, especially for searching large databases. The gain from set-oriented search suggests that the search method in EDS should explore searching with a certain width of paths rather than one path at a time. Backtracking based on single-path search often costs too much, among other reasons, because there is so much redundant processing across different paths.

(3) The merit of heuristic search.

Heuristic search has been the key to successful AI problem solving, because (1) for many AI problems searching, all paths is prohibitively expensive; (2) most AI problems require finding only a small number of good solutions (which need to satisfy certain criteria but need not be the best possible); and (3) quite useful heuristic information is available in many cases (e.g. information that can provide rough but usable cost evaluation functions). Heuristic search may be even more important in an EDS, because it may involve even larger search spaces.

Finding some but interesting answers, rather than all answers, in EDS applications requires the exploration of set-oriented heuristic search which we called a **breadth-first flavored heuristic search**. It should have the following characteristics:

(1) It should incorporate relational database techniques such as data storage structures, query optimization techniques and set-oriented relational operations;

(2) It should study heuristics for dynamically cutting off unpromising paths. Only initial paths that satisfy search constraints, heuristic criteria and planning rules should be preserved. Other paths should either be tossed (depending on the danger and cost associated with missing a successful path) or stacked for some modified kind of backtracking.

(3) The key problems concern balance and control of search width and depth, finding good heuristic functions that apply to database search, and control of backtracking.

9.3.3. Development of a Plan Database

Compared to human planning, the planning of RELPLAN is still a rigid process. Experience tells us that for many situations there are often several plans available, and the best plan is often identified on the basis of experience. This observation encourages us to develop a plan database, which is a special database relation. This relation would be generated by planning processes and enriched by executions of generated plans. It would register a variety of information important for planning, such as problem domain characteristics, database status, available plans, query execution costs of the plans, etc. Performance information would be accumulated from the history of executing plans selected out of the database.

During the plan selection phase, the available plans stored in the database which meet database characteristics and query requirements would be retrieved, and their performance history would be compared. The best performer for the current kind of situation would be selected. Feedback from execution of the generated plan would be registered. Thus, a plan database is essentially a database for storing available plans and their performance histories. Such a database, if it could be adequately implemented and studied, would make experience accumulation (learning) and database retrieval (remembering) central to successful planning and scheduling. Expert human planners and schedulers clearly make use of something comparable.

REFERENCES

[AhoU 79] A. Aho and J. D. Ullman, "Universality of Data Retrieval Languages", *Proceedings of the 6th ACM Symposium on Programming Languages*, San Antonio, Texas, Jan. 1979.

[Baro 81] A. Baroody and D. DeWitt, "An Object-Oriented Approach to Database System Implementation", *ACM Transactions on Database Systems 6(4)*, Dec. 1981.

[Barr 81] A. Barr and E. Feigenbaum, *The Handbook of Artificial Intelligence (Vol. I & II)*, Heuristic Press & William Kaufmann Inc., 1981.

[Bitt 83] D. Bitton, D. DeWitt and C. Turbyfill "Benchmarking Database Systems : A Systematic Approach ", *Proceedings of the 9th International Conference on Very Large Databases*, Oct. 1983.

[Blas 77] M. Blasgen and K. Eswaran "Storage and Access in Relational Data Base", *IBM System Journals*, No. 4, pp 363-377, 1977.

[Brod 80] M. Brodie and S. Zilles (Eds.), *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling*, Pingree Park, Colo., June 1980.

[Brod 84] M. Brodie, J. Mylopoulos and J. Schmidt (Eds.), *On Conceptual Modeling*, Springer-Verlag, 1984.

[Brod 84b] M. Brodie and M. Jarke, "On Integrating Logic Programming and Databases", *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984.

[Care 85] M. Carey, and D. DeWitt, "Extensible Database Systems", *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, Florida, Feb. 1985.

[Chak 84] U. Chakravarthy, D. Fishman and J. Minker, "Semantic Query Optimization in Expert Systems and Database Systems", *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984.

[Cham 75] D. Chamberlin, J. Gray and I. Traiger, "Views, Authorization and Locking in a Relational Data Base System", *Proceedings of the NCC*, May 1975.

[Chan 73] C. Chang and R. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[Chan 78] C. Chang, "DEDUCE 2: Further Investigations of Deduction in Relational Databases", In [Gall 78].

[Chan 81] C. Chang, "On the Evaluation of Queries Containing Derived Relations in a Relational Data Base", In [Gall 81].

[Chou 83] H. Chou, D. DeWitt, R. Katz and A. Klug, "Design and Implementation of the Wisconsin Storage System", *Computer Science Technical Report No. 524*, University of Wisconsin-Madison, Nov. 1983

[Clar 82] K. Clark and S.A. Tarnlund (Eds.), *Logic Programming*, Academic Press, 1982.

[Cloc 81] W. Clocksin and C. Mellish, *Programming in PROLOG*, Springer-Verlag, 1981.

[Codd 72] E. F. Codd, "Relational Completeness of Database Sublanguages", In *Data Base Systems* R. Rustin, (Ed.), Prentice-Hall, New York, 1972.

[Codd 80] E. F. Codd, "Extending the Relational Database Model to Capture More Meaning", *ACM Transactions on Database Systems 4(4)*, 1980.

[Cohe 83] P. Cohen and E.A. Feigenbaum, *The Handbook of Artificial Intelligence (Vol. III)*, Heuristic Press & William Kaufmann Inc., 1983.

[Cope 84] G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, Mass., 1984.

[Dahl 82] V. Dahl, "On Database Systems Development Through Logic", *ACM Transactions on Database Systems 7(1)*, 1982.

[Date 81] C. Date, *An Introduction to Database Systems, Vol. 2.*, Addison-Wesley, Reading, Mass., 1981.

[Davi 80] R. Davis, "Meta-Rules : Reasoning about Control", *Artificial Intelligence 15(3)*, 1980.

[Daya 85] U. Dayal and J. Smith, "PROBE: A Knowledge-Oriented Database Management Systems", *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, Florida, Feb. 1985.

[Forg 79] C. Forgy, "On the Efficient Implementation of Production Systems", *Ph.D. Thesis*, Carnegie-Mellon University, 1979.

[Gall 78] H. Gallaire and J. Minker, (Eds.), *Logic and Databases*, Plenum Press, 1978.

[Gall 81] H. Gallaire, J. Minker and J. Nicolas, (Eds.), *Advances in Data Base Theory, Vol. 1,* Plenum Press, 1981.

[Gall 84] H. Gallaire, J. Minker and J. Nicolas, "Logic and Databases : A Deductive Approach", *Computing Survey* 16(2), 1984.

[Gall 84b] H. Gallaire, J. Minker and J. Nicolas, (Eds.), *Advances in Data Base Theory, Vol. 2,* Plenum Press, 1984.

[Gold 83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Gran 81] J. Grant and J. Minker, "Optimization in Deductive and Conventional Relational Database Systems", In [Gall 81].

[Gutt 84] A. Guttman, "New Features for Relational Database Systems to

Support CAD Applications", *Ph.D. Thesis*, University of California-Berkeley, June 1984.

[Haer 78] T. Haerder, "Implementing a Generalized Access Path Structure for a Relational Database System", *ACM Transactions on Database Systems 3(3)*, 1978.

[Han 84] J. Han, "Planning in Expert Database Systems by Using Rules", *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984.

[Han 85a] J. Han and L. Travis, "Knowledge-Directed Recursive Rule Compilation in Expert Database Systems", *Proceedings of the 2nd Conference on Artificial Intelligence Applications*, Miami, Flori., Dec. 1985.

[Han 85b] J. Han and Z. Li, "DQL -- The Deductive Augmentation of the Relational Database Query Language QUEL", *Proceedings of the 2nd International Congress on Logica, Informatica, Diritto*, Florence, Italy, Sept. 1985.

[Han 85c] J. Han and L. Travis, "Using Expert Knowledge in Database-Oriented Problem Solving", *Proceedings of the 6th International Conference on Information Systems*, Indianapolis, Ind., Dec. 1985.

[Han 85d] J. Han and H. Lu, "Some Performance Results on Recursive Query Processing in Relational Database Systems", to appear in *Proceedings of the 2nd International Conference on Data Engineering*", IEEE Computer Society, 1985.

[Han 85e] J. Han and L. Travis, "Pattern-Based Compilation of Recursive Rules in Expert Database Systems", submitted for publication.

[Haye 83] F. Hayes-Roth, D. Waterman and D. Lenat (Eds.), *Building Expert Systems*, Addison Wesley, 1983.

[Hens 84] L. Henschen and S. Naqvi, "On Compiling Queries in Recursive

First-Order Databases", *JACM 31(1)*, 1984.

[Hens 84b] L. Henschen, W. McCune and S. Naqvi, "Compiling Constraint Checking Programs from First Order Formulas", in [Gall 84b].

[Ioan 85] Y. Ioannidis, "A Time Bound on the Materialization of Some Recursively Defined Views", *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, Aug. 1985.

[Jark 84] M. Jarke, J. Clifford and Y. Vassiliou, "An Optimizing PROLOG Front-End to a Relational Query System", *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, Mass., May 1984.

[Jark 84b] M. Jarke and J. Koch, "Query Optimization in Database Systems", *Computing Survey* 16(2), 1984.

[Kell 78] C. Kellogg, P. Klhar and L. Travis, "Deductive Planning and Path Finding for Relational Data Bases", In [Gall 78].

[Kell 81] C. Kellogg and L. Travis, "Reasoning with Data in a Deductively Augmented Data Management System", In [Gall 81].

[Kern 78] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[King 81] J. King, "QUIST : A System for Semantic Query Optimization in Relational Databases", *Proceedings of the 7th International Conference on Very Large Data Bases*, Cannes, France, 1981.

[Kowa 79] R. Kowalski, *Logic for Problem Solving,* American Elsevier, 1979.

[Kowa 81] R. Kowalski, "Logic as a Database Language", *Proceedings of the Advanced Seminar on Theoretical Issues in Data Bases*, Certraro, Italy, Sept. 1981.

[Kung 84] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro and M. Stonebraker, "Heuristic Search in Data Base Systems", *Proceedings* *of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984.

[Love 78] D. Loveland, *Automated Theorem Proving : A Logical Basis*, Elsevier North-Holland, 1978.

[Lu 85] H. Lu and M. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network", *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, Aug. 1985.

[McKa 81] D. McKay and S. Shapiro, "Using Active Connection Graphs for Reasoning with Recursive Rules", *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Vancouver, Canada, 1981.

[Mink 78] J. Minker, "Search Strategy and Selection Function for an Inferential Relational Systems", *ACM Transactions on Database Systems 3(1)*, 1978.

[Mink 81] J. Minker and J. Nicolas, "On Recursive Axioms in Relational Databases", *Tech. Rep. No. 1119*, University of Maryland, College Park, MD, 1981.

[Naqv 83] S. Naqvi and L. Henschen, "Synthesizing Least Fixed Point Queries into Non-Recursive Iterative Programs", *Proceedings of the 1983 International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, Aug. 1983.

[Newe 72] A. Newell and H. Simon, *Human Problem Solving*, Prentice-Hall, 1972.

[Nils 80] N. Nilsson, *Principles of Artificial Intelligence*, Tioga, 1980.

[Pear 84] J. Pearl, *Heuristics : Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[Reit 78] R. Reiter, "Deductive Question-Answering on Relational Databases", In [Gall 78].

[Reit 84] R. Reiter, "Towards a Logical Reconstruction of Relational Database Theory", In [Brod 84].

[Robi 65] J. Robinson, "A Machine Oriented Logic Based on the Resolution Principle", *JACM* 12(1), 1965.

[Rowe 79] L. Rowe and K. Schoens, "Data Abstraction, Views, and Updates in RIGEL", *Proceedings of the 1979 ACM-SIGMOD International Conference on Management of Data*, Boston, Mass., May 1979.

[Sace 77] E. Sacerdoti, *A Structure for Plans and Behavior,* American Elsevier, 1977.

[Seli 79] P. Selinger, D. Astrahan, D. Chamberlin, R. Lorie and T. Price, "Access Path Selection in a Relational Database System", *Proceedings of the 1979 ACM-SIGMOD International Conference on Management of Data*, Boston, Mass., May 1979.

[Sell 85] T. Sellis and L. Shapiro, "Optimization of Extended Database Query Languages", *Proceedings of the 1985 ACM-SIGMOD Conference on Management of Data*, Austin, Texas, May 1985.

[Seve 76] D. Severance and G. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", *ACM Transactions on Database Systems 1(3)*, 1976.

[Shap 80] S. Shapiro and D. McKay, "Inference with Recursive Rules", *Proceedings of the 1st Annual National Conference on Artificial Intelligence*, Palo Alto, Calif., 1980.

[Simo 83] H. Simon, "Search and Reasoning in Problem Solving", *Artificial Intelligence 21(2)*, 1983.

[Smit 84] J. Smith, "Expert Database Systems : A Database Perspective ", *Proceedings of the 1st International Workshop on Expert Database*

Systems, Kiawah Island, South Carolina, Oct. 1984.

[Ston 75] M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification", *Proceedings of the 1975 ACM-SIGMOD Conference on Management of Data*, 1975.

[Ston 76] M. Stonebraker, E. Wong and P. Kreps, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems 1(3)*, 1976.

[Ston 84] M. Stonebraker, "QUEL as a Data Type", *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, Mass. 1984.

[Tars 55] A. Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications", *Pacific Journal of Mathematics*, 5(2), June 1955.

[Ullm 82] J. D. Ullman, *Principles of Database Systems*, 2nd ed. Computer Science Press, 1982.

[Ullm 85] J. D. Ullman, "Implementation of Logical Query Languages for Databases", *ACM Transactions on Database Systems 10(3)*, 1985.

[Vald 85] P. Valduriez, "Join Indices", *MCC Technical Report No. DB-052-85*, July 1985.

[Vald 85b] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices", *MCC Technical Report No. DB-069-85*, 1985.

[Warr 81] D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic", *Proceedings of the 7th International Conference on Very Large Data Bases*, Cannes, France, 1981.

[Warr 84] D. Warren and E. Sciore, "Towards an Integrated Database-PROLOG System", *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984.

[Wate 78] D. Waterman and F. Hayes-Roth (Eds.), *Pattern Directed Inference*

Systems, Academic Press, 1978.

[Wong 84] E. Wong, Y. Ioannidis and L. Shinkle, "Enhancing INGRES with Deductive Power", *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984.

[Wood 68] W. Woods, "Procedural Semantics for a Question-Answering Machine", *Fall Joint Computer Conference*, 1968.

[Zani 84] C. Zaniolo, "Prolog : A Database Query Language For All Seasons", *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984.

[Zani 85] C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects", *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, 1985.

APPENDIX I

SYNTACTIC SPECIFICATION OF RELPLAN

The syntactic specification of RELPLAN adopts the extended BNF, where { ... } denotes a set of zero or more occurrences, [...] denotes one or zero occurrences, and (.. | ..) denotes one of several occurrences.

```
<RELPLAN >            : <Data_Definition> <Data_Manipulation>
<Data_Definition>     : { <Data_Def1> }  { <Module_Definition> }
<Data_Def1>           : <Schema_Definition>
                      | <Var_Declaration>
                      | <Rule_Definition>
<Schema_Definition> : schema { Rel_Name '(' Attr_Name
                        {',' Attr_Name } ')' }
<Var_Declaration>     : range of Var_Name  {',' Var_Name }
                        is [module] Rel_Name
<Rule_Definition>     : <Virtual_Rel_Defn>
                      | <Constraint_Defn>
<Virtual_Rel_Defn>    : define virtual relation [ Var_Name ':' ] Rel_Name
                        '(' <Attr_Reference> {',' <Attr_Reference> } ')'
                        [ where <Qualification> ]
<Constraint_Defn>     : define constraint [ Var_Name ':' ] Constraint_Name
                        '(' <Attr_Reference> {',' <Attr_Reference> } ')'
                        [ where <Qualification>]
```

```
<Attr_Reference>       : Attr_Name
                       | Attr_Name  '='  <Expression>
<Module_Definition>    : module Module_Name <Module Body> end module
<Module Body>          : { <Data_Def1> } { < Mod_Rule_Defn >}
                           [<Plan_Definition> ]
<Mod_Rule_Defn>        : <Step> '=>' <Num_Query> { <Num_Query> }
                       | constraint [for <For_Step> ] '=>'
                         <Num_Clause> { <Num_Clause> }
                       | (upper | lower) bound '=>' <Num_Attribute>
<For_Step>             : start | iteration | final
<Num_Attribute>        :[ '(' Integer ')' ] <Attribute>
<Num_Query>            : [ '(' Integer ')' ] <Query>
<Num_Clause>           : [ '(' Integer ')' ] <Clause>
<Attribute>            : Var_Name '.' Attr_Name
<Plan_Definition>      : plan '=>' <Plan Prelude> <Plan_Steps>
<Plan Prelude>         : <Data_Def1> <Data_Manipulation>
<Plan_Steps>           : for tuples in Var_Name ':' Rel_Name do
                           <Step> { <Step> } end for
<Step>                 : step Integer ':' <Modification> { <Modification> }
<Modification>         : (append | replace) <Stereo_Typed_Rule_Defn>
                       | delete ( <Step> constraint [for <Step> ]
                         (upper | lower) bound )
<Data_Manipulation>    : {  { <Var_Declaration> } <Query> }
<Query>                : retrieve <Target List1> where <Qualification>
                       | retrieve into Rel_Name <Target List2>
                         where <Qualification>
<Target List1>         : '(' <Attribute> {',' <Attribute> } ')'
```

```
<Target List2>    :   '(' <Expression> {',' <Expression> } ')'
<Qualification>   :   '(' <Qualification> ')'
                  |   not <Qualification>
                  |   <Qualification> (and | or) <Qualification>
                  |   <Clause>
<Clause>          :   <Expression> <Relop> <Expression>
                  |   Constraint_Name '(' <Attribute> {',' <Attribute> } ')'
<Relop>           :   = | != | < | <= | > | >=
<Expression>      :   <Term> ( + | - ) <Term>
<Term>            :   <Factor> ( * | / ) <Factor>
<Factor>          :   <Attribute> | Constant | String
```