SQUALID: A Deductive DBMS

Nigel Stanger

A thesis submitted for the
degree of Master of Science
at the University of Otago,
Dunedin, New Zealand.

August 1991

To my mother,
for being there.

Abstract

Most modern companies probably could not function without a database management system (DBMS). Although current DBMSs are becoming increasingly sophisticated, they are still deficient in several areas. One of these areas is deduction.

Human beings have the interesting ability to derive facts from a set of data, even though these facts are not explicitly represented in the data. That is, given appropriate information, humans can deduce new information by applying rules. A *deductive database* is a database which can perform similar deductions on the data stored within it. This has the advantage that some data can be stored implicitly using rules, rather than explicitly. This reduces the amount of storage the database occupies. The use of rules also allows us to store new kinds of data, such as recursive data or indefinite data.

Several deductive database systems have been developed, but many of them only approach the problem from a theoretical point of view; practical considerations such as efficiency and ease of use for end-users have often been neglected. Many systems were also developed completely from scratch, rather than taking advantage of existing facilities. That is, it should be possible to take an existing DBMS, and extend it with deductive capabilities.

In this work we introduce the concept of a deductive database, and some of the problems associated with implementing such a system. We then discuss the implementation of a system called SQUALID (for Structured Query And Logical Inference Database). This system is based on an existing DBMS, Rdb/VMS, which has been extended with facilities for creating and manipulating rules. An extended version of the standard query language SQL is used. All rules and data are stored in a conventional database, allowing SQUALID to take advantage of the efficiency of the underlying DBMS.

Acknowledgments

I would like to thank the following people, who may or may not be in some semblance of order:

- Ian McDonald for his guidance and support;
- Christopher Robertson for many entertaining discussions, and for bestowing such an uplifting name upon my work (although he didn't realise it at the time);
- Ashe and Tom for being, well, Ashe and Tom;
- Kevin, John, Paul and the other guys, who kept me sane (more or less);
- my various flatmates since 1988, for putting up with me, my computer, my stereo, my bicycle, my cooking and my weird working hours;
- the staff of Computer Science;
- the staff and students of Information Science, and especially Prof. Philip Sallis, who gave me a job so I could finish my thesis in the black;
- to everyone else who had ideas and suggestions: thanks!
- and last, but most definitely *not* least, my family, and in particular my mother, for supporting me throughout the course of this epic work. It's been a long and eventful three and a half years, but you were always there.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Opening Comments

A database management system (DBMS) is an amazingly useful piece of software. In fact, the huge amounts of data that need to be stored and manipulated means that most modern companies could not even function without a DBMS of some kind. Unfortunately, these increasingly huge amounts of data require increasingly huge amounts of space to store them. It would be useful if we could reduce the amount of space needed to store this information.

An interesting characteristic of humans is their ability to examine a collection of data and derive from them relationships that may not be obvious at first glance. For example, given a list of people and their parents, we could deduce their grandparents, great-grandparents, and so on, despite the fact that these facts were not explicitly represented in the original data. What we have done is to use a *rule*: "the parent of a parent is a grandparent (and so on)." The use of such rules allows us to store data *implicitly*. In a conventional database, all data must be stored explicitly; it cannot apply rules to the data as humans do.

A *deductive database* is a database which can use rules in a manner similar to humans, that is, to derive new data which are implicitly represented in the existing data. The immediate advantage of this is that it can potentially reduce the space needed to store data. There are also other, less obvious, advantages:

- By using rules, we can store new kinds of data which cannot be stored using conventional database methods, for example, recursively defined data (i.e. defined in terms of itself), or indefinite data (e.g. we know that an object can have one of several properties, but we are not sure which particular one it has).

- Deductive databases are based on the principles of mathematical logic, as is the field of logic programming. In fact, both fields grew out of the same original research. If we so wish, we can increase the scope of a deductive database so that it incorporates the features and power of a logic programming language (e.g. Prolog).

The idea of using rules to derive information is not new: automated deduction was a major area of research in the 1960's, and developments in that field have been used in logic programming, deductive databases and EXPERT SYSTEMS [Minker 1988]. Expert systems work in a similar way to deductive databases, using rules to deduce information. They are not exactly the same, however—an expert system can be thought of as being *rule-oriented*, whereas a deductive database is *data-oriented*. In other words, a deductive database is oriented towards storing and manipulating large amounts of data, whereas an expert system stores and manipulates large numbers of rules.

1.2 Research Aims

Much of the research into deductive databases to date has been into developing a theoretical base for the field. Although there have been several systems implemented, many of these are small experimental systems. This is fine for demonstrating principles, but often is not much use when it comes to handling anything but a small amount of data. Also, many of the prototype systems have been more interested in demonstrating their deductive ability, while neglecting such practical aspects as efficiency and ease of use. Fortunately, this criticism has become less relevant in the last two or three years, and some promising systems have been or are currently being implemented.

Of particular interest is that many recent deductive database systems have been built as extensions of existing database systems. These systems combine the deductive power of logic with the efficiency and manipulative power of a commercial DBMS. There are no commercially available deductive DBMS's yet, but this is likely to be the direction that they will come from.

The main aim of this research was to implement such a system. The major difference between this system and most other similar systems is that it is oriented towards end-users. Many systems are based on a logic programming language such as Prolog, and merely use the database as a back-end server. This approach has the problem that users must learn a logic programming language to use the system effectively. The system that has been implemented is based on the principle that the deductive process should be as transparent to users as possible, rather than being explicitly forced upon them.

The system is called SQUALID (for *Structured QUery and Logical Inference Database*). It is based on Rdb/VMS, a relational database product developed by Digital Equipment Corporation. The system itself consists of extensions to the standard Rdb SQL interface to support the creation, use and manipulation of rules.

1.3 Structure of this Thesis

This thesis is both a summary of deductive databases in general (including recent research), and a report on the development of SQUALID. There are six chapters including this chapter, and four appendices:

- Chapter 2 introduces deductive databases and the theory behind them in more detail. First, it defines what a deductive database is, and shows how a database may be defined using logic. Second, it discusses some of the problems associated with deductive databases and some possible solutions. Third, it reviews the history of deductive database research, surveys some of the more recent research interests, and looks briefly at some deductive database implementations.

- Chapter 3 discusses issues and ideas that had to be considered while building SQUALID. First, it covers two alternate representations for storing rules. Second, it discusses an algorithm central to the working of SQUALID. Finally, it covers the SQUALID query language, SQLD (for **SQL D**eductive).

- Chapter 4 introduces an abstract model (developed by Bell *et al.* [1990]) for what are known as *integrated* deductive databases. This is relevant, as SQUALID is an integrated deductive database.

- Chapter 5 gives an overview of the implementation of SQUALID. It discusses such technical details as data structures, deduction mechanisms and how they were implemented, and so on.

- Chapter 6 is the final chapter and contains a summary and conclusions.

- Appendix A contains examples of some of the methods discussed in Chapter 2.

- Appendix B contains a brief overview of SQL for those who have not encountered this language before.

- Appendix C describes the main test data that was used with SQUALID. It also shows some examples of the use of SQUALID.

- Appendix D contains the source code listings for SQUALID.

Items shown in SMALL CAPITALS are defined in the glossary. All abbreviations used in this document are also defined in the glossary.

Chapter 2

Background

2.1 Introduction

This chapter introduces the concept of a deductive database more formally, gives a brief coverage of the history of the field and discusses some problems which are encountered when working with deductive databases.

The structure of the chapter is as follows:

- Section 2.2 introduces the concept of deductive databases. It introduces some of the basic concepts and terms involved.
- Section 2.3 discusses the basis of deductive databases: first-order logic. It defines terminology which will be used later.
- Section 2.4 discusses how a relational database can be represented using first-order logic.
- Section 2.5 covers some important issues which must be considered when implementing a deductive database system.
- Section 2.6 lists some problems that are encountered with deductive databases, and some possible solutions for them.
- Section 2.7 is a history of research into deductive databases.
- Section 2.8 discusses some possible future directions for the field.
- Section 2.9 summarises the chapter.

2.2 What is a Deductive Database?

In very general terms, a conventional database consists of a collection of facts. Using some form of query language, these facts may be accessed and manipulated as required. However, facts must be *explicitly* stored in the database for them to be of any use. If a fact is not explicitly represented in the database, then as far as the database is concerned, that fact effectively does not exist. A *deductive database* contains not only facts, but also general rules. These rules can be used to deduce new facts that are not explicitly represented in the database; that is, data can be stored *implicitly*.

To clarify this idea of implicit data representation, consider the following example. Suppose we have a conventional database that contains the two facts:

"John is Bill's father."
"Bill is Derek's father."

Now most people will quickly realise that John is also Derek's grandfather (assuming, of course, that the two Bills are the same person). In other words, this fact is implicit in the two existing facts. However, the fact "John is Derek's grandfather" is not stored in the database, so as far as the database is concerned, there is no relationship whatsoever between John and Derek.

Now consider the following deductive database:

"John is Bill's father."
"Bill is Derek's father."
"IF *x* is *y*'s father AND *y* is *z*'s father, THEN *x* is *z*'s grandfather."

In addition to the two facts, this database also contains a rule that specifies the *grandfather* relationship between a pair of entities. The database can prove the statement "John is Derek's grandfather" by making the following substitution into the rule: *x* = "John", *y* = "Bill" and *z* = "Derek".

The effect of rules of this form is to define new relations that are not explicitly represented in the database. These are known as *implicit* or *virtual* relations. Explicit relations, like the *father* relation above, are known as *base* relations. The set of virtual relations is called the *intensional* database (IDB), while the set of base relations is called the *extensional* database (EDB).

Deductive database systems are based on first-order logic. First-order logic allows us to express both facts and rules about the facts using the same syntax. Incidentally, this uniformity of representation is claimed as one of the main advantages of logic programming. For example, the representation of the database above in first-order logic would be:

$\rightarrow$ *father*(John, Bill)
$\rightarrow$ *father*(Bill, Derek)
$(\forall x\ \forall y\ \forall z)(\mathit{father}(x, y) \wedge \mathit{father}(y, z) \rightarrow \mathit{grandfather}(x, z))$

where $\wedge$, $\rightarrow$ and $\forall$ represent conjunction, implication and the quantifier "for all" respectively.

Much of the research into deductive databases has concentrated on representing databases in terms of logic, specifically first-order predicate calculus, for example Dahl's system [Dahl 1982] and Minker's MRPPS [Minker 1978]. The next section covers some basic aspects of first-order logic, and defines some terminology.

2.3 First-Order Logic—A Brief Introduction

This section is only meant to be a brief introduction to the basic principles of first-order logic. A more in-depth discussion of logic can be found in Mendelson [1979].

The *first-order predicate calculus* is one particular language for expressing logical statements. It consists of the following basic components:

- parentheses,
- variables (represented by $x, y, z, \ldots$), constants (represented by $a, b, c, \ldots$), functions ($f, g, h, \ldots$) and PREDICATE symbols ($P, Q, R, \ldots$).
- the logical connectors $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, and
- the quantifiers $\forall$ and $\exists$.

The symbols used above are fairly standard, but there are many variants of them. See Table 2-1 for the meanings of these symbols, and some alternate forms. Note also the two additional logical connectors, $\vdash$ and $\sqsubset$.

A *term* is either a constant, a variable or a function. Most work with logic and databases assumes function-free terms however.

Table 2-1. "Standard" logic symbols.

"Standard" Symbol	Alternate Form(s)	Meaning
$\wedge$	&	conjunction (and)
$\vee$	|	disjunction (or)
$\neg$	~, !	negation (not)
$\rightarrow$	$>, \Rightarrow$	implication
$\leftrightarrow$	$\Leftrightarrow$, iff	equivalence (if and only if)
$\forall x$	∇x, (x)	for all x
$\exists x$	Δx, $E(x)$	there exists x
$a \sqsubset b$		b is a logical consequence of a
$a \vdash_T b$		b is provable from a under theory T

An *atomic formula* is an expression of the form

$$P(t_1, \ldots, t_n)$$

where P is an n-ary predicate name and each t_i is a term. An atomic formula is said to be *ground* if its arguments are all constants. A *literal* is either an atomic formula or a negated atomic formula (i.e. preceded by $\neg$).

A *well-formed formula* (*wff*) is recursively defined as follows:

- An atomic formula is a wff.
- If w_1 and w_2 are wffs, then so are $\neg w_1$, $w_1 \vee w_2$, $w_1 \wedge w_2$, $w_1 \rightarrow w_2$ and $w_1 \leftrightarrow w_2$.

A *clause* is a formula of the form

$$\neg A_1 \vee \ldots \vee \neg A_m \vee B_1 \vee \ldots \vee B_n$$

where the A_i and B_j are atomic formulae. Clauses are usually written in this form:

$$A_1 \wedge \ldots \wedge A_m \rightarrow B_1 \vee \ldots \vee B_n.$$

The left hand side of a clause is called the *antecedent*, the right hand side the *consequent*.

A *Horn clause* is a clause of the form

$$A_1 \wedge \ldots \wedge A_m \rightarrow B.$$

Every formula in first-order logic can be reduced to a collection of clauses, which is known as the *clausal* form of the formula.

2.4 Defining Relational Databases Using Logic

In terms of logic, a database is just a collection of formulae. Nicolas and Gallaire [1978] showed that a database in this form can be viewed in two ways:

- The formulae are to be evaluated based on the semantic definition of truth. That is, queries are evaluated by assuming that all entries in the database are true.
- The formulae are theorems that are to be proved. That is, queries are evaluated by deriving data from axioms.

The first view is known as the *model-theoretic* view; the second as the *proof-theoretic* view [Reiter 1984]. The concepts underlying both of these views will be explained with the aid of the following example, taken from Chisholm *et al.* [1987]. The example concerns students taking language courses at a university. Informally, the information stored in the database can be defined as follows:

- every student of the third course learns Latin or Greek,
- nobody learns Greek and German at the same time,

- modern languages and dead languages are languages,
- Toto, Jean and Paul are students,
- Jean and Toto are students of the third course,
- English and German are modern languages,
- Latin and Greek are dead languages,
- Paul learns English and German, and
- Toto does not learn English.

The relations that describe this data are shown in Table 2-2. The constants (or *domain elements*) of the database are:

{english, german, latin, greek, paul, toto, jean}.

Table 2-2. Relations in the example database.

learns(*x*, *y*)	*x* learns *y*,
stud(*x*)	*x* is a student,
third(*x*)	*x* is a student of the third course,
lang(*x*)	*x* is a language,
mod(*x*)	*x* is a modern language,
dead(*x*)	*x* is a dead language.

2.4.1 Semantics—The Model-Theoretic View

The model-theoretic view deals with the semantics of logical formulae. In logic, "semantics" is defined as the specification of truth values to wffs; that is, each formula is assigned a truth value.

The first concept that needs to be introduced is the *Herbrand base* of a database. This is the set of all possible combinations of relation names and database constants. The Herbrand base of the example database is therefore:

{ *stud*(english), *stud*(german), *stud*(latin), *stud*(paul), *stud*(toto), *stud*(jean),
third(english), …, *third*(jean),
lang(english), …, *lang*(jean),
mod(english), …, *mod*(jean),
dead(english), …, *dead*(jean),
learns(english, english), …, *learns*(english, jean),
…
learns(jean, english), …, *learns*(jean, jean) }.

An *interpretation* of a database is a subset of the Herbrand base. One such interpretation of the example database is:

{ *stud*(toto), *stud*(jean), *stud*(paul),
third(jean), *third*(toto),
mod(english), *mod*(german), *dead*(greek), *dead*(latin),
lang(english), *lang*(german), *lang*(greek), *lang*(latin),
learns(paul, english), *learns*(toto, latin), *learns*(jean, greek) }.

The atoms appearing in an interpretation can be considered to be facts of the database—all other atoms can be assumed to be false.

Now, suppose we have a database *DB*, and let *p* be a formula in *DB*. An interpretation *I* of *DB* is said to *satisfy p* if the value of *p* is true whenever the value of each atom in *I* is true. If *I* satisfies *p*, then *I* is said to be a *model* of *p*. An interpretation of *DB* is a model of *DB* if it satisfies every formula in *DB*. For example, the information in the example database can be formalised by the set of formulae (interpretation) shown in Table 2-3. This interpretation is also a model of the database. However, any interpretation that does not include the atom *learns*(paul, english) is not a model; the absence of this atom means that its value in the interpretation is false, so the database formula *learns*(paul, english) will never be satisfied.

A *minimal* model of a database can be obtained by taking the intersection of all the possible models of the database. (Note that for indefinite databases[1], there may be more than one minimal model).

A wff *w* is said to be a *logical consequence* of a set of wffs *W* iff *w* is true in all models of *W*. This is written as $W \vDash w$.

Table 2-3. An interpretation of the example database.

$(\forall x)(third(x) \rightarrow (learns(x, \text{latin}) \vee learns(x, \text{greek})))$		
$\neg(\exists x)(learns(x, \text{greek}) \wedge learns(x, \text{german}))$		
$(\forall x)(mod(x) \vee dead(x) \rightarrow lang(x))$		
stud(toto)	*stud*(jean)	*stud*(paul)
third(jean)	*third*(toto)	
mod(english)	*mod*(german)	
dead(latin)	*dead*(greek)	
learns(paul, english)	*learns*(paul, german)	¬*learns*(toto, english)

2.4.2 Syntax—The Proof-Theoretic View

The proof-theoretic view deals with the syntax of logical formulae. In logic, "syntax" is defined as the derivation of a wff from a given set of wffs. The basis

[1]See page 13.

of proof theory is a set of formulae (*axioms*) and a set of *inference rules*, which are used to infer other formulae. The first-order predicate calculus itself consists of a collection of formulae and the two inference rules MODUS PONENS and GENERALISATION.

A *first-order theory* is obtained from the first-order predicate calculus by the addition of other wffs as axioms (called *proper* or *nonlogical* axioms). For example, a set of nonlogical axioms could be:

$$man(\text{Turing}),$$
$$(\forall x)(man(x) \rightarrow mortal(x)).$$

The proof theory definitions of *interpretation* and *model* are similar to the model theory definitions. Thus, an interpretation of a theory is a subset of the set of axioms, and a model of a theory is an interpretation in which all the axioms are true. A wff w is said to be *derivable* from a set of wffs W under a theory T iff w is deducible from both W and the axioms of T by a finite application of the inference rules. This is written $W \vdash_T w$ (or $W \vdash w$ if T is clear).

It was noted earlier that the first-order predicate calculus has two inference rules. Other inference rules can be used to derive theorems—many theorem-proving systems are based on the ROBINSON RESOLUTION PRINCIPLE [Robinson 1965], which allows a new clause to be derived from two given clauses. This principle is used mainly to perform refutation proofs. Thus, to prove $W \vdash w$, we try to show that W and $\neg w$ are not simultaneously satisfiable (i.e. that the formula $W \wedge \neg w$ is a contradiction). The proof process consists of applying resolution to all clauses in the original set to create new clauses, adding these new clauses to the set, and iterating the process until either the empty clause (false) is reached, or no new clauses are added.

An inference system is said to be *sound* iff

$$(W \vdash w) \rightarrow (W \models w), \forall W, w.$$

That is, if w can be proven from W then w is also a logical consequence of W. An inference system is said to be *complete* iff

$$(W \models w) \rightarrow (W \vdash w), \forall W, w.$$

That is, if w is a logical consequence of W, then w is also provable from W. The inference rules modus ponens, generalisation and resolution are all complete and sound.

2.4.3 Definition of a Relational Database

In general, relational databases have been usually been viewed from a model-theoretic point of view. That is, a database consists of a set of facts which are known to be true. To put it another way, each fact stored in the database has the value “true” associated with it. Note that there are usually no facts in the

database which have the value “false” associated with them. There are important reasons for this, which will be discussed shortly.

Relational databases can also be viewed from a proof-theoretic point of view (i.e. as a particular first-order theory). The proof-theoretic view, as already stated, involves deriving new clauses from existing clauses using inference rules, which is exactly what a deductive database does. Thus a deductive database can be viewed as a special first-order theory.

We will now define the concept of a (deductive) logic database more formally. A database consists of a finite set of constants $c_1, \ldots, c_n$, and a set of first-order clauses that do not include functions.

The general form of clauses that appear in the database is

$$P_1 \wedge P_2 \wedge \ldots \wedge P_k \rightarrow R_1 \vee \ldots \vee R_q.$$

These clauses fall into seven categories, as defined by Minker [1983]. These categories are as follows:

1. $k = 0$, $q = 1$. Clauses are of the form

$$\rightarrow R(t_1, \ldots, t_m).$$

   (a) If all the t_i are constants, then this represents a fact in the database.

   (b) If some or all of the t_i are variables, then this represents a general statement in the database. For example, the clause

$$\rightarrow \textit{ancestor}(\text{Adam}, x)$$

   states that Adam is the ancestor of every other entity in the database.

2. $k = 1$, $q = 0$. Clauses are of the form

$$P(t_1, \ldots, t_n) \rightarrow.$$

   (a) If all the t_i are constants, this represents a negative fact. This may seem rather peculiar, since negative facts are not usually stored in databases. Negative data and their use will be discussed shortly.

(b) If some of the t_i are variables then we can think of this as an INTEGRITY CONSTRAINT (see type 3 below), or as the “value does not exist” meaning for NULLS.

3. $k > 1$, $q = 0$. Clauses are of the form
$P_1 \wedge \ldots \wedge P_m \rightarrow$.

These can be thought of as integrity constraints, and can be interpreted as saying “the conjunction of all atoms on the left-hand side of the clause

implies the empty clause (false).” For example, suppose we wanted to impose the constraint that “no individual can be both the mother and the father of another individual.” This can be represented by the clause
$father(x, y) \wedge mother(x, y) \rightarrow$.

If the fact *father*(Bert, Sally) already exists, then any attempt to add the fact *mother*(Bert, Sally) will fail.

Clauses of this form can also be thought of as queries. In this case they are interpreted as “find $x_1, \ldots, x_n$ such that $P_1 \wedge \ldots \wedge P_m$ is true.”

4. $k \geq 1$, $q = 1$. Clauses are of the form
$P_1 \wedge \ldots \wedge P_m \rightarrow R$.

These may be thought of as integrity constraints or as deductive laws (specifically, the definition of a virtual relation).

5. $k = 0$, $q > 1$. Clauses are of the form
$\rightarrow R_1 \vee \ldots \vee R_n$.

If the terms of the R_i are all constants, then this represents an indefinite assertion. That is, some combination of one or more R_i is true, but we do not know which ones are true and which ones are not.

6. $k \geq 1$, $q > 1$. Clauses are of the form
$P_1 \wedge \ldots \wedge P_m \rightarrow R_1 \vee \ldots \vee R_n$.

These can be interpreted as either integrity constraints or the definition of indefinite data. As an example of the former, an integrity constraint that states “each individual has at most two parents” could be given by the clause
$P(x_1, y_1) \wedge P(x_1, y_2) \wedge P(x_1, y_3) \rightarrow (y_1 = y_2) \vee (y_1 = y_3) \vee (y_2 = y_3)$.

That is, if any individual has three parents, then two of them must be the same person. An example of the latter case is the clause
$parent(x, y) \rightarrow mother(x, y) \vee father(x, y)$.

This states that for any pair of individuals x and y, if x is y’s parent, then x is either y’s mother or y’s father, but we do not know which.

7. $k = 0$, $q = 0$. This denotes always false, and should not be a part of the database.

Conventional databases treat most of these clauses as integrity constraints. In particular, they do not handle indefinite assertions (type 5) very well, if at all. The most common type of clause found in conventional database (in fact, often the *only* type of clause found) is type 1(a). Other types of clause may also be found in the form of integrity constraints. If we want to add a general rule to a conventional database, it usually has to be expressed procedurally by a program written in some host language (e.g. Pascal). However, deductive databases can treat some of these clauses as deductive laws. The most common are type 4 clauses, though type 6 clauses can also appear occasionally.

If a database has no clauses of types 5 or 6 (i.e. no indefinite data), then it is known as a *definite* or *Horn* database (since it contains only Horn clauses). Because they contain no indefinite data, we can apply certain practical assumptions about the nature of the data (see Section 2.5.1). If a database includes indefinite clauses (i.e. clauses of types 5 or 6), it is known as an *indefinite* database.

The deductive laws (i.e. clauses of types 4 and 6) implicitly define a relation in terms of one or more other relations (as stated earlier). These other relations may be other virtual relations, or may be base relations. Relations may also be partly base, and partly virtual (these are often called *hybrid* relations). That is, some of the information in the relation is explicitly stored, and some is derived from rules.

It is interesting to note that rules are a generalisation of the concept of views in conventional databases. A view is a relation that is defined in terms of other relations by way of some form of relational algebra or calculus expression. A virtual relation (rule) reduces to a view when:

- the relation is purely virtual, that is, there are no base (explicit) facts stored in the relation, and
- there are no recursive rules in the definition of the relation.

To summarise, a database can be defined from either the model-theoretic view (i.e. as a a collection of formulae which are assumed to be true), or from the proof-theoretic view (i.e. as a collection of axioms from which data can be derived using inference rules). Conventional database have traditionally been seen from the model-theoretic view, but they can also be seen from the proof-theoretic view, as a particular logic database. Deductive databases can only be seen from the proof-theoretic view.

2.5 Implementation—Practical Considerations

This section briefly discusses some issues and problems that must be taken into account when implementing any deductive database system. Some possible solutions to the problems will be discussed in a later section.

2.5.1 Fundamental Assumptions

There are three fundamental assumptions that are usually made about real world data in order to construct a practical database system. They are [Gallaire *et al*. 1984]:

1. *Unique Name Assumption*: every object in the database has associated with it a unique identifying name. This is equivalent to the notion of primary keys as defined in the relational model.

2. *Domain Closure Assumption*: the universe is restricted to only those objects that are named in the database. If an object does not appear in the domain of the database, it does not exist.

3. *Closed World Assumption* (CWA): if we cannot prove a relationship between some objects to be true, then we assume it to be false. That is, if we cannot find a fact in the database, then the negation of that fact is assumed. This assumption is also known as *negation as failure* (among other things). For example, suppose we have a database that consists of the facts

   $$on(\mathrm{a}, \mathrm{b}),\ on(\mathrm{b}, \mathrm{c}).$$

   Then the CWA allows us to conclude

   $$\neg on(\mathrm{a}, \mathrm{c}),\ \neg on(\mathrm{b}, \mathrm{a}),\ \neg on(\mathrm{c}, \mathrm{a}),\ \ldots.$$

These assumptions allow query evaluation to be carried out more efficiently, but they restrict the kinds of information that can be stored in a database. The CWA, in particular, causes major problems with two important kinds of information—indefinite data and negative data. When applied to indefinite data, the CWA can cause inconsistencies. For example, consider a database consisting of the single fact

$$cat(\mathrm{felix}),$$

and the single rule

$$cat(x) \rightarrow black(x) \vee white(x).$$

Under the CWA, we cannot prove *black*(felix), so we can therefore conclude ¬*black*(felix). Similarly, we can also conclude ¬*white*(felix), giving us

$$\neg \textit{black}(\text{felix}) \wedge \neg \textit{white}(\text{felix}),$$

that is, Felix is neither black nor white. However, this contradicts the original rule!

Negative data also causes problems under the CWA. The CWA allows us to store negative (i.e. false) data implicitly; if it is missing, it is false. If negative data are stored explicitly however, we cannot assume this; rather, we must treat missing data as being *unknown* instead of false.

The alternative to the CWA that immediately springs to mind is the *Open World Assumption* (OWA). That is, if the negation of a predicate is not explicitly stated, then one cannot assume that it exists. However, in general this is impractical—we must explicitly state that each relationship that does not hold is false (as implied above). This leads to an excessive amount of negative information that must be stored.

Because of these problems, database clauses are generally restricted to be Horn clauses, that is, no indefinite data is allowed. However, they can still suffer from the problems with negative data.

2.5.2 Efficiency

A major issue in implementing any database system is that of efficiency. Deductive databases, indefinite ones in particular, also suffer from this problem, perhaps more so than conventional databases. This is usually due to the deduction method used, which is often some variation of resolution. Prolog uses a form of resolution to control its flow of execution, so the problems associated with this approach can be illustrated by the way in which a Prolog program executes—given a clause to be proven, the program starts at the top of the list of predicates, and attempts to unify[2] the clause with one of them. If the unification is successful, the program then tries to prove the right-hand side of the unified program clause. If this fails, the interpreter backtracks and tries the next option, until it either finds one that succeeds, or fails to find any solution. This is known as *SLDNF-resolution*, for "resolution with selection function for definite clauses, augmented by negation as failure." This method is fine if the database involved is small, but as the number of facts gets larger, this type of search can become very inefficient.

One solution to this problem is to introduce the idea of *typing*. That is, each object in the database has associated with it a specific type. This should not be confused with the concept of data types, but is rather the division of the facts in the database into groups, or *domains*, of objects that are of the same kind. For

[2]See ROBINSON RESOLUTION PRINCIPLE in the glossary.

example, the database described on page 7 would have the following set of types associated with it:

{student, language, course}.

The use of types improves the efficiency of the unification process, because even in the worst case, only some of the database need be searched, rather than all of it. A useful side-effect is that an object of a particular type cannot be inserted into a relation unless its type matches the type associated with the relation field. For example, an attempt to insert an object of type “reptile” into a relation that only stores objects of type “mammal” will fail. This is equivalent to automatic integrity checking on the fields of relations. The concept of typing also allows us to assign more meaning to the data, and provides a possible connection with object-oriented databases and semantic nets. Semantic nets, in particular, have already been used for typing in a deductive database [Minker 1978].

Lloyd and Topor [1985] discuss the theory behind typing in greater depth.

2.5.3 How to Use Rules

Conceptually, a deductive database can be divided into two parts: an underlying conventional database (CDB), which contains all of the true facts, and a set of deductive rules. This is not meant to imply that all deductive databases are built like this, however; remember that this is only a conceptual view. We can choose to exploit the underlying CDB in two different ways. The first way is to keep the CDB implicit; in other words, some proportion of the data that might have been stored explicitly in the CDB is instead stored implicitly using rules. The other option is to make the underlying CDB explicit; in this case *all* of the data is stored explicitly; none is stored as rules. In the first case, the rules are said to be used as *derivation rules* (the rules are used to *derive* data) while in the second, they are said to be used as *generation rules* (the rules are used to *generate* data).

An analogy should make these concepts clearer. Consider that the rules are effectively a program written in a high-level language (in this case, first-order logic). There are two ways we can “execute” this program: we can use an interpreter, or we can use a compiler. The derivation approach is equivalent to “interpreting” the rules, that is, the rules are evaluated every time we make a query. Conversely, the generation approach is equivalent to “compiling” the rules, that is, the rules are evaluated once to generate all the possible deducible facts, *before* any queries are made.

One point to note about “generation paths” as opposed to “derivation paths” is that they stop naturally, even if there are recursive rules or cycles amongst the rules [Gallaire *et al*. 1984]. Also, using generation rules can be viewed as an automatic check on the integrity of the database, as the database must be “recompiled” every time it is changed. Unfortunately, this recompilation can be a major disadvantage if the rule base is very large.

An example of a deductive DBMS using generation rules is the BDGEN system described by Nicolas and Yazdanian [1982]. An example of a deductive DBMS using derivation rules is Minker's MRPPS [Minker 1978].

2.5.4 Inference Methods

There are two methods for performing the inference process: the *interpretive* method and the *compiled* method.

The interpretive method interleaves the deduction process with searches of the extensional database. That is, it performs some deduction, then accesses the EDB, then performs some more deduction, and so on. In the compiled method, deduction is performed until either the problem is solved, or all that remains is to search for facts in the EDB. That is, the deduction and data retrieval operations are kept separate.

Both of these methods work well when the rules are assumed to be free of cycles and recursion. If this is not the case, then they both have difficulties handling the termination problem, that is, recognising the point at which no further new solutions will be found. However, if we limit ourselves to definite deductive databases, then we should be able to find some form of terminating condition, because only a finite number of tuples can be generated in response to any given query (see Section 2.6.3).

Unfortunately, both of these methods have problems associated with them. The interpretive approach has efficiency problems, since it may have to access the EDB several times during query evaluation. Also, the order in which subproblems are evaluated can have a considerable effect on efficiency. Because of this, we must at each step invoke a selection function and a choice function [Gallaire *et al.* 1984]. It is interesting to note that this is the same problem that plagues Prolog.

The main problem with the compiled approach is that it can return redundant answers. When using a database system, we expect to get back the set of all possible answers, not just *an* answer. The simplest way to do this is to backtrack once we have found an answer, and explore the alternate deduction paths. Unfortunately, different deduction paths may lead to the same answer. One possible solution to this problem is an iterative method proposed by Henschen and Naqvi [1984], which uses queues instead of backtracking. This is more general, but can still result in redundancy.

Examples of how both of these inference methods work can be found in Appendix A.

2.6 The Major Problems and Some Possible Solutions

There are many problems that remain to be solved in the field of deductive databases. This section lists the major problems, and some proposed solutions to them.

2.6.1 Indefinite Data

There are many different kinds of indefinite data. The most common form found in databases currently is the *null*. Nulls can generally be interpreted in two ways:

1. "value not known but is one of the objects within the database", or

2. "value not known and is not one of the objects in the database."

The second option destroys domain closure, so it is usually ignored. That is, the null is assumed to be one of the objects within the database. For example, suppose the database consists of

$$P(\mathrm{a}, \mathrm{b}), P(\mathrm{b}, \mathrm{c}), P(\mathrm{a}, \mathrm{c}).$$

If we add the fact $P(\mathrm{a}, \omega)$, where ω is the null, then we can represent it as

$$P(\mathrm{a}, \mathrm{b}) \vee P(\mathrm{a}, \mathrm{c}).$$

This presents problems if there are a large number of objects in the database, which can result in the disjunction of a large number of literals [Grant and Minker 1986].

As mentioned earlier, indefinite data can also cause inconsistencies under the CWA. There have been several formalisms proposed in recent years to solve this problem, some of which are listed briefly below (see Chisholm *et al.* [1987] for more detailed descriptions). The most promising appear to be circumscription, subimplication and negation as inconsistency, though they too have problems associated with them, generally with efficiency, but also with ease of implementation.

- *Generalised CWA* (GCWA) [Minker 1982]. This is an extension of the CWA to handle indefinite data. Basically, it states that if a fact cannot be found in the database, then assume it is false, *unless* it is part of an indefinite clause. This would solve the problem in the example with Felix the cat—even though we could not prove *black*(felix) or *white*(felix), both *black* and *white* are part of the clause

  $$cat(x) \rightarrow black(x) \vee white(x),$$

  so we cannot assume that either of them is false. Therefore, Felix is either black or white, *but we do not know which*.

- *Extended GCWA* (EGCWA) [Yahya and Henschen 1985]. This is an extension of the GCWA which allows non-atomic ground negative clauses as answers to queries. Specifically, we can assume the clause

  $$\neg L_1 \vee \ldots \vee \neg L_n$$

to be true iff it is true in every minimal model of the database, that is, no minimal model of the database contains all of the atoms $L_1, \ldots, L_n$. For example, suppose we have the database

$$p, q, r \vee s.$$

This has two minimal models: p, q, r and p, q, s. Thus we can assume the following negative clauses:

$$\neg r \vee \neg s,$$
$$\neg r \vee \neg q \vee \neg s,$$
$$\neg p \vee \neg r \vee \neg q \vee \neg s,$$
$$\ldots$$

- *Subimplication* [Bossu and Siegel 1984, 1985]. This method is based on the contents of the intersection of the set of all minimal models of the database. Answers to queries may be one of "yes", "no" or "indefinite", depending on whether the query is true in, respectively, all, none or some of the minimal models. Unfortunately, this method slows down as the amount of indefiniteness (i.e. the number of non-Horn clauses) in the database increases. In the worst case, the running time of the algorithm can be exponential in the size of the database.

- *Circumscription* [McCarthy 1980a, 1980b]. Circumscription is a NON-MONOTONIC formalism that allows one to "jump to certain conclusions" from the given facts, even though these conclusions may not be strict logical consequences of what is already known. It has been shown that the CWA is in fact a special case of circumscription [Lifschitz 1985].

  Circumscription can be formally defined as follows: let *DB* be the conjunction of all clauses in the database, *P* be a predicate symbol and *A* be a predicate expression of the same ARITY as *P*. Denote by *DB*(*A*) the result of replacing all occurrences of *P* in *DB* by *A*. The *circumscription* of *P* in *DB* is then:

$$(DB(A) \wedge (\forall x)(A(x) \rightarrow P(x))) \rightarrow (\forall x)(P(x) \rightarrow A(x)).$$

  The main problem with circumscription is automatically determining a suitable formula for the predicate *A* [Reiter 1982].

- *Negation as Inconsistency* [Gabbay and Sergot 1986]. This is a slightly weaker version of classical logical negation. Failure to prove a fact does not mean that fact is false, but rather that it *cannot be proven*. That is, the fact is *inconsistent* with the existing database.

2.6.2 Negative Data

Generally, negative data are not stored explicitly in a database—instead they are implicitly represented using the CWA. However, the CWA can cause problems with negated queries, for example, consider the database

$$P(\mathrm{a}), P(\mathrm{b}), Q(\mathrm{c}), Q(\mathrm{d})$$

From the CWA we can assume $\neg P(\mathrm{c})$ and $\neg P(\mathrm{d})$. Now the query $\neg P(\mathrm{c})$ will succeed, because $P(\mathrm{c})$ fails, but consider the query $\neg P(x)$. It can be argued that this should return the answer $x = \mathrm{c} \vee \mathrm{d}$ (under domain closure), but it does not because $P(x)$ can be proven by matching with either $P(\mathrm{a})$ or $P(\mathrm{b})$, and so the entire query fails.

Some of the formalisms mentioned in the previous section on indefinite data also address the problem of negative data, especially negation as inconsistency.

2.6.3 Recursion

Recursive rules are another problem. A classic example of a recursive rule is the definition of *ancestor*:

$$parent(x, y) \rightarrow ancestor(x, y)$$
$$ancestor(x, y) \wedge parent(y, z) \rightarrow ancestor(x, z)$$

When we introduce recursive rules into a database, we have a problem with terminating the use of these rules, as they may lead to infinite derivation paths. This is only really a problem with indefinite deductive databases though—in definite deductive databases axioms are function-free, are Horn clauses and contain a finite number of symbols. Thus, only a finite number of tuples that can be generated, so the derivation must terminate eventually (though *when* is still an open question).

Several approaches have been developed for processing recursion in a database. Date [1990] provides a brief overview of the simpler methods; a more thorough summary of recent methods can be found in Bancilhon and Ramakrishnan [1986].

It has been recently suggested [Tsang 1990] that recursive query processing could be handled by specialised hardware, and that it is not absolutely essential to develop further techniques.

2.6.4 Rules: Deductive Laws or Integrity Constraints?

How do we decide whether to interpret a general rule as a deductive law or as an integrity constraint? There are no set guidelines, only some general suggestions [Gallaire *et al.* 1984]:

- deductive laws have to be function free to obtain finite and explicit answers to queries. If a clause does not satisfy this restriction, it should be an integrity constraint.

- only use definite clauses as deductive laws—this avoids inconsistency with the CWA. Other clauses should be used as integrity constraints. (Note that this applies to definite databases only.)

- purely negative clauses will never produce any new facts, so use them only as integrity constraints.

- general knowledge that does not add any new information to the database should be used as integrity constraints.

A related problem is deciding, when designing a database, which relations should be base relations, and which should be virtual relations. This is really a subjective decision on the part of the database administrator, based on the needs of the system and the users of the database. For example, if a large and possibly complicated table can be generated from a simple rule, you should, if possible, use the rule.

2.6.5 Functions

Should we allow functions into database clauses? Though this allows more generality in the kinds of data we can store, it also causes major problems, due to the introduction of *intensional entities* into the database. For example, although we do not know who John's father is, there is nothing to stop us forming the description *father*(John). Different descriptions may denote the same individual, so how can we test the equality of such objects? One solution is to allow only those functions whose results can be computed.

Functions also cause problems in that they may result in infinitely many answers to a query. Consider the database:

$$P(0)$$
$$(\forall x)(P(x) \rightarrow P(f(x)))$$

and the query "Find all x such that P(x)." The answer to this query is the infinite set $\{0, f(0), f(f(0)), f(f(f(0), \ldots\}$. The only solution to this problem is for the system to return *descriptions* of sets, rather than just their contents. Thus, in the previous example, it might return the answer $\{x: x = 0, f(x)\}$.

2.7 An Historical Survey of Deductive Database Research

2.7.1 Early Work (1957–1978)

The initial ideas behind deductive databases evolved out of the work done in the 1960's on automated deduction systems. The earliest known deductive system was developed in the late 1950's [Gurk and Minker 1961, Minker 1988] for the U.S. Army. This was a prototype database system that performed inferences on Army intelligence data.

There was much research into automated deduction during the 1960's, but the emphasis was on deduction as an artificial intelligence problem rather than anything else. There was a great deal of research into automated theorem-proving and question-answering systems, and many important results and methods were developed from this. In particular, the resolution principle was proposed by Robinson in 1963 [Robinson 1965]. Research into deductive databases as a separate entity was fairly minimal.

The period from 1970 to 1977 saw database research expand considerably, spurred on mainly by Codd's landmark paper on the relational model [Codd 1970]. More people began to research into deductive databases and the related field of logic programming. Kowalski first proposed using logic as a programming language in 1974 [Kowalski 1974]. The first logic programming languages (Prolog in particular) appeared during this period. Several prototype deductive database systems were also developed during this time (for example, Minker's MRPPS [Minker 1978]).

2.7.2 The *Logic and Data Bases* Period (1978–1986)

In 1978 the book *Logic and Data Bases* was published [Gallaire and Minker 1978]. This was the proceedings of a workshop on deductive databases held in Toulouse, and had a major impact on the fields of deductive databases and logic programming. Most of the papers given at the workshop were very significant: Nicolas and Gallaire introduced the idea that deductive databases are based on proof theory as opposed to model theory [Nicolas and Gallaire 1978]; Reiter introduced the Closed World Assumption [Reiter 1978]; Clark presented some new results on negation [Clark 1978]; Kowalski discussed using logic to describe data [Kowalski 1978]; and there were reports on several implemented systems [Minker 1978, Chang 1978, Kellogg *et al.* 1978].

To say that this book has been the basis of most of the research into deductive databases in the last decade would not be an understatement. It focussed attention on the use of logic for deductive databases, unlike anything that had gone before. However, it cannot take all the credit—shortly after the book was published, the Japanese announced their "Fifth Generation Project", which aimed to develop architectures based on the principles of logic programming. This effort has been important in directing people's attention towards logic programming and deductive databases.

This period has been characterised by the development of a solid theoretical background for logic programming and deductive databases. Most of the formalisms that were developed to alleviate some problem or other with deductive databases were introduced during the last decade (including the GCWA, EGCWA, subimplication and circumscription). Lloyd and Topor have developed a sound theoretical basis for deductive database systems using a typed first-order logic to express data, queries and integrity constraints [Lloyd and Topor 1984, 1985, 1986].

2.7.3 The "State of the Art" (1986–Present)

Much of the current "rush" of deductive database research was inspired by Reiter's paper on logic and the relational model [Reiter 1984]. In some ways, this paper did for deductive databases what Codd's original paper [Codd 1970] did for database theory in general.

What sort of areas are researchers currently looking at? This section discusses some of the current ideas and issues, and attempts to make some sort of survey of recent research into deductive databases. Note that this is by no means an exhaustive list.

"Classical" Problems

The problems that are listed in Section 2.6 have not gone away, though there have been many solutions or partial solutions proposed. Indefinite and negative data have become less troublesome than they were in earlier years; current research into areas like non-monotonic reasoning, circumscription and higher-order logic looks very promising.

Recursive query processing has become more efficient due to the introduction of recent techniques such as *magic functions* [Bancilhon *et al.* 1986, Gardarin 1987]. As stated earlier, it has been claimed that recursive query processing could be handled by specially optimised hardware.

Stratified Databases

One recent development are what are known as *stratified databases*. These deal with extended Horn clauses that have negated atoms in the antecedent (they are extended because the antecedent of a Horn clause normally cannot contain negated atoms). A database is said to be *stratified* if the clauses can be ordered such that if a negated atom appears in the antecedent of a clause, the definition of that atom precedes the clause in which it appears in the antecedent. For example, consider a database that contains these two clauses:

$$P \wedge Q \wedge \neg N \rightarrow R$$
$$A \wedge B \rightarrow N$$

In this case, the only negated atom is $\neg N$. This database is stratified, because the clauses can be ordered so that the definition of N (i.e. the clause of which N is the consequent) precedes the clause containing $\neg N$ in its antecedent. That is, the following ordering of clauses is possible:

$$A \wedge B \rightarrow N$$
$$P \wedge Q \wedge \neg N \rightarrow R$$

A stratified database can be said to be "free of recursive negation" [Minker 1988], as stratification prevents recursion on negation. Several important results have been proven about stratified databases.

Database Design Methodologies

One area that has recently appeared in the literature is design methodologies for deductive databases. How should a deductive database be designed at the logical and the physical levels?

Helman and Veroff [1988] discuss a formal design methodology, the intent of which is to produce "a physical deductive database schema which supports the initial [logical] schema at minimal cost with respect to the anticipated transactions and the given cost function." That is, they are attempting to build an efficient physical schema which is optimised for a particular set of transactions. They do this by passing the initial logical schema through a series of *schema transformations*. The resulting schema is then evaluated against a cost model, the aim being to find the schema with the minimum cost.

The inputs to the methodology are:

- An existing deductive database schema *S*, consisting of an EDB schema, an IDB and specifications of any indexes and join supports. The IDB may contain either derivation rules or integrity constraints (e.g. functional dependencies).

- A set of queries and their associated frequencies. This is used by the cost model to evaluate different schemas proposed by the methodology.

- An expected population distribution. This is also used by the cost model. It includes (at least) an indication of the number of tuples expected in each relation in the EDB. Other types of information, such as "attribute *A* in relation *R* will have the following range of values with the following distribution" would also be useful.

There are eleven transformations that can be applied to the schema, ten of which can be grouped into pairs of conceptual inverses:

1. *Join/Decompose*. *Join* creates a new relation R which is the join of two existing relations R_1 and R_2. The original relations remain intact. This might be desirable if there are many queries which require the join of R_1 and R_2. *Decompose* divides a relation R into two new relations R_1 and R_2. The original relation is deleted from the EDB. This is useful for normalisation, and might also be desirable if there are queries which ask for a specific subset of the attributes of R.

2. *Generalise/Specialise*. *Generalise* combines two relations R_1 and R_2 with the "same" attributes into a single relation R. The original relations are deleted from the EDB. This would be useful if there are queries that access both R_1 and R_2, and that would benefit from a common index. *Specialise* divides the rows of a relation R, forming two new relations R_1 and R_2. The original relation is deleted from the EDB. This might be useful if there are queries that ask for a specific subset of the tuples of R, based on some set of attribute values (cf. *Decompose*).

3. *Copy/Delete*. *Copy* creates an identical copy R_2 of an existing relation R_1. *Delete* deletes a relation *R* from the schema.

4. *Move from IDB to EDB*. This transformation creates a new explicit relation *R* from one which is already implicitly defined in the IDB. This might be desirable if there are many queries against *R*.

5. *Add index structure/Delete index structure*. *Add index* builds an index structure for a relation *R*. This is useful if there are queries that involve a qualification on a specific subset of attributes of *R*. *Delete index* deletes an existing index structure.

6. *Add join structure/Delete join structure*. *Add join* builds a join structure between two relations R_1 and R_2. This is useful if there are queries that require the join of R_1 and R_2, but do not justify the creation of the explicit join relation (see *Join* above). *Delete join* deletes an existing join structure.

Once the schema has been transformed, it is evaluated against the cost model, which measures the cost of the schema in terms of the amount of storage taken up by the database, and the amount of time required to process the required queries. It appears that the problem of automatic schema optimisation is NP-complete; this has not yet been proven however. Because of this, a heuristic approach is taken for the optimisation of schemas.

Work is still continuing in this area.

New Data Models

There has been some interest in developing a data model that directly supports deduction. A model that claims to be such is the *partition model* [Spyratos 1987]. This section will give a very brief overview of the basic ideas behind the model.

The partition model is based on the idea of *partitioning* a set of objects into disjoint subsets (which can include the empty set). Each subset in the partitioning represents all objects with some given property, and is known as a *fact*. For example, suppose we want to store information about seven vehicles, numbered 1 to 7, of which 1, 2 and 3 are buses, 4 and 5 are limousines and 6 and 7 are vans. To represent the *type* of the vehicle, we would partition the set of vehicles as follows:

$$\{\{1, 2, 3\}, \{4, 5\}, \{6, 7\}\}$$

If we make the substitution $b = \{\{1, 2, 3\}\}$, $l = \{\{4, 5\}\}$ and $v = \{\{6, 7\}\}$, we can denote the set of all types of vehicle by writing $type = b \cup l \cup v$.

What if we also wish to store information about where the vehicles are stationed? Suppose vehicles 1 and 2 are stationed in Wellington, 3, 4 and 5 in Christchurch and 6 and 7 in Dunedin. With respect to this property, the set is partitioned as follows:

$$\{\{1, 2\}, \{3, 4, 5\}, \{6, 7\}\}$$

Making the substitution $W = \{\{1, 2\}\}$, $C = \{\{3, 4, 5\}\}$ and $D = \{\{6, 7\}\}$, we can denote the set of all vehicle stationings by writing $station = W \cup C \cup D$ (as above).

Given these two partitions, we can extract further information. For example, suppose we want to find all vehicles that have the same type *and* the same station. We can find this by taking the product of *type* and *station* (denoted $type \times station$), where product is defined as all possible intersections of a member of *type* with a member of *station*, that is:

$$\begin{aligned} type \times station &= \{b \cap W, b \cap C, b \cap D, l \cap W, l \cap C, l \cap D, v \cap W, v \cap C, c \cap D\} \\ &= \{\{1, 2\}, \{3\}, \emptyset, \emptyset, \{4, 5\}, \emptyset, \emptyset, \emptyset, \{6, 7\}\} \end{aligned}$$

(where Ø represents the empty set). A fact is false if it is empty, otherwise it is true. For example, $b \cap D = \emptyset$, so this fact is false (i.e. there are no buses in Dunedin).

Spyratos goes on to transform the set of all partitionings into a LATTICE, then defines a database as a function from a sublattice into the lattice of partitionings. He also describes how rules may represented in this model and describes a process for deductive query answering. Finally, he shows how the relational model may be embedded within the partition model.

The model as it stands is restricted to definite databases only. It also does not consider the problems of incomplete information and efficient updating. Apart from these omissions (which Spyratos himself acknowledges), there is nothing obviously wrong with the model, and it would be interesting to implement a system based on it.

Integrated vs. Homogeneous

There have generally been two approaches to implementing a deductive database system. The first approach is to take a separate inference unit and a DBMS and integrate them in some way. These are known as *integrated* deductive databases. The second approach is to implement the deductive database as a single entity; the inference and DBMS components are functionally dedicated modules of a larger system, rather than independent components. Such databases are known as *homogeneous* deductive databases.

There has been much interest in integrated deductive databases in recent years. The main reason for this is probably because existing systems can be used, making the implementation process that much easier. In the long term, a true homogeneous deductive DBMS would be more preferable, but there is no reason why the current experiments with integrated databases would not lead to practical homogeneous systems.

Integrated deductive databases are discussed further in Chapter 4.

2.7.4 Implementations—A Comparison of Some Systems

The first major deductive database systems were implemented in the late 1970's (see Gallaire and Minker [1978] for details). Several systems have been developed since that time as well.

Some of the more notable and/or interesting systems that have been developed are listed below. The method of implementation (i.e. integrated or homogeneous) is listed in parentheses after each description.

- MRPPS-3.0 [Minker 1978] was developed by Minker and his students at the University of Maryland (hence MRPPS: *Maryland Refutation Proof Procedure Systems*). Earlier versions were pure theorem-proving systems, incorporating a large number of inference methods and search strategies; 3.0 was oriented towards deductive data retrieval. The system used an interpretive approach to deduction, and was designed to handle large databases. Unfortunately, it only produced a single answer to any given query, that is, it did not find all possible answers. Also, it is unclear as to how efficient it really was with large databases, as it appeared to have never been tested with one. Typing was built into the representation using a semantic network, which could be compiled and incorporated as part of the unification algorithm. This system is still one of the best examples of an interpretive system. (Homogeneous.)

- Dahl's system [Dahl 1982] used a many-sorted logic, and introduced the notions of set expressions and distributive and collective relations. For example, the statement "A and B are parallel" is an example of a collective relation (it applies to all objects mentioned in the statement), while the statement "John and Mary laugh" is an example of a distributive relation (it is true if both laugh, or neither of them laugh, but undefined otherwise). The system used a three-valued logic (true, false, unknown). Dahl argued that this was necessary for cases where the answer to a query is undefined; the system should say "answer unknown" rather than "no answer found", on the grounds that the user might be misled by the former. (Appears to be integrated, but this is not clear from the paper.)

- MU-Database [Naish and Thom 1983] was developed by Naish at the University of Melbourne (hence MU). It uses the idea of *partial match queries* (i.e. the specification of some of the secondary keys of a record) to improve efficiency. It also attempts to optimise queries, something that seems to have been neglected in many other systems. The system itself is written in C and a dialect of Prolog called MU-Prolog (developed by Naish as part of his Master's degree), which uses a more logical approach to handling negation than just simple negation as failure. The system is based on UNIX, and uses the UNIX facility of multiple background processes to perform deduction in parallel. Each deductive path resulting from a query is passed to a different process, and all are evaluated concurrently. This is one of the few systems that actually addresses the problems of efficiency—many of the others seem to be more interested in the theoretical aspects of deduction, and often ignore the practical aspects. (Appears integrated.)

- Boas and Boas [Boas and Boas 1986] looked at implementing a deductive DBMS by embedding a logic programming language into a relational

database system, rather than adding database facilities to a logic programming language, which is the more usual approach. The system that they were working with was IBM's Business System 12. Unfortunately, there has been no further word on how this project has developed. (Integrated.)

- Exegesis [Small 1988] is a system developed at London University. It forms the middle layer of an integrated software system called TRISTARP-1. It is a basic definite deductive database, with a couple of additional interesting features:

  – *Default rules* can be specified. These enable the DBMS to make assumptions about "missing" information. Default rules represent laws which are "usually" true, for example: "if something is a bird, then in the absence of information to the contrary assume it can fly." They can also be used to implement the CWA for selected predicates. The operator *M* is defined for this purpose. It can be loosely interpreted as "it is consistent to believe."

  – Integrity constraints which define valid database states can be specified. Integrity constraints also use the *M* operator.

  The deduction method used is an interpreted method similar to that used by Prolog. (Integrated.)

2.8 Future Directions

Many issues have yet to be addressed in the field of databases. Handling deletions and insertions efficiently and intelligently has not yet been achieved. Effective computation methods are also required for different classes of database, for example, databases that are non-Horn, contain nulls or are not function-free. Current work into obtaining approximate answers to queries looks promising.

As the fields of deductive databases and logic programming grow, they are moving more and more towards integration with many of the ideas espoused by artificial intelligence (the current "bandwagon" seems to be "AI and databases"). Already deductive databases make use of many AI principles, and there is no reason why this growing together should stop. Indeed, it has become apparent that as AI systems become larger and more complex, they will need some method of accessing and manipulating a large amount of data and/or rules. This will most likely be in the form of a deductive database or similar system. In fact, it is reasonably likely that the boundary between expert systems and deductive databases, which is already fuzzy, will disappear completely.

2.9 Summary

In this chapter we have introduced some of the basic concepts of first-order logic, and have seen how a deductive relational database may be defined in first-

order logic as a special first-order theory over a set of clauses. That is, given a set of clauses and a set of inference rules, we may derive new clauses from the original set using the inference rules.

There are, however, problems with this definition. We have covered some of the major problems associated with deductive databases: negative data, indefinite data, recursion and functions; and some possible solutions to these problems have been proposed. We have also noted the practical assumptions that are usually made when developing a deductive DBMS.

Finally, we have covered the history of deductive databases: how the field started, and how it has evolved since. We have briefly summarised some of the more recent work and current research interests, and some of the currently working implementations. Finally we have taken a brief look ahead at some possible future directions in the field.

Chapter 3

Design Considerations

3.1 Introduction

This chapter covers some issues and ideas that had to be taken into consideration during the development of SQUALID. The structure of this chapter is as follows:

- Section 3.2 discusses two structures for representing rules, and their advantages and disadvantages.
- Section 3.3 introduces what is called the *dynamic view*—the concept of translating a rule into a query in SQL, in a way similar to that in which views are defined. An algorithm for performing this translation is also described.
- Section 3.4 discusses extensions that were made to SQL to handle deduction and rules, and some of the issues that these extensions raise.
- Section 3.5 summarises the chapter.

3.2 Representations for Storing Rules

This section describes two representations for storing rules. But first, a list of important points. A rule representation should have the following features:

- it should be dynamic—the number of rules is likely to be highly variable, and the internal structure of the rules themselves will also vary quite considerably, so a static structure is not satisfactory. Thus, some sort of graph-oriented (i.e. pointer) structure would be the best choice.
- it should be able to support the main deductive method to be used, but also be extensible so that other deduction algorithms can be used.
- it should represent the rules efficiently, both in terms of access time and storage used.

One idea that has been proposed is that rules could be stored in relations, either literally or logically. There are two advantages to this scheme:

- it standardises the representation of rules and data. Prolog (i.e. logic programming) does this by encoding data procedurally as rules. Storing rules in relations is the opposite of this: encoding procedures as data.

- it is consistent with Codd's twelve rules of relational databases, especially the first rule, which states: "All information in a relational database is represented explicitly at the logical level and in exactly one way—by values in R-tables."

Both of the representations discussed below may be converted into a relational equivalent, with varying degrees of efficiency. The relational storage of each representation will be discussed separately.

3.2.1 Hierarchical Tree Representation

A general rule (in clausal form) looks like this:

$$L_1 \wedge L_2 \wedge \ldots \wedge L_p \rightarrow R_1 \vee R_2 \vee \ldots \vee R_q.$$

On initial inspection, it was observed that a rule in this form can be represented hierarchically in a tree-like structure (hence the name hierarchical tree representation, or HTR for short). See Figure 3-1 for an example of this representation.

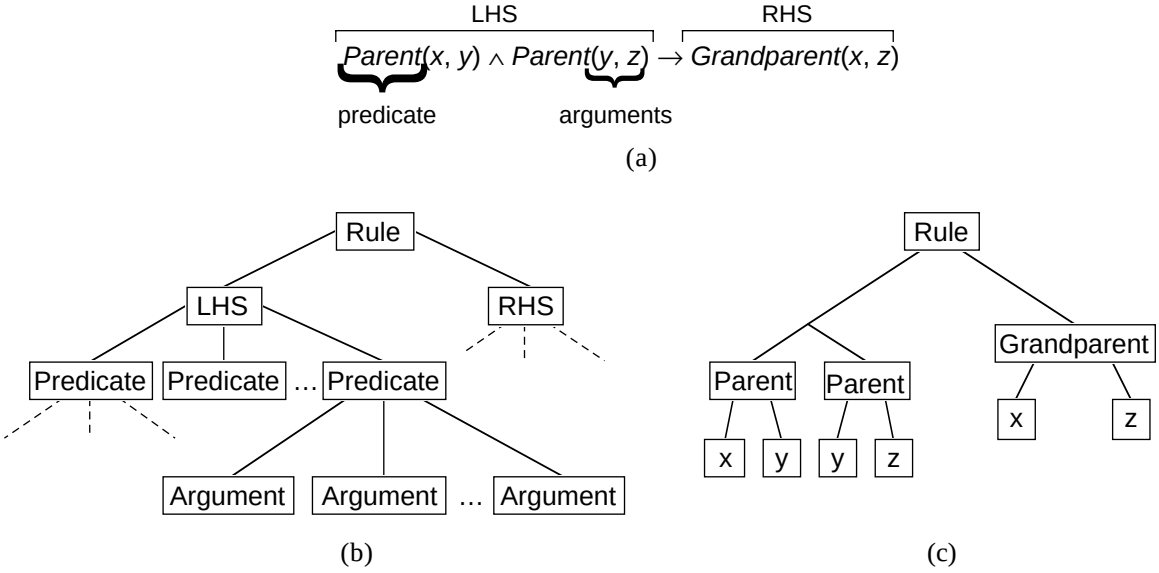


Figure 3-1. (a) The components of a rule; (b) Generic tree structure for a rule; (c) HTR corresponding to the rule $Parent(x, y) \wedge Parent(y, z) \rightarrow Grandparent(x, z)$.

However, the simple structure shown in Figure 3-1(c) misses out a vital factor—the fact that many of the arguments refer to the same "object". Thus, the structure in Figure 3-1(c) captures the syntax of a rule, but omits some of the semantics. To remedy this situation, those arguments which refer to the same object must be combined; see Figure 3-2 for an example. This makes building the HTR more complicated—for each argument, you must check that it does not already exist before creating it.

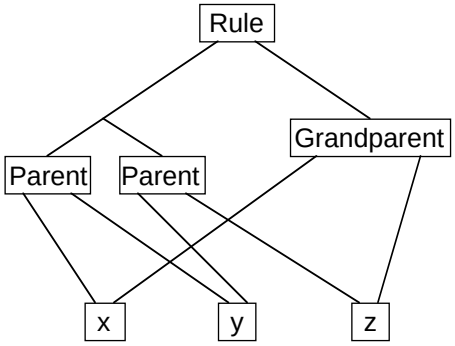


Figure 3-2. What the HTR *should* look like.

This representation is really a naive approach to storing rules, however. One major disadvantage of HTRs is the inability to collect related rules together into one structure. For example, if the *Parent* relation in Figure 3-2 was a virtual relation itself (which is entirely possible), there would be no easy way of including the structure of the *Parent* rule into the structure of the *Grandparent* rule. If we are to do this (i.e. group related rules together), we need a more free-form structure than a tree. One such structure is discussed in Section 3.2.3.

3.2.2 Storing HTRs in Relations

After analysing the general HTR structure shown in Figure 3-1, it became obvious that it was impossible to store a HTR structure in a single, normalised relational table—HTRs (and rules) are a one-to-many construction. To be specific, each rule has many predicates, and each predicate has many arguments. Because of this, the only efficient way of storing HTRs in a relational database is to use a collection of relations. If we translate the HTR structure directly into relational form, we obtain something similar to the four relations listed below (Δ denotes the primary key).

1) Relation RULES—links left- and right-hand sides with rules:

Field Name	Domain	Description
RULE_ID Δ	ID code	ID code for the rule
LHS_ID	ID code	ID code of the rule's LHS
RHS_ID	ID code	ID code of the rule's RHS

2) Relation PREDICATES—links predicates with left- and right-hand sides:

Field Name	Domain	Description
SIDE_ID Δ	ID code	ID code of the side
PRED_ID Δ	ID code	ID code of a predicate in the side
NAME	String	Name of the predicate
ARITY	Integer	Arity of the predicate

3) Relation ARGUMENTS—links arguments with predicates:

Field Name	Domain	Description
PRED_ID Δ	ID code	ID code of the predicate
ARGUMENT_NO Δ	Integer	Position of the argument (1, 2, ..., n).
VALUE_ID	ID code	ID code of the object this argument refers to

4) Relation VALUES—actual values of arguments:

Field Name	Domain	Description
VALUE_ID Δ	ID code	ID code of the object
TYPE	Logical	CONSTANT or VARIABLE
VALUE	String	Value of the object, if applicable

The most important consideration with this set of relations is probably efficiency, in both rule retrieval and the storage occupied by the relations. One observation is that the size of the ARGUMENTS table will grow very quickly as more rules are added. To be more specific, consider the following (assuming a database with *n* rules):

- For each rule, there is only one row in the RULES table, so the size of RULES is *n*.
- For each predicate, there is one row in the PREDICATES table. Each rule can have several predicates associated with it, so the worst-case size of PREDICATES is approximately $O(n^2)$.
- For each argument, there is one row in the ARGUMENTS table. Each predicate can have several arguments, so the worst-case size of ARGUMENTS is approximately $O(n^3)$.
- The number of rows in VALUES is completely independent of the number of rules—it depends only on the number of unique "objects" referred to in the rule base. Because of this, the size of this table is entirely dependent on the content of the rule base.

Of course, this analysis is far from precise, but it should give some idea of the magnitudes involved. The ARGUMENTS table would appear to cause the most problems—as the number of rules increases, the size of this table can increase very rapidly. This happens when the rules are very long and/or complex, but even with relatively simple rules, this table still grows more rapidly than is really desirable.

An Example of Storing HTRs in Relations

Using the set of relations described above, the HTR in Figure 3-2 would be stored as follows:

RULES

RULE_ID	LHS_ID	RHS_ID
RULE1	LHS1	RHS1

PREDICATES

SIDE_ID	PRED_ID	NAME	ARITY
LHS1	PRED1	Parent	2
LHS1	PRED2	Parent	2
RHS1	PRED3	Grandparent	2

ARGUMENTS

PRED_ID	ARGUMENT_NO	VALUE_ID
PRED1	1	V1
PRED1	2	V2
PRED2	1	V3
PRED2	2	V4
PRED3	1	V5
PRED3	2	V6

VALUES

VALUE_ID	TYPE	VALUE
V1	VARIABLE	x
V2	VARIABLE	y
V3	VARIABLE	y
V4	VARIABLE	z
V5	VARIABLE	x
V6	VARIABLE	z

We can see from this that the sizes of the relations can become large very quickly, even when there is only one rule.

3.2.3 Production Compilation Networks

This is a method of representing rules developed by a group in France [Cheiney and de Maindreville 1989]. Rules are compiled into a directed graph representation called a production compilation network (PCN), which is an an execution model for the rule language RDL1. This structure is more general than the HTR structure, and in particular allows related rules to be grouped together into a

single PCN (thus introducing the notion of rule groups or *modules* as they are called in RDL1).

Each rule is represented by a *transition*, and each relational predicate involved in the rule by a *place*. The relational predicates that occur on the LHS of a rule are the "input" places to the transition representing the rule, and the predicates on the RHS are the "output" places of the transition. The condition part of the rule is represented in the transition's *inscription*. These terms should be made clearer in the following example.

Consider the following rule definitions:

$$Parent(x, y) \rightarrow Ancestor(x, y)$$
$$Parent(x, y) \wedge Ancestor(y, z) \rightarrow Ancestor(x, z)$$

The equivalent rule module in RDL1 is:

```
MODULE : ANCESTOR
target : ANCESTOR {Asc, Desc};
rules  :
PARENT (x) → + ANCESTOR (x);
PARENT (x) and ANCESTOR (y) and x.Child = y.Asc
           → + ANCESTOR (Asc = x.Par, Desc = y.Desc );
end.
```

The PCN corresponding to this pair of rules is shown in Figure 3-3. Two relational predicate names appear in the rules. These are equivalent to the places *Parent* and *Ancestor* (represented by the circles P and A in Figure 3-3). The first rule is represented by the transition T_1, whose inscription is the formula **true** (a parent is always an ancestor). The second rule is represented by transition T_2, whose inscription is the formula *P.Child* = *A.Asc*. The arcs leaving *Parent* are labeled (*P.Par*, *P.Child*); this specifies the columns of *Parent* that are to be input into the transition. Similarly, the arc leaving *Ancestor* is labeled (*A.Asc*, *A.Desc*). Finally, the arc from T_2 to *Ancestor* is labeled (*P.Par*, *A.Desc*); this defines the columns of the derived relation in terms of the input relations.

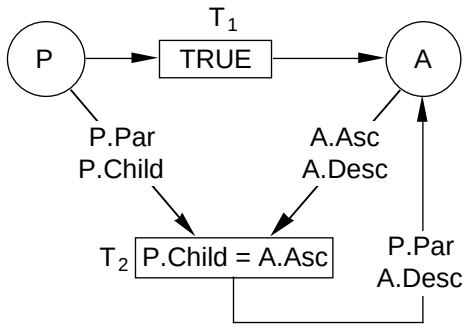


Figure 3-3. PCN for the *Ancestor* Rule Module.

The big advantage of this representation over HTRs is that multiple related rules can be represented in the same graph—Figure 3-3 is a case in point. As al-

ready stated, HTRs cannot easily store multiple rules in a single structure. To further demonstrate this point, consider the following pair of rules:

$$Parent(x, y) \wedge Male(x) \rightarrow Father(x, y)$$
$$Father(x, y) \wedge Parent(y, z) \rightarrow Grandfather(x, z)$$

The PCN representing this pair of rules is shown in Figure 3-4.

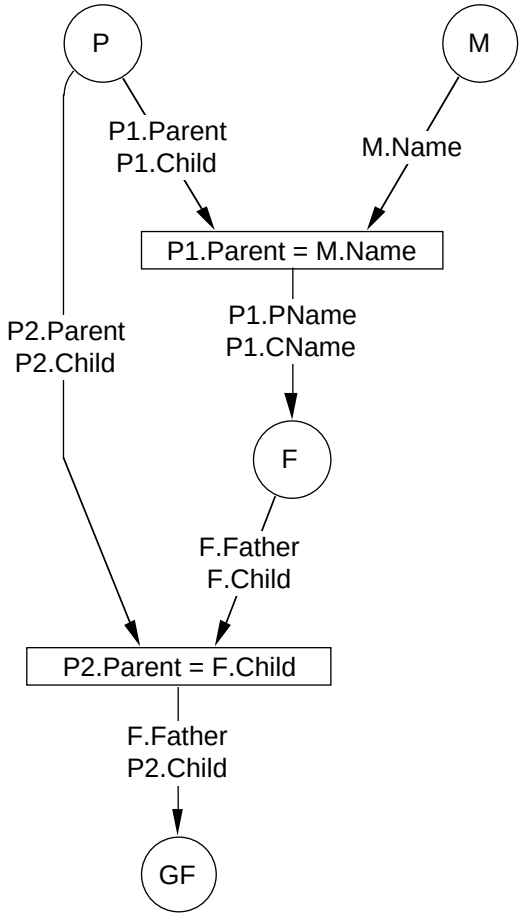


Figure 3-4. PCN for the *Grandfather* set of rules.

Because the *Parent* relation is an input place to more than one transition, we introduce two aliases for it: P_1 and P_2. The rest of the structure works in a similar way to that shown in Figure 3-3.

3.2.4 Storing PCNs in Relations

One of the main problems with HTRs is the possibility of repetition of predicates within the representation, if a predicate occurs more than once in a rule. PCNs do not have this repetition—each predicate is represented only once in the graph, but can have multiple connections with the other predicates in the graph. Thus the inefficiency of the PREDICATES table is eliminated. Arguments are not

explicitly represented in a PCN, so the problem of storing these efficiently also vanishes.

In Cheiney and de Maindreville [1989] PCNs are represented relationally using the three relations listed below (again, Δ indicates the primary key):

1) Relation MODULE—a list of rule modules:

Field Name	Domain	Description
MODULE_ID Δ	ID code	ID code of the rule module
MODULE_NAME	String	Name of the rule module

2) Relation TRANSITION—describes transitions and links them with rule modules:

Field Name	Domain	Description
MODULE_ID Δ	ID code	ID code of the rule module containing this transition
TRANS_ID Δ	ID code	ID code of the transition
TRANS_SCRIPT	String	The transition's inscription

3) Relation PLACE-TRANSITION-PLACE (PTP)—links places via transitions:

Field Name	Domain	Description
MODULE_ID Δ	ID code	ID code of the relevant rule module
PREPLACE Δ	String	Name of the relation that has input into the transition
TRANS_ID Δ	ID code	ID code of the transition that receives input from PREPLACE
INPUT_LABEL	String	The columns of PREPLACE which are input into the transition
OUTPUT_LABEL	String	The columns which are output to POSTPLACE
POSTPLACE Δ	String	Name of the relation that receives output from the transition

How efficient is this method of storing rules? Performing a similar analysis to that on the HTR relations, we get the following (again assuming *n* rules):

- For each rule, there is at most one entry in the MODULE relation. To be more precise, there is one entry in MODULE for each rule module. So at worst, the size of MODULE will be O(*n*), i.e. one rule per module.

- For each rule, there is one entry in the TRANSITION relation. Therefore, the size of TRANSITION is *n*.

- Each rule has several places (predicates). If there are *p* predicates on the LHS of a rule and *q* predicates on the RHS, then the number of entries in PTP for that rule will be at most $p \times q$. If we restrict ourselves to definite databases (i.e., $q = 1$), there will only be *p* entries per rule. Thus, the worst

case size of PTP is approximately $O(n^2)$ if we only allow definite data. Allowing indefinite data gives a worst case of approximately $O(n^3)$.

Thus it would appear that on average, the PCN method of storing rules relationally is more efficient in terms of storage than the HTR approach, even if we allow indefinite data.

An Example of Storing a PCN in Relations

The relational equivalent of the PCN in Figure 3-4 is as follows:

MODULE

MODULE_ID	MODULE_NAME
G1	Grandfather

TRANSITION

MODULE_ID	TRANS_ID	TRANS_SCRIPT
G1	T1	P1.Parent = M.Name
G1	T2	P2.Parent = F.Child

PTP

MODULE_ID	PREPLACE	TRANS_ID	INPUT_LABEL	OUTPUT_LABEL	POSTPLACE
G1	Male	T1	M.Name	<none>	Father
G1	Parent	T1	P1.Parent, P1.Child	P1.Parent, P1.Child	Father
G1	Parent	T2	P2.Parent, P2.Child	P2.Child	Grandfather
G1	Father	T2	F1.Father, F1.Child	F.Father	Grandfather

Contrast this with the example of storing HTRs relationally on page 34. Using PCNs, we have stored two rules in less space than was required to store *one* rule using HTRs.

3.3 Converting Rules into SQL Queries

It is possible to convert a rule directly into a statement in an appropriate database language (in this case SQL). In particular, it is possible to convert a query on a rule into an SQL SELECT statement. This procedure is best explained by an example. Suppose we have the following base tables in our database:

PARENT

PARENT	CHILD
Mary	John
Bill	John
John	William
John	Diana
Josephine	Diana

MALE

NAME
Bill
John
William

and the following rules:

$$Parent(x, y) \wedge Parent(y, z) \rightarrow Grandparent(x, z),$$
$$Grandparent(x, y) \wedge Male(x) \rightarrow Grandfather(x, y).$$

Thus we have the following virtual tables:

GRANDPARENT

GPARENT	GCHILD
Mary	William
Mary	Diana
Bill	William
Bill	Diana

GRANDFATHER

GFATHER	GCHILD
Bill	William
Bill	Diana

Now consider the query "List all grandfathers and their grandchildren." This can be formulated as the following SQL statement:

```
SELECT *
FROM   Grandfather;
```

Note that we are making a direct query upon a virtual table (rule), treating it as if it were just another base table. From the rule definitions, *Grandfather*(*x*, *z*) can be expanded to *Parent*(*x*, *y*) ∧ *Parent*(*y*, *z*) ∧ *Male*(*x*). From this we can derive the following SQL statement, which is directly equivalent to the statement above, but only involves base tables:

```
SELECT Parent.Parent, P2.Child
FROM   Parent, Parent P2, Male
WHERE  Parent.Child = P2.Parent
  AND  Parent.Parent = Name;
```

The SELECT clause is built by looking at the "attributes" of the virtual relation *Grandfather*, and finding the equivalent "attributes" on the LHS of the expanded *Grandfather* rule:

$$Parent(x, y) \wedge Parent(y, z) \wedge Male(x) \rightarrow Grandfather(x, z)$$

Thus attribute *GFather* of *Grandfather* maps to *Parent* in the first occurrence of the *Parent* relation (using *x* in the rule definition). Similarly, *GChild* maps to *Child* in the second occurrence of the *Parent* relation (using *z*). Note that *GFather* also maps to *Name* in the *Male* relation. The answer will be the same no matter which one we pick, so we take the first occurrence.

The FROM clause is built from the three tables in the rule: *Parent* (twice) and *Male*. We use an alias for the second occurrence of *Parent* to treat it as a "different" table; we are effectively rewriting the expanded rule as:

$$Parent(x, y) \wedge P2(y, z) \wedge Male(x) \rightarrow Grandfather(x, z)$$

where *P2* is the second occurrence of *Parent*.

The WHERE clause is built by taking each pair of predicates that have arguments in common, and joining over that common argument. Thus the sub-clause $Parent(x, y) \wedge P2(y, z)$ translates to the join condition *Parent.Child = P2.Parent* (joining over the *y* column, which is equivalent to the *Child* column in *Parent* and the *Parent* column in *P2*). Similarly, the sub-clause $Parent(x, y) \wedge Male(x)$ translates to the join condition *Parent.Parent = Name.*

Note that this process is very similar to the definition of a view. However, there is a subtle difference, in that a view remains static—only the data in a view changes, while the basic structure remains the same. The type of view defined here can vary in two ways: changing the contents of the base tables will change the contents of virtual tables which are defined by them, in the same way that the data in a view changes; altering a *rule* will change the *structure* of any virtual relations that the rule defines. Rules may also contain recursive information, whereas a conventional view cannot.

As a result, this type of view is more "dynamic" than a conventional view. Thus we could refer to conventional views as static views and rules as dynamic views. As has been stated in the literature [Gallaire *et al.* 1984], a dynamic view (rule) is just a more general form of a static (conventional) view, so it should come as no great surprise that they can be specified using similar means.

3.3.1 An Equivalence Between Relational Algebra and First-Order Logic

As support for the notion of dynamic views, we now define a simple equivalence between the first-order predicate calculus (usually used to define rules), and statements in the relational algebra. For the moment, we will only consider the case of Horn clauses.

Basically, we want to define equivalences between the operators of the two systems. One possible equivalence is shown in Table 3-1. This equivalence assumes a definite database under the CWA (thus the rather simple definition of negation).

Table 3-1. An equivalence between relational algebra and first-order logic.

Logical Operation	Equivalent Relational Operation
$A \wedge B$	$A \bowtie B$
$A \vee B$	$A \cup B$
$\neg A$	$\bigcup_{i=1}^{n} (R_i - A)$, for all relations R_i
$A \rightarrow B$	(none)

3.3.2 Extending the Equivalence to Handle Indefinite Data

The equivalence stated in the previous section can potentially be extended to handle indefinite data as well. An example should show the basic idea. Suppose our database contains the following rule:

$$cat(x) \rightarrow black(x) \vee white(x) \vee brown(x),$$

where *black*, *white* and *brown* are hybrid tables. Now consider the query "List all things that are black". We can find all things that are definitely black by querying the *black* table. To find all things which might be black, but which we are not sure about, we want to find all things which are cats and are not brown or white. This can be represented by the following formula:

$$cat(x) \cap [\neg brown(x) \cup \neg white(x)]$$

Now suppose we extend the rule base by adding the rule:

$$dog(x) \rightarrow black(x) \vee brown(x)$$

If we ask the same query again, we must also include all dogs that are not brown in the result. This can be represented by:

$$[cat(x) \cup dog(x)] \cap [\neg brown(x) \cup \neg white(x)]$$

The general case can be stated as follows. Given a rule base containing one or more rules of the form:

$$L_{i1} \wedge L_{i2} \wedge \ldots \wedge L_{ip} \rightarrow R_{i1} \vee R_{i2} \vee \ldots \vee R_{iq} \; (i=1,n),$$

the result of a query on table R_{ml} (l=1,q) may be expressed by the formula:

$$R_{ml} \cup \left[\left[\bigcup_{i=1}^{n} \bigcup_{j=1}^{p} L_{ij} \right] \cap \left[\bigcap_{i=1}^{n} \bigcap_{j \neq l} \neg R_{ij} \right] \right]$$

The first term of this formula is the *definite* term; the second term is the *indefinite* term.

The obvious problem with this formula is the presence of the "¬" (negation) operator. There is no equivalent relational operator, and negation in general causes major problems in a database. We cannot use the simple definition of negation described earlier, because it could cause inconsistencies in the data. The formalisms for indefinite data described in Section 2.6.1 may help this problem, but this is unclear at this point in time.

3.3.3 The SQLD_TO_SQL Translation Algorithm

The algorithm SQLD_TO_SQL processes dynamic views. From an initial SQLD statement, which may contain references to virtual tables (rules) and columns within those tables, the algorithm produces a "conventional" SQL statement. This SQL statement produces the same answer as the original query. The algorithm is stated below.

Algorithm SQLD_TO_SQL

Inputs:	F_{old}	the FROM clause of an SQLD statement
	S_{old}	the SELECT clause of an SQLD statement
	W_{old}	the WHERE clause of an SQLD statement
Outputs:	`Stmt`	an SQL statement
Local Vars:	F_{new}	the FROM clause of `Stmt`
	S_{new}	the SELECT clause of `Stmt`
	W_{new}	the WHERE clause of `Stmt`
	C_{temp}	temporarily stores WHERE conditions
	`E`	a list of expanded tables, including aliases and a pointer to a list of join conditions. The expanded tables contain the original rule's column names and the SQL column names for later reference.

Notes:
- Auxiliary routine `Alias`: given a base SQL table name, returns a unique alias for it. It is not included here.
- Auxiliary routine `JoinList`: given an expanded virtual table, returns a list of join conditions for the WHERE clause of the expanded statement (see below).

```
BEGIN (* Algorithm SQLD_TO_SQL *)

  (* Process FROM clause *)
  for each table reference Tj in Fold do
  begin
    if Tj is a rule reference (rule Rj) then
    begin
      expand Rj until there are no more rule
        references in Rj ⇒ expanded table Fj
      append Fj to Fold
      append (Tj + Tj's column names + Fj + JoinList(Fj)) to E
      delete Tj from Fold
    end
    else
    begin
      if Tj is in Fnew then append Tj + Alias(Tj) to Fnew
      else append Tj to Fnew
    end
  end
```

```
  (* Process SELECT clause *)
  for each column reference Ti.Ci in Sold do
  begin
    if Ci is empty then error (no column name)
    if Ti is empty then
    begin
      find the table Ti of which Ci is a column (search Fold)
      if no Ti found then error (column not in any FROM table)
      else if multiple Ti's found then
        error (ambiguous column name)
    end
    if Ti is a rule reference then
    begin
      find Ti in E ⇒ list of expanded tables Xi
      find first table in Xi which has Ci as a column
        ⇒ base table name (if no alias) or alias name Bi
        ⇒ SQL column name Qi
      replace Ti with Bi
      replace Ci with Qi
    end
    append Ti.Ci to Snew
  end

  (* Process WHERE clause *)
  for each join list Jk in E do append Jk to Wnew
  for each condition Wk in Wold do
  begin
    for each column reference Tk.Ck in Wk do
    begin
      if Ck is empty then error (no column name)
      if Tk is empty then
      begin
        find the table Tk of which Ck is a column (search Fold)
        if no Tk found then
          error (column not in any FROM table)
        else if multiple Tk's found then
          error (ambiguous column name)
      end
      if Tk is a rule reference then
      begin
        find Tk in E ⇒ list of expanded tables Xk
        find first table in Xk which has Ck as a column
          ⇒ base table name (if no alias) or alias name Bk
          ⇒ SQL column name Qk
        replace Tk with Bk
        replace Ck with Qk
      end
    end
    append Wk to Wnew
  end
  Stmt ← Snew + Fnew + Wnew
  return (Stmt)
END. (* Algorithm SQLD_TO_SQL *)
```

Auxiliary Routine JOINLIST

Inputs: `fromList` an expanded FROM clause

Outputs: `joinConds` a list of join conditions corresponding to fromList

```
BEGIN (* Algorithm JOINLIST *)
  for each table Tj in fromList do
    for each table Tl in fromList (l ≥ j + 1) do
      if Tj and Tl share a column Cj then
        append join condition (Tj.Cj = Tl.Cj) to joinConds
  return (joinConds)
END. (* Algorithm JOINLIST *)
```

There are some important points to note about the SQLD_TO_SQL algorithm:

- it does not handle the GROUP BY, HAVING or ORDER BY clauses of an SQL statement, but including these is not difficult.
- it only processes SELECT statements, but again, it is not difficult to extend this to handle the other SQL statements that might refer to virtual tables (DELETE, UPDATE and INSERT in particular).
- it does not consider the case of aliases for virtual tables. The main result of this is that it is impossible to join a virtual table with itself.
- it does not consider SELECT statements nested in the WHERE clause.

3.4 Extensions to SQL: The SQUALID Query Language

SQL is a very versatile and powerful query language, but from a deductive point of view it is woefully deficient, having no capacity to create and/or manipulate rules. For this purpose a superset of SQL has been developed, called SQLD (for **SQL D**eductive). This section describes the extensions that have been made to standard SQL statements to handle rules. (For those who have not encountered SQL before, Appendix B contains a brief overview of the language.)

Table 3-2. SQL statements that directly affect or use rules.

ALTER TABLE	INSERT
CREATE TABLE	REVOKE[3]
DELETE	SELECT
DROP TABLE	UPDATE
GRANT	

[3]The REVOKE statement is not part of ANSI standard SQL. It is specific to VAX SQL.

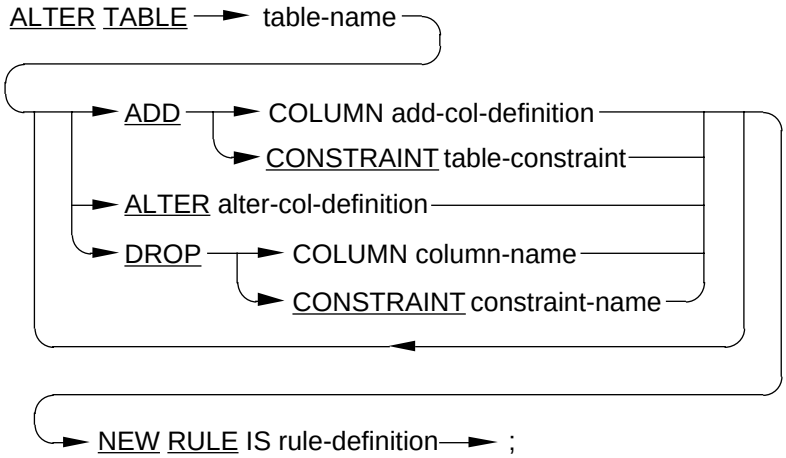
The number of SQL statements that may directly affect or refer to rules is not large; these statements are listed in Table 3-2. There are also some statements that may indirectly involve rules, for example the CREATE SCHEMA statement, which may include zero or more CREATE TABLE statements. These are not really affected, however.

The following sections describe the differences between the SQL and the SQLD versions of the statements listed in Table 3-2. The syntax diagrams use the same notation as that used in the VAX SQL documentation [DEC 1989b].

3.4.1 Data Definition Statements

ALTER TABLE

The ALTER TABLE statement has been changed to allow changes to virtual tables. The syntax of the new statement is:



The statement has been modified by adding the optional NEW RULE IS clause. This clause is only used for virtual tables; it is ignored if used with a base table. This clause must be included if the table is a virtual table. If it is not included, an error will result.

CREATE TABLE

The CREATE TABLE statement has been extended to allow the definition of virtual tables. The syntax of the modified CREATE TABLE statement is shown in Figure 3-5. The statement has been modified by adding the optional FROM RULE clause. In all other respects, the statement has not changed. The FROM RULE clause defines the how the virtual table is constructed from the underlying base tables.

One interesting feature of the modified CREATE TABLE statement is that when creating virtual tables, it actually creates a physical table in the database. This table is used as a placeholder for such things as table comments, indexes and access privileges; that is, those things that really need a physical table in the database to work effectively. This argument is not entirely valid however, as the

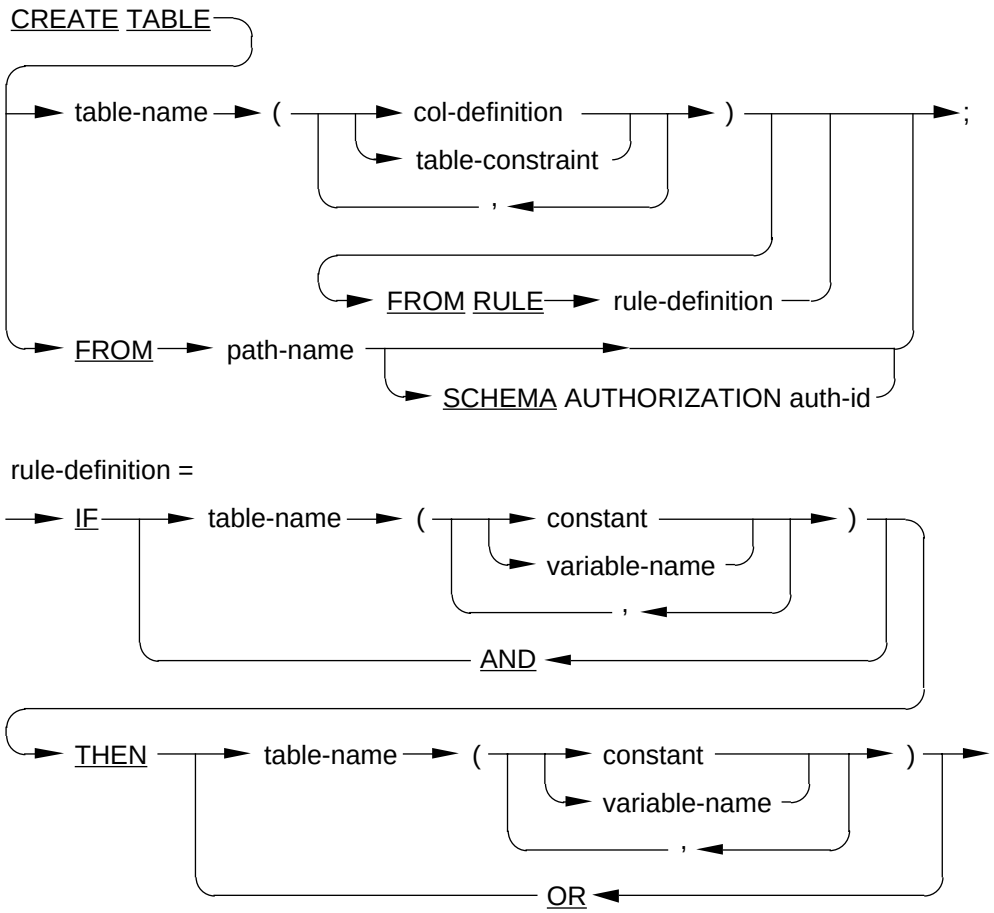


Figure 3-5. Syntax of the CREATE TABLE statement.

system could simply expand the virtual table reference as it does for the SELECT statement, and apply the appropriate operation to the base tables making up the rule definition. Two better arguments for the former approach are:

- it reduces the amount of processing required to perform some operations; and

- it removes any conflicts that might occur if the underlying base tables are changed. For example, suppose we have two virtual tables which share base tables in their respective rule definitions. For example, in Figure 3-6 we have two virtual tables, *Father* and *Grandfather*, which share the underlying base table *Parent*. Now suppose that for some reason we want to deny any public access to the *GFather* column of *Grandfather*, but allow public read-only access to the *Father* column of *Father*. If access privileges for rules are stored in the underlying base tables, we get a conflict of interests, as shown in Figure 3-6.

This does not mean that the access privileges in the underlying tables should be ignored. Indeed, the way SQUALID has been built, they *cannot* be ignored. If a

user makes an attempt to retrieve data from an underlying table, and the user does not have access to that table (or the relevant columns), the attempt will fail, regardless of whether the user has access to the columns in the virtual table or not.

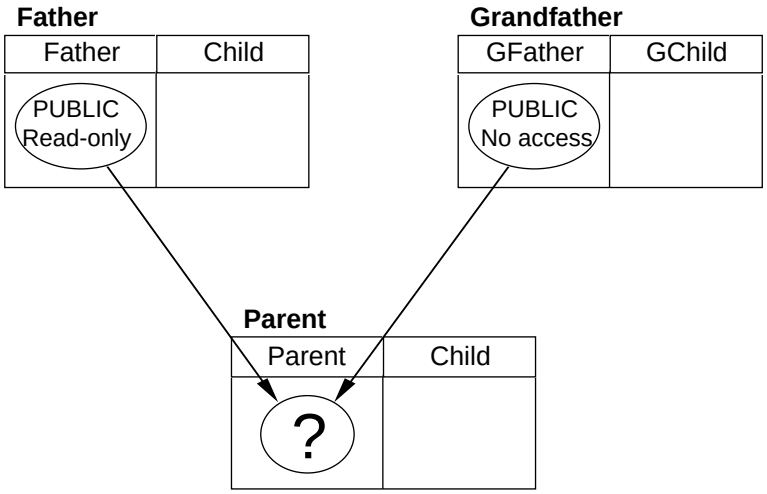
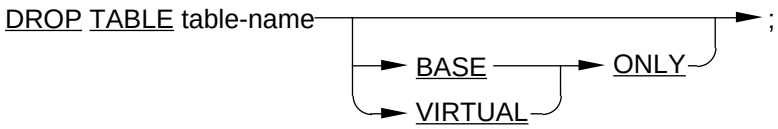


Figure 3-6. Why a base table is necessary.

One potentially useful side effect of this approach is that it allows us to insert data directly into the physical table, that is, all virtual tables in SQUALID are actually hybrid tables (we will continue to call them virtual tables to save confusion). We would need this capability if we had some data which we knew to be true, but which could not be derived from the existing data and rules. In this case we would have to store the data explicitly. An example of this is not immediately obvious, but it is conceivable that such a situation could arise. One point to note, however, is that this approach is potentially dangerous: useful data can be lost when the table is dropped.

DROP TABLE

This statement has been changed to handle virtual tables. The syntax of the modified statement is:



The statement has been modified by adding the optional BASE ONLY and VIRTUAL ONLY clauses. These clauses only affect virtual tables; they are ignored if applied to base tables. These clauses control which parts of the virtual table are dropped; BASE ONLY drops the base component only, whereas VIRTUAL ONLY drops the virtual component only.

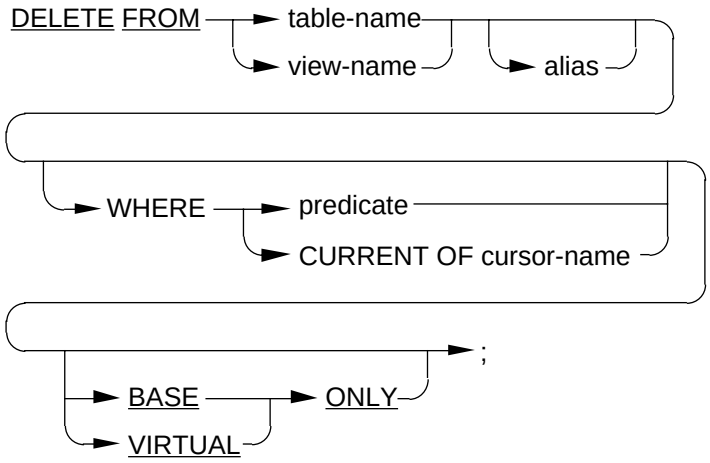
Using BASE ONLY can be dangerous; data stored in the base component of the table may not be derivable from the data underlying the virtual component. If the base component of the table is dropped, this data will be lost.

The most important thing to remember when dropping any table is to check whether it is involved in any rule definitions. If this is the case, the table cannot be dropped. This is a generalisation of the restriction which prevents us from dropping tables which are part of a view definition.

3.4.2 Data Manipulation Statements

DELETE

The DELETE statement has been modified to deal with virtual tables. The syntax of the modified statement is:



The statement has been modified by adding the optional BASE ONLY and VIRTUAL clauses. These clauses are similar to those in the DROP TABLE statement (see above). The default for a deletion on a virtual table is to apply it to both components of the table. BASE ONLY restricts the deletion to the base component, VIRTUAL ONLY restricts it to the virtual component.

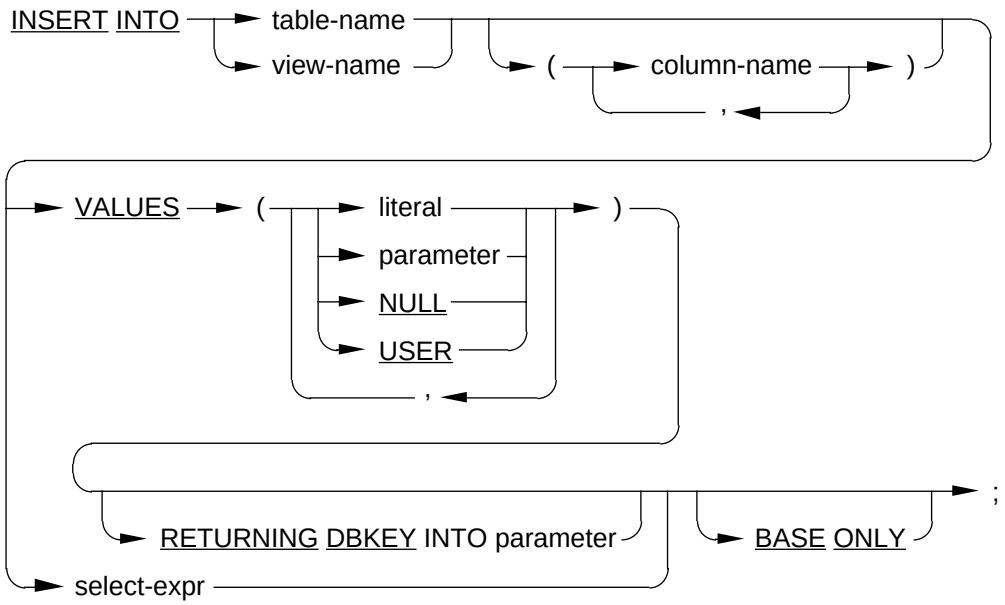
Deletions on virtual tables are interesting, in that they can result in multiple deletions over several tables. For example, given the rule:

$$\mathit{Mother}(x, y) \wedge \mathit{Parent}(y, z) \rightarrow \mathit{Grandmother}(x, z),$$

where *Mother* and *Parent* are base tables, a DELETE on *Grandmother* will be converted into three DELETEs, one on *Mother*, one on *Parent* and one on the base component of *Grandmother*. If BASE ONLY is specified, there will only be one deletion on the base component of *Grandmother*. Note however, that it may not be possible to apply this approach in all cases (see the description of INSERT below).

INSERT

The INSERT statement has been changed to handle insertions into virtual tables. The syntax of this statement is:



The optional BASE ONLY clause of the statement is only used for virtual tables; it has no effect on insertions into base tables.

The default for insertion into a virtual table is for the base tables underlying it to be updated. If BASE ONLY is specified, the base tables are not updated; instead the data is inserted directly into the base component of the virtual table.

Note that an insertion into a rule will generate multiple insertions over several tables (as for DELETE), and may not be possible in all cases. For example, consider an insertion into the *Grandparent* rule. We know who the grandparent and grandchild are, but who is the parent in the middle? There are two possible solutions to this problem:

1. Insert nulls for the unknown data. This fails if the column we are inserting into is part of a primary key, as in the example above;

2. Insert some sort of placeholder value. This similar in principle to Prolog's ANONYMOUS VARIABLE concept.

This problem is an extension of the view updating problem ("updating" includes DELETE, INSERT and UPDATE). It is a well-known fact that some views are theoretically updatable, while others are not [Date 1990]. If a single-table view includes the primary key of the underlying table, the view is theoretically updatable, otherwise it is not. Views based on joins cause further problems, and general rules, with the possibility of recursion, make it very difficult to deter-

mine whether a given view is updatable or not. The ad hoc manner in which many implementations deal with this problem only serves to make matters worse.

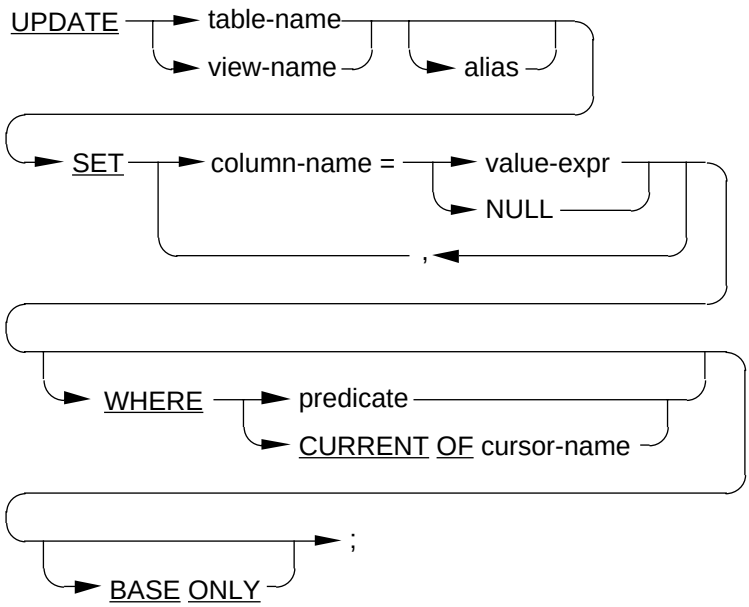
SELECT

The SELECT statement has not been changed. All necessary processing for a SELECT statement is carried out by the SQLD_TO_SQL algorithm described earlier. Virtual tables with a base component can be handled by making two queries, one on the base tables underlying the virtual component and one on the base component.

In the future, it may be desirable (or necessary) to modify this statement. Possible extensions are BASE ONLY and VIRTUAL ONLY clauses (as for DELETE and DROP TABLE), to restrict queries to particular components of a virtual table.

UPDATE

This statement has been modified to handle virtual tables. The syntax of the modified statement is:



This statement has been modified by adding the BASE ONLY clause. This clause has an identical purpose to that used in the INSERT statement. The way the update is carried out is similar to the way that an insertion works, that is, multiple updates are carried out over the underlying tables (if possible). The BASE ONLY clause restricts the update to the base component of the virtual table.

3.4.3 Security Statements

There are two other statements that are affected by the introduction of virtual tables: the GRANT and REVOKE statements. These are database security statements; they are used to grant and revoke access privileges on schemas, tables,

columns, and so on. They have not been changed directly, but because of changes to the CREATE TABLE statement, can now refer to virtual tables. This is possible because the CREATE TABLE statement actually creates a physical table in the database—the access privilege is placed on/removed from this table instead of the virtual table, which effectively does not exist.

3.5 Summary

In this chapter we have discussed the issues that had to be addressed during the implementation of SQUALID: how to physically store rules, how queries on virtual tables should be handled and how SQL could be extended to handle virtual tables.

Rules should ideally be stored in the database along with the "conventional" data, so that the advantages of using a relational DBMS can be applied to them also. We have discussed two possible methods of doing this, the HTR structure and the PCN structure. Of these, PCNs seem best suited, because of their less restrictive construction and more efficient storage.

Rules are a generalisation of conventional relational views. Therefore, it should be possible to treat rules in a similar manner to conventional views. We have developed a method of generating view definitions from SQL queries on virtual tables. This is different from a conventional view in that the view definition is not determined until query time; conventional views are static and do not usually change between one query and the next. From this we define the terms static view and dynamic view, which refer to conventional views and rules respectively.

Finally we have proposed possible extensions to VAX SQL for handling virtual tables and discussed some of the implications of these extensions.

Chapter 4

Integrated Deductive Databases

As stated in Chapter 2, an integrated deductive database is one which consists of two separate components, a logical inference unit (LIU) and a relational database management system (RDBMS). These components are integrated in some way to produce a deductive database system. This is relevant here, as SQUALID is itself an integrated system. This chapter presents an abstract model for integrated deductive databases [Bell *et al.* 1990], and the different methods of implementing them.

4.1 An Abstract Model for Integrated DDBs

A deductive database must have both a LIU and a RDBMS—the LIU is needed to perform deductions on the data, and the RDBMS is required to handle the large amounts of data that will be encountered. Neither of these components on their own can perform the actions of the other. Integrating the two components is feasible—both logic programming and relational data retrieval theory are based on first-order logic, and relational data retrieval can in fact be considered a special case of deduction (cf. views being a special case of rules). This is indicated by the fact that a conventional RDBMS can be fully implemented in a logic programming language such as Prolog (it would not be particularly efficient of course, but that is irrelevant in this context).

An integrated system can be considered at three levels: the logical level, the function level and the physical level. Each level represents a coupling between certain aspects of the LIU and the RDBMS. Figure 4-1 shows these levels and how they relate to each other. Further aspects of the diagram will be explained shortly. Each of the three levels corresponds, respectively, to one of the following questions:

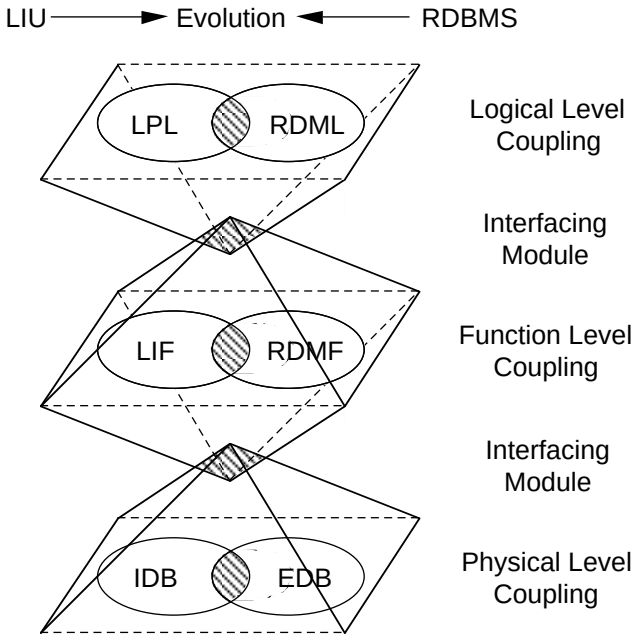
- How can we implement a query language which supports both logical deduction and relational data manipulation?
- How should we implement logical deduction against the underlying database?
- How should we represent the explicit data and deductive information in the underlying database?

The logical level corresponds to the user interface of the system. The problem here is to integrate the useful features of both a logic programming language

(LPL) and a relational data manipulation language (RDML). That is, the combined language should be able to deal with both logical deduction and with relational data manipulation in a useful and usable fashion.

The next level is the function level. This level links the *functionalities* of the LIU and the RDBMS, hence the name. The job of this level is to receive requests from the logical level, and determine which combination(s) of logical inference functions (LIFs) and relational database management functions (RDMFs) to use to retrieve the data in an efficient manner.

The physical level deals with the actual storage of data—how should explicit data and deductive information be stored? Should they be stored together in a uniform fashion, or should they be kept separate? The physical part of a deductive database generally consists of two parts: the extensional database (EDB), which contains all the explicit facts; and the intensional database (IDB) which stores the deductive information.



LIU	Logical Inference Unit
RDBMS	Relational Database Management System
LPL	Logic Programming Language
RDML	Relational Database Manipulation Language
LIF	Logical Inference Function
RDMF	Relational Database Management Function
IDB	Intensional Database
EDB	Extensional Database

Figure 4-1. An abstract model of integrated deductive databases (from Bell *et al.* [1990]).

Because of the integration process, there may be some overlap between components, as shown in Figure 4-1. These overlaps, although drawn as such, are not

conventional set intersections. Rather, they represent the extent to which the two components are *coupled*. The extent of coupling between each pair of components can be defined by their intersection, that is:

$$\text{LPL} \cap \text{RDML}$$
$$\text{LIF} \cap \text{RDMF}$$
$$\text{IDB} \cap \text{EDB}$$

If this intersection is empty (i.e. the components are independent of each other), then the two components are said to be *loosely coupled*, otherwise they are *tightly coupled*. Note that a system can have different types of coupling at different levels.

The query language, system functions and underlying database of an integrated system are defined, respectively, as the union of each pair, that is:

$$\text{LPL} \cup \text{RDML}$$
$$\text{LIF} \cup \text{RDMF}$$
$$\text{IDB} \cup \text{EDB}$$

Note that in the tightly coupled case, one component may dominate the other, giving the appearance of orientation towards the dominating component.

4.2 Logical Level Coupling

A language which combines the power of relational data manipulation and logical deduction is essential for a deductive database. RDMLs, although complete, lack certain facilities like recursion[4], which is necessary for logical deduction. By contrast, LPLs have the problem that they are *tuple-oriented*, while relational languages are generally *set-oriented*. This effectively means that a relational language specifies *what* to retrieve, while a logic programming language specifies *how* to retrieve it.

A loose coupling at the logical level is implemented by creating an interface between a LPL and a RDML, for example, between Prolog and SQL. Users might be able to access SQL from Prolog, and vice versa. This allows SQL to use the deductive power of Prolog, while Prolog can exploit the set-oriented nature of SQL. The main problem with this is efficiency—this is entirely dependent on the quality of the interface between the two languages. Also, users must learn two completely different languages to use the system effectively.

A tightly coupled approach involves creating a single language which contains the features required for both deduction and data manipulation. This language can be oriented towards either data manipulation or deduction. For example, if we embedded a language like SQL into Prolog syntax, we would have a strong orientation towards deduction. The opposite approach, extending a

[4]Date notes [Date 1990] that there is no reason in principle why recursive operators could not be added to classical relational algebra, and cites an example of such [Agrawal 1988].

relational language with deductive capabilities, has the advantage that these facilities are added in a way that many existing database users will understand.

4.3 Function Level Coupling

The function level coupling is one of the most important aspects of the implementation of an integrated deductive database—the way in which the LIFs and the RDMFs are coupled will have a major effect on the functionality and efficiency of the system.

Because of the different orientations of the two approaches (logic programming and relational databases) LIFs and RDMFs process queries in different ways. As already stated, LIFs process one tuple at a time, whereas RDMFs are set-oriented. In practice, this means that a RDMF will often only have to access the database once, whereas an LIF will usually have to access the database many times. For example, suppose we define the following rule in Prolog:

```
a(X, Y) :- X > Y.
```

and suppose that there are 1000 tuples in relation `a` that satisfy the condition `X > Y`. Processing the query

```
?- a(X, Y).
```

will require 1001 accesses to the underlying database: 1000 to retrieve the tuples, plus one to find that there are no more tuples satisfying the condition. The equivalent SQL query is:

```
SELECT X, Y
FROM a
WHERE X > Y
```

Using RDMFs, processing this query would require only one access to the underlying database—all tuples that satisfy the condition are returned to the user as a set. This is much more efficient than the LIF approach.

However, we cannot just merge existing LIFs and RDMFs, because there is some overlap between the two groups. For example, views are a special case of general rules. Also, LIFs and RDMFs as they are still cannot solve all problems. For example, optimisation of non-recursive queries is not particularly hard, but optimising recursive queries is considerably more difficult. It cannot be handled by RDMFs, as they do not currently deal with recursion. LIFs cannot do it either, as they often perform almost no optimisation. As noted in Chapter 2, this is an active area of research.

A loose coupling at the function level is implemented in much the same way as at the logical level—the two sets of functions remain distinct, with some sort of interpreter linking them. This has the advantage that the operation of the system is independent of any particular set of LIFs or RDMFs. Any LIF-RDMF pair may be coupled by modifying the interface module between them. However, as

stated above, this introduces function redundancy between the two, and separating query processing into deduction and retrieval functions is not easy.

The tightly coupled approach merges the two groups of functions into a fully integrated module. As a result, interaction between these functions may take place at any time, rather than during set intervals. As for the logical level, the coupling can be oriented towards either the LPL or the RDML. This approach avoids the problems of function redundancy mentioned above, but it seems to gain only the power of a LIU, or of a RDBMS, depending on the orientation of the coupling. How to balance the two orientations is still an open question.

4.4 Physical Level Coupling

Compared with the other two levels, the coupling at the physical level is fairly straightforward. The important questions here are how the IDB and EDB should be organised in relation to each other, and how they can interact so that data manipulation and deduction are as efficient as possible.

The loosely coupled approach is fairly simple to implement, and most existing integrated systems seem to use this approach. The EDB and IDB exist as separate entities, and interact only when required.

The tightly coupled approach stores the EDB and the IDB together as a single entity. The classic example of this approach is Prolog, where base facts and rules can be represented using a uniform notation. Of course, the major disadvantage with Prolog appears when the database becomes very large—program execution becomes very inefficient. Another possibility is to store the IDB in a relational database, in the same way that the EDB is stored. An example of this is the relational equivalent of the PCN representation of rules described in Chapter 3.

4.5 Summary

In this chapter we have presented an abstract model for integrated deductive databases [Bell *et al.* 1990]. It proposes that the linkage between the two components of an integrated system can be broken down into three levels of coupling: the logical level, the function level and the physical level. Each of these levels corresponds to a particular part of the integrated system. Using this model, we may analyse the extent to which the two components of an integrated deductive database are coupled together. We will apply this analysis to SQUALID in Chapter 6.

Chapter 5

Implementation

5.1 Introduction

This chapter covers some of the more important aspects of the implementation of SQUALID. The system itself was implemented in four phases. These phases were:

1. A deductive module was designed and implemented. The deduction method used was a simple form of SLD-resolution (based on Prolog), and the rules were stored using the hierarchical tree representation (HTR). This module is discussed in Section 5.2.

2. A group of modules was developed for interfacing with VAX SQL. These are described in Section 5.3.

3. The modules listed in 1 and 2 were integrated to produce the initial version of SQUALID, and the basic structure of the SQLD preprocessor was developed. This phase is described in Section 5.4.

4. The basic system was extended in various ways, including rewriting the rule storage method to use a modified form of the production compilation network (PCN) instead of HTRs. These extensions, and the current state of SQUALID, are described in Section 5.5.

Section 5.6 discusses some technical aspects of some of the data structures that were developed. Section 5.7 summarises the chapter.

5.2 The Deduction Module

The initial intention with the deduction module was to write something akin to a Prolog interpreter which could make simple queries to a rule base. It was not intended to be a full-blown logic programming system, as this was beyond the scope of the project. All that was really needed was a simple deduction system that could accept information sent to it from the SQLD preprocessor, evaluate that information and send back the appropriate results.

One obvious question is why Prolog was not used for the deduction module in the first place, to avoid “reinventing the wheel,” as it were. The reason for this was that the versions of Prolog that were available did not have useful I/O predicates nor any real means of interfacing with other programming languages. It would also have been very difficult to interface Prolog with SQL. It was de

cided that it would be simpler to write a separate dedicated system to handle deductive processing. A final point to note on this subject is that that deductive module is independent from the rest of the system, so it should not be too much of a problem to change the deduction module in the future. All that will really need to be changed is the interface between this module and the rest of the system.

The algorithms that SQUALID uses are based on the methods that Prolog uses. These algorithms will be described shortly, but first, a brief introduction to the concept of matching with respect to Prolog is in order.

5.2.1 A Brief Description of Matching in Prolog

From a syntactic point of view, all data objects in Prolog are *terms*. A term itself may be any one of the following:

- an atom, e.g. fred, “Tom”, mount_Athabasca.
- a number, e.g. 42, -97, 0.23.
- a variable, e.g. X, _zero, Variable_name (note that all variables in Prolog start with either an upper-case letter or an underscore character).
- a structure, e.g. father(fred, bert); this is equivalent to a predicate in logic. Structures are distinguished by their *functor* (*father* in the example) and their *arity* (the number of components in the structure, which in this case is two).

Note that Prolog terms are not the same as first-order logic terms, which were defined in Section 2.2. In fact, they incorporate both logical terms and atomic formulae.

The most important operation that is performed on Prolog terms is that of *matching*. This process takes two terms as input and checks to see if they match. If they do not match, the matching process fails; if they do match, the process succeeds and the variables in both terms are instantiated so that the terms become identical (i.e. unified).

In general, two terms *S* and *T* match if any of the following conditions apply [Bratko 1986]:

1. If *S* and *T* are constants, then *S* and *T* match only if *S* and *T* are the same object.
2. If *S* is a variable and *T* is anything, they successfully match, and *S* is instantiated to *T*. If *S* is a not variable and *T* is a variable, then *T* is instantiated to *S*.
3. If *S* and *T* are structures, they match only if
   (a) *S* and *T* have the same principal functor (predicate name), and

(b) all their corresponding components match.

The resulting instantiation is determined by the matching of the components.

5.2.2 The Deduction Algorithms

The deduction algorithms used by SQUALID are listed below. The DEDUCE algorithm is based on a procedure called *execute*, described by Bratko [1986]. It takes a list of rules *R* and a list of goals *G*, and attempts to prove *G* from *R*. The MATCH algorithm is called from DEDUCE. It takes two terms and attempts to match them, using the matching rules described in the previous section.

Algorithm DEDUCE

Inputs:	`RuleList`	a list of rules
	`GoalList`	a list of goals
	`Success`	a boolean value
Outputs:	`Success`	**true** if the goals in `GoalList` succeed, **false** otherwise
Local vars:	`Goal, R, L1, ..., Ln`	individual goals
	`OtherGoals, NewGoals`	lists of goals
	`Satisfied, MatchOK`	booleans
	`Instant`	a variable instantiation

Side Effects: None

```
BEGIN
  if GoalList is empty then Success ← true (* no more goals *)
  else
  begin
    Goal ← first element of GoalList
    OtherGoals ← rest of GoalList
    Satisfied ← false
    while (not Satisfied) and (RuleList not empty) do
    begin
      take a rule L1 ∧ L2 ∧ … ∧ Ln → R in RuleList
      Match(Goal, R, MatchOK, Instant)
      if MatchOK then
      begin
        append [L1, L2, ..., Ln] to OtherGoals giving NewGoals
        substitute Instant into NewGoals
        Deduce(RuleList, NewGoals, Satisfied)
      end
    end
    Success ← Satisfied
  end
  return (Success)
END. (* Algorithm DEDUCE *)
```

Algorithm MATCH

Inputs:	$Term_1$, $Term_2$	terms
	Matched	boolean
	Instant	a variable instantiation
Outputs:	Matched	**true** if the rules match, **false** otherwise
	Instant	a variable instantiation
Local vars:	MatchOK	boolean
	TempInst	a variable instantiation

Side Effects: None

```
BEGIN
  if Term1 and Term2 are identical then Matched ← true
  else if Term1 and Term2 are predicates then
  begin
    if names and arities of Term1 and Term2 are identical then
    begin
      MatchOK ← true
      while there are components C1, C2 of Term1, Term2
            (respectively) and MatchOK do
      begin
        Match(C1, C2, MatchOK, TempInst)
        if MatchOK then append TempInst to Instant
      end
    end
    else MatchOk ← false (* they don't match *)
    Matched ← MatchOK
  end
  else
  begin
    if Term1 and Term2 are both constants and
        are identical then
      Matched ← true
    else if Term1 is a variable then
    begin
      Term1 is instantiated to Term2 (append to Instant)
      Matched ← true
    end
    else if Term2 is a variable then
    begin
      Term2 is instantiated to Term1 (append to Instant)
      Matched ← true
    end
    else Matched ← false
  end
  return (Matched, Instant)
END. (* Algorithm MATCH *)
```

5.2.3 Rule Storage in the Deduction Module

The rule base in the original deduction module was stored in main memory as a linked list of HTR-like structures. Despite its shortcomings, the HTR structure seems well suited to the algorithms described above. Further technical aspects of this structure are discussed in Section 5.6.1.

The rules themselves were stored in a text file, and were read into memory by a YACC-generated parser when the associated Rdb database was opened (the rules file had the same name as the database file, but with the extension .RLS). The rules in the file were represented using an extended form of first-order logic notation. A typical rule in the rules file looked something like this:

```
PARENT(_X:PARENT, _Y:CHILD) ^ MALE(_X:NAME) -> FATHER(_X:FATHER, _Y:CHILD)
```

The first-order logic equivalent of this rule is:

$$Parent(x, y) \wedge Male(x) \rightarrow Father(x, y)$$

Conjunction, disjunction, negation and implication are represented by the symbols `^`, `|`, `~` and `->` respectively. Variables are prefixed with an underscore (e.g. _X). SQL column names are included using the `:<column-name>` construct; for example, `MALE(_X:NAME)` means that this relation has one column, called NAME. Any line beginning with an exclamation mark (`!`) is a comment.

This notation is cumbersome, highly repetitive and very easy to get wrong. It is also rather difficult to modify the rule base, due to it being stored in a text file. The notation was only intended as an interim measure until a more effective method of storing rules could be found. This objective has been achieved—rules are now stored relationally, and the rules file has become defunct. Section 5.5 discusses these changes in more detail.

5.3 The SQL Interface

This part of the system is built around the dynamic SQL processor of VAX SQL. VAX SQL offers three distinct interfaces:

- Interactive SQL—this is how many end-users interact with Rdb databases. It is invoked from the command line by the `sql` command, and users issue SQL statements directly to the database.

- Embedded SQL—this takes the form of SQL statements embedded in source files of some host language. The languages currently supported by this facility are Ada, C, COBOL, FORTRAN, Pascal and PL/I. Embedded SQL programs are processed by a precompiler which converts the SQL statements into host language statements. The resulting source file is then compiled and linked normally.

- SQL Module Language—this is a “programming language” version of SQL, although it comes nowhere near the scope and flexibility of a

language such as Pascal. Programmers can create a module of SQL procedures, which is compiled and linked with modules written in some other language(s) to form an executable image. The SQL procedures are called as external procedures from the other language.

Dynamic SQL can only be used from embedded SQL or SQL modules. Dynamic SQL lets programs accept or generate arbitrary SQL statements at run time. The usual approach is to include SQL statements as part of the source code or as SQL module language procedures. These statements are always processed at compile time, and cannot be changed without changing and recompiling the program. Dynamic SQL was absolutely essential to the working of SQUALID, as you cannot predict the kinds of SQL statements that a user may enter.

The SQL interface was implemented as two modules: DYNAMIC.PAS, a Pascal module, and SQLDYN.SQLMOD, a SQL module. Embedded SQL was not used because the Pascal precompiler was not available at the time the modules were written.

The SQL module contains several SQL procedures which handle the following: preparing SQL statements for execution; declaring, opening and closing cursors for use with SELECT statements; fetching rows from a cursor; commiting and rolling back transactions; and executing SQL statements. Procedures in this module are called from DYNAMIC.PAS.

The Pascal module prepares the SQL statement for execution and then executes it by calling the appropriate SQL procedures. If the statement is a SELECT statement, a cursor is opened and rows FETCHed from it. These rows are then displayed on the screen by a Pascal routine.

5.4 Integrating the Pieces

Now that the two main components of SQUALID were written, it only remained to link them together. It was always the intention that the system would take an SQL statement with embedded rule references, perform deduction to remove these references and send the final result (via dynamic SQL) to VAX SQL for processing. It was from this notion that the SQLD_TO_SQL algorithm described in Section 3.3.3 was developed.

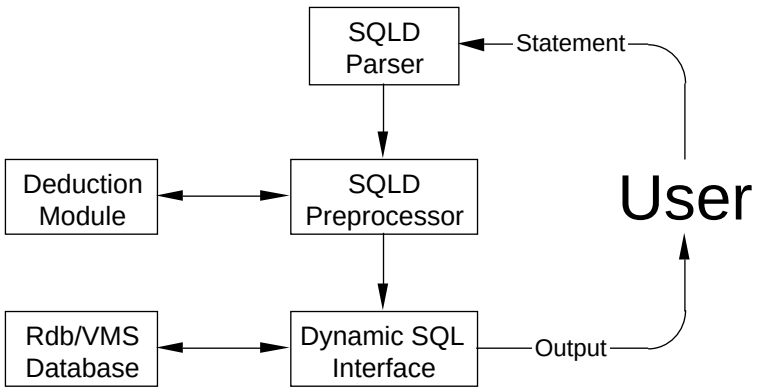


Figure 5-1. Structure of SQUALID.

The overall structure of SQUALID is shown in Figure 5-1. Incoming statements are first parsed for syntax errors (the SQLD parser was also generated using YACC). If the statement is valid, it is passed to the SQLD preprocessor, which in conjunction with the deduction module, expands any rule references in the statement. The preprocessed statement is then passed to the dynamic SQL module for processing against the database. The results of the statement are returned to the user (if applicable).

Now let us consider how a typical statement is processed by SQUALID. The database used is the one described in Section 3.3. Suppose a user enters the following statement:

```
SELECT *
FROM GRANDPARENT;
```

This statement is executed as follows:

1. The statement is parsed by the SQLD parser. As this happens, an internal representation of the statement is built in main memory.

2. The representation of the query built by the parser is sent to the SQLD preprocessor. This is then scanned for any rule references. In this example, there is only one rule reference: *Grandparent*. This is passed to the deduction module, which expands this rule until there are no further rule references in the expanded rule (as described in Section 3.3).

3. Once deduction is completed, the expanded rule is returned to the preprocessor, which then rewrites the statement using the SQLD_TO_SQL algorithm. When this is finished, the internal representation is converted back into an SQL statement in text form. Continuing our example, the converted SQL statement is:

```
SELECT PARENT.PARENT, P2.CHILD
FROM   PARENT, PARENT P2, MALE
WHERE  PARENT.CHILD = P2.PARENT
  AND  PARENT.PARENT = MALE.NAME;
```

4. The processed SQL statement is passed to the dynamic SQL interface. This opens a link to the underlying Rdb database and readies the expanded statement (via dynamic SQL) to be executed. In this example, the results of the statement are returned to the program row by row and displayed on the user's screen.

5.5 Extensions and the Current State

This section describes the extensions and modifications that have been made to SQUALID since the first version was completed. At the same time, it describes the current state of the system, and what it is capable of.

5.5.1 Implementation of Various SQL Statements

The following SQL statements have been implemented in some form or other:

SELECT
CREATE TABLE

There were many other SQL statements which could have been implemented, but these two are sufficient to demonstrate the underlying principles of SQUALID.

The SELECT statement is arguably the most used SQL statement, so it was very important to implement this statement. The other data manipulation statements (DELETE, INSERT and UPDATE) are very similar in structure to the SELECT statement. Thus the SELECT statement can be considered representative of all the data manipulation statements. Similarly, the CREATE TABLE statement is representative of all the data definition statements. For this reason it was not absolutely essential that the other statements be implemented.

5.5.2 Rule Storage Using PCNs

Rules in SQUALID are no longer stored permanently in main memory using the HTR representation. Instead, they are stored in the database using a modified subset of the PCN representation described in Section 3.2.3. Rule modules have not been implemented because they do not really fit into the current structure of SQUALID; they are better suited to a system that is oriented more towards logic programming.

Rules are stored in two relations, TRANSITION and PTP, which have the following structures:

1) Relation TRANSITION—describes transitions (i.e. the structure of a rule):

Field	Type	Description
TRANS_ID Δ	ID code	ID code of the transition
TRANS_SCRIPT	String	The transition's inscription

2) Relation PTP—links predicates on left- and right-hand sides of rules, and transitions:

Field	Type	Description
TRANS_ID Δ	ID Code	ID of the transition for this rule
PREPLACE Δ	String	Name of a predicate on the LHS of the rule
PREPOSITION Δ	Integer	Position of the predicate in the rule
PRECOLUMNS	Integer	Number of columns in PREPLACE
POSTPLACE Δ	String	Name of a predicate on the RHS of the rule

The internal structure of the rule is stored in TRANSITION, while the PTP relation stores the information about the relations involved in the rule. For example, this is how the rule $Parent(x, y) \wedge Parent(y, z) \rightarrow Grandparent(x, z)$ would be stored:

TRANSITION

TRANS_ID	TRANS_SCRIPT
PPG14-DEC-1990 16:45:32.13	1.1=-1.1,1.2=2.1,2.2=-1.2,

PTP

TRANS_ID	PREPLACE	PREPOSITION	PRECOLUMNS	POSTPLACE
PPG14-DEC-1990 16:45:32.13	Parent	1	2	Grandparent
PPG14-DEC-1990 16:45:32.13	Parent	2	2	Grandparent

The transition inscription in the TRANSITION table can be interpreted as follows. Each "=" condition in the inscription defines a set of two or more columns that share the same variable. The number to the left of the period refers to the position of a relation on the LHS of the rule, unless the number is negative, in which case the relation is on the RHS; the number to the right of the period refers to the position of a column within that relation. Thus, the condition "1.1=-1.1" indicates that the first column of the first relation on the LHS of the rule and the first column of the first relation on the RHS of the rule share the same variable. This represents the variable *x* in the *Grandparent* rule above. Multiple conditions are separated by commas.

Whenever there is a reference to a virtual table, all rows which have that table as a POSTPLACE are retrieved from PTP. The transition inscription for the rule is then retrieved from TRANSITION (there is an implicit assumption here that virtual tables are defined by only one rule), and this, along with the information from PTP, is used to build a representation of the rule in memory. The old HTR structure is still used to store rules in memory, so only minor changes were required to the deduction module. The algorithms used to convert between the two storage representations are listed below.

The end result of all this is that rules now only have to be read into memory when they are needed, rather than being stored in memory all the time. It also makes the resulting deductive database more portable, as the rules and data are stored in the same file, rather than separately, as they were earlier.

Algorithm LOAD_RULE

Inputs: `ruleName` name of rule to be loaded

Outputs: `ruleHTR` HTR equivalent of the rule `rulename`

Local Vars: none

Side Effects: none

```
BEGIN
  search database for all PCNs that refers to ruleName
  if none found then exit algorithm
  ruleHTR.RHS ← ruleName + ()
    (* arguments will be fixed shortly *)

  for each PCN Pi that was found do
  begin
    Load_Rule(Pi.prePlace)
      (* only loads it if Pi.prePlace is a rule *)

    ruleHTR.LHS ← ruleHTR.LHS + Pi.prePlace + ()
    Si ← inscription of PCN Pi
    for each '=' condition Ei in Si do
    begin
      find arguments in ruleHTR that correspond to positions
          listed in Ei
      create a common variable for these arguments
    end
    create variables for all unassigned arguments in ruleHTR
  end
  return (ruleHTR)
END. (* ALGORITHM LOAD_RULE *)
```

Note: this algorithm correctly handles the case of multiple rules defining the same virtual table, but does not handle indefinite clauses.

Algorithm SAVE_RULE

Inputs:	`ruleHTR`	HTR representation of the rule to be saved
Outputs:	`ruleTrans`	entry in TRANSITION relation for `ruleHTR`
	`rulePTP`	entry in PTP relation for `ruleHTR`
Local Vars:	`script`	inscription of `rulePCN`
Side Effects:	none	

```
BEGIN
  for each predicate Li in ruleHTR.LHS do
    for each argument Pj in Li do
      if Pj has not already been used in script then
      begin
        find matching arguments Mk in predicate Pl
          of ruleHTR.LHS
        find matching arguments Mq in predicate Pr
          of ruleHTR.RHS
        script ← script + "i.j=k.l=-q.r,"
      end
```

```
  for each predicate Ri in ruleHTR.RHS do
    for each argument Pj in Ri do
      if Pj has not already been used in script then
      begin
        find matching arguments Mq in predicate Pr
          of ruleHTR.RHS
        script ← script + "–i.j=–q.r,"
      end
  ruleTrans.inscription ← script
  generate ruleTrans.transID
  for each predicate Pl in ruleHTR.LHS do
  begin
    rulePTP.transID     ← ruleTrans.transID
    rulePTP.prePlace    ← name of Pl (* see below† *)
    rulePTP.prePosition ← l
    rulePTP.preColumns  ← number of columns in Pl
    rulePTP.postPlace   ← name of predicate on ruleHTR.RHS
  end
  return (ruleTrans, rulePTP)
END. (* ALGORITHM SAVE_RULE *)
```

†**Note:** this algorithm makes the assumption that there is only one predicate on the RHS of the rule, that is, it does not handle indefinite clauses.

5.5.3 Current Limitations

SQUALID is now a working, functional system, but there are several features which have not been implemented fully. First, and most obvious, only a small subset of SQL has been implemented, in particular the subset required to demonstrate the major principles underlying the system. Other features within this subset were not implemented because of technical problems, or because they were not considered absolutely essential. This section lists the major features that have not been implemented.

The SELECT Statement

The SELECT statement has been almost fully implemented; however, a few features have been omitted or are not yet fully implemented. These are:

- SQL statements which define aliases for tables or views are not yet supported. This is partly because the original SQLD_TO_SQL algorithm did not deal with them. The two main effects of this are that: 1) a table cannot be joined with itself; and 2) nested SELECT statements in the WHERE clause are restricted somewhat in their use.

- The `ORDER BY <number>` form of the ORDER BY clause has not been implemented due to technical difficulties with the SQLD parser. The `SELECT <table-name>.*` form of the SELECT clause has also not been implemented for similar reasons.

- The GROUP BY and HAVING clauses are not fully implemented.

These features will eventually be fully implemented, but there is no real urgency for any of them at this time. Most of them are fairly simple conceptually, but would be somewhat difficult to integrate into the current system structure. A considerable rewrite of some of the components will probably be required to incorporate these features fully.

5.6 Data Structures

This section discusses the more technical details of the major data structures used in the implementation of SQUALID. The first section will cover the data structures used to represent and store rules in memory. The second section discusses the data structures used for storing SQL statements.

5.6.1 Data Structures for Storing Rules

The data structure for storing rules in memory is a linked list of HTR-like structures. As already stated, this appears well suited to the algorithms used, despite the shortcomings of the HTR structure. An interesting point to note is that the data structure used is fairly similar to a data structure used by C-Prolog to store program clauses, albeit considerably simpler.

The data structure itself is illustrated in Figure 5-2, using the by now familiar *Grandparent* rule. Some parts of the structure are not shown for the sake of clarity. Note that the logical operators are not explicitly represented in the structure. This is because the predicates on the left-hand side of a clause are always separated by $\wedge$'s, those on the right-hand side are always separated by $\vee$'s and there is always a $\rightarrow$ in between. The only operator that needs to be explicitly represented is $\neg$, which does not occur very often and causes other problems which are not relevant here. Negation is currently represented in the data structure by a boolean flag.

One aspect of this structure that is not entirely clear at first glance is the need for the extra level of indirection between the predicates and the argument definitions. Why not connect pointers directly from a predicate to the appropriate argument in the argument list, missing out the extra level entirely? The extra level is there for a very good reason—since we do not know how many arguments a predicate may have, we need some way of representing an arbitrary number of arguments. The best way to implement this sort of structure is to include an extra level of indirection. The other approach is to have more than one `next` pointer in each argument definition, but this would be more difficult to handle.

One problem with this structure is building the list of predicate arguments at the bottom of the tree when the rule is created. The list has been implemented as a doubly-linked global list, to which pointers are attached, as shown in Figure 5-3 (in simplified form). Each time an argument definition is encountered when creating a rule, the argument list is searched to see if an occurrence of that argument already exists. If not, a new argument definition is created and added to the list; otherwise a pointer from the rule to the existing argument definition is created. The list is doubly-linked for historical reasons—initially it was necessary

to traverse the list in both directions, but this no longer applies. The list was left as it was in case this structure became useful later.

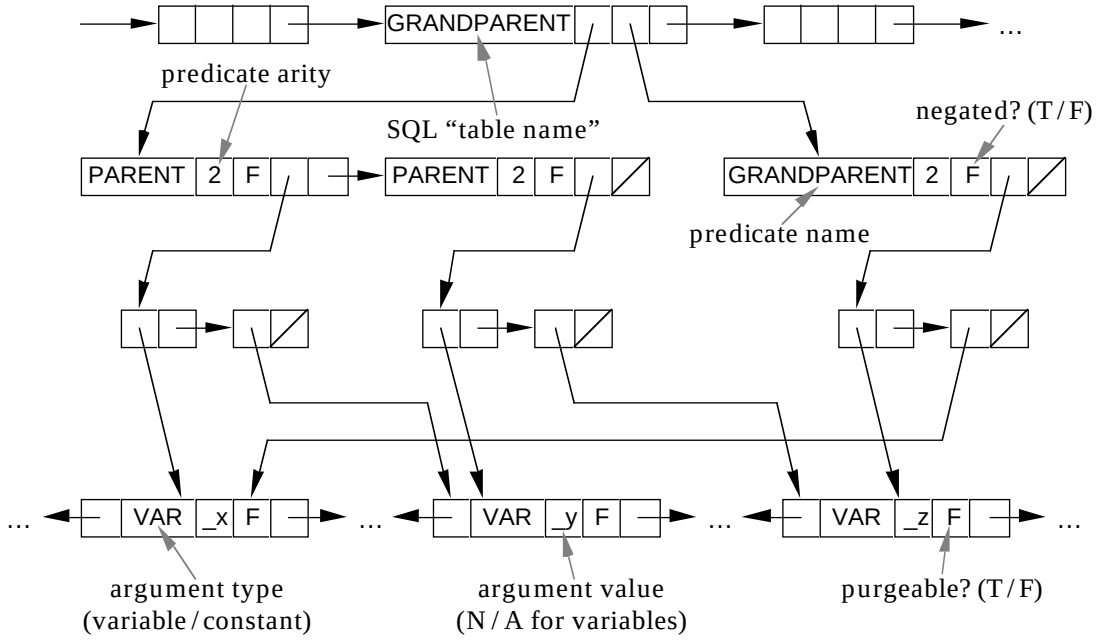


Figure 5-2. Data structure for rules (*Grandparent* rule).

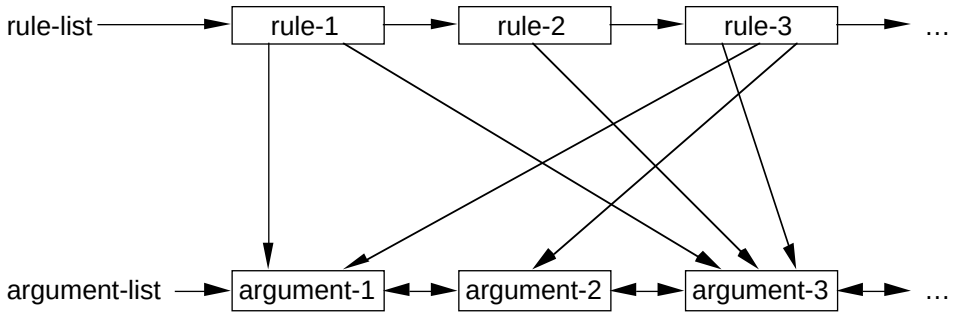


Figure 5-3. The argument list.

A problem encountered with the original deduction module was that the deduction routines permanently altered the rule list, so a copy had to be made of any rule that was going to be used in the deduction process. This rule was appended to the rule list, and was marked as being *purgeable*. After each deduction phase, all purgeable rules and their associated arguments were deleted from the rule and argument lists. This technique has been retained in the current version of SQUALID to help with memory management.

5.6.2 Data Structures for Storing SQL Statements

The only statements that really need to be stored in memory are those that may potentially contain rule references. All others can be sent directly to SQL. Those statements which do need to be stored fall into two distinct categories: data-

affecting statements (SELECT, INSERT, DELETE and UPDATE); and schema-affecting statements (ALTER/CREATE/DROP TABLE, GRANT and REVOKE). The data-affecting statements are all similar in nature, although they differ in detail; a similar state of affairs exists for the schema-affecting statements.

```
StatementKinds = ( CreateTabl,
                   DeleteStmt,
                   InsertStmt,
                   SelectStmt,
                   UpdateStmt  );

SQLStatement = record
                 case stmtKind : StatementKinds of
                   CreateTabl : ( creBody : CreateTabl_ptr );
                   DeleteStmt : ( delBody : DeleteStmt_ptr );
                   InsertStmt : ( insBody : InsertStmt_ptr );
                   SelectStmt : ( selBody : SelectStmt_ptr );
                   UpdateStmt : ( updBody : UpdateStmt_ptr );
               end;
```

Figure 5-4. The `SQLStatement` data structure (Pascal declarations).

The top-level structure used for storing statements is the `SQLStatement` data type; see Figure 5-4 for the Pascal TYPE declaration of this structure. This is implemented as a variant record, which consists of two fields: a field to indicate the type of statement being stored, for example a SELECT statement, and a pointer to a structure which represents the appropriate statement. The `stmtKind` field is the tag field for the variant record, and determines what the second field points to.

Data-Affecting Statements

The only data-affecting statement that has been implemented is the SELECT statement. The same basic structures can be used in all four statements however, so a description of the structure used to store a SELECT statement is sufficient.

An example of this structure (the `SelectStmtType` data structure) can be found in Figure 5-5. Note that some parts of the structure have been omitted from the diagram for the sake of clarity; the full structure contains additional bookkeeping information about the initial state of the statement.

The structure consists of six pointers to the various clauses of the SELECT statement. Of these clauses, the FROM, GROUP BY and ORDER BY clauses are represented by simple linked lists. The SELECT, WHERE and HAVING clauses are more interesting however. Each of these clauses may contain, or consist wholly of, arbitrary expressions. These expressions can (in general) include any of the following items: column names; aggregate functions like SUM and AVG; nested SELECT statements; numeric or character literals; parenthesised expressions; and operators or predicates.

The simplest method of representing this form of expression is to rewrite the expression into PREFIX FORM and store it as a binary tree, thus removing the

need to explicitly store parentheses. For example, consider the representation of the WHERE clause in Figure 5-5. The expression

```
Parent >= "John"
```

can be rewritten in prefix form as

```
>= (Parent, "John").
```

This can then be directly converted to the binary tree shown in Figure 5-5 by reading from left to right. The item outside the parentheses is the parent node; the two items inside the parentheses are the left and right branches of the tree.

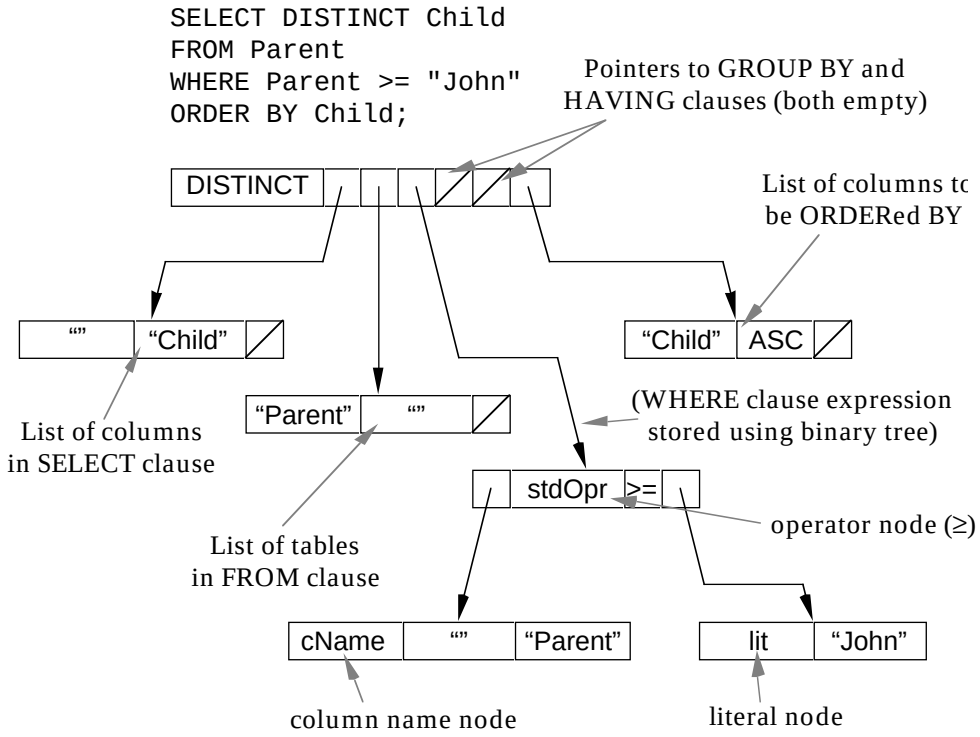


Figure 5-5. An example of the `SelectStmtType` structure in use.

Let us consider a more complicated example. Suppose we have the following SELECT statement:

```
SELECT *
FROM   Employee
WHERE  (   (Salary > 20000)
        OR (Surname IN ( SELECT Name
                         FROM   Manager ))
   AND (Surname LIKE "J%"))
```

This statement will return all employees whose surnames begin with J, who are either earning more than $20,000 or are a manager. This expression would be represented in prefix form as this:

```
AND (LIKE (Surname, "J%"), OR (> (Salary, 20000), IN (Surname, <subselect>)))
```

This can be represented by the binary tree structure in Figure 5-6.

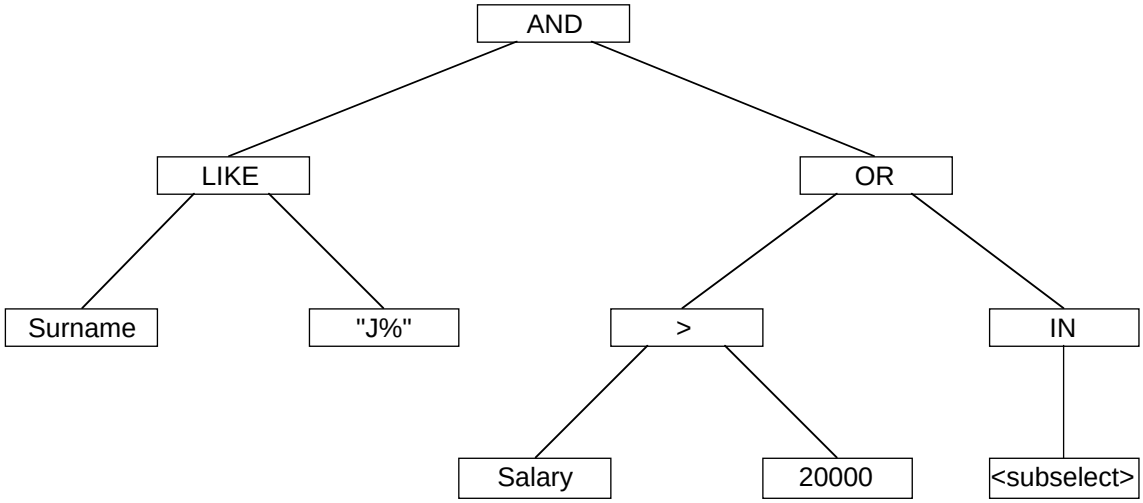


Figure 5-6. Binary tree for a complicated expression.

The advantages of using this binary tree structure are that it is relatively easy to build (the SQLD parser builds the expression trees automatically as part of the parsing process), and the precedence of the operators in the expression is implicitly encoded in the structure. To rebuild the expression, we perform a depth-first traversal of the tree, inserting parentheses around each sub-expression as it is completed.

Schema-Affecting Statements

The only schema-affecting statement that has been implemented is the CREATE TABLE statement. The other schema-affecting statements are similar in structure to this statement, however. An example of the structure used to store this statement (the `CreateTablType` data structure) can be found in Figure 5-7.

5.7 Summary

In this chapter we have described the major parts of the implementation of SQUALID: the deduction module, the SQL interface, how they are linked together and the main internal data structures used. The system is written mostly in VAX Pascal; one module is written in VAX SQL module language, and one is a YACC grammar specification file.

```
CREATE TABLE Grandparent
  ( GParent CHAR(20) PRIMARY KEY,
    GChild  CHAR(20) NOT NULL     )
FROM RULE
  IF   Parent(x,y) AND Parent(y,z)
  THEN Grandparent(x,z);
```

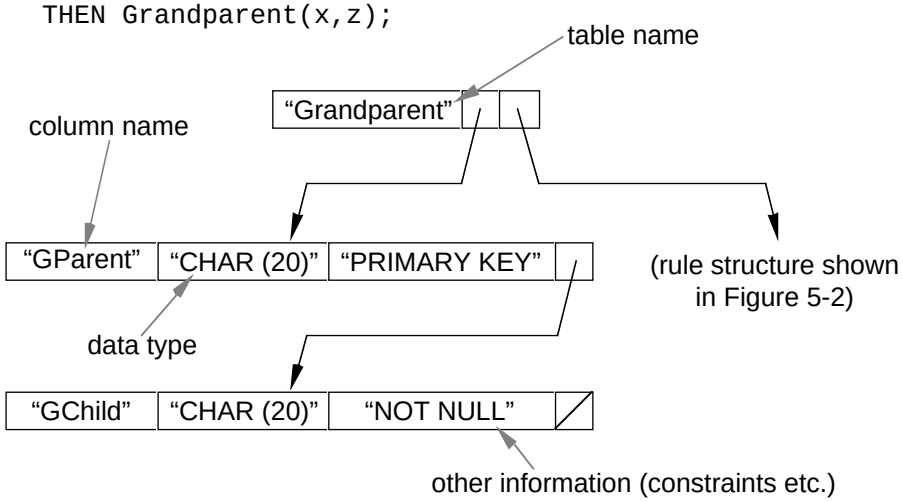


Figure 5-7. An example of the `CreateTablType` structure in use.

The deduction module is based on Prolog's deductive mechanism, and uses a relatively simple resolution scheme to expand references to virtual tables. It does not deal with recursive rules. The deduction module is completely separate from the rest of the system, thus making SQUALID independent of the deductive processor.

The SQL interface of SQUALID is based on the dynamic SQL processor of VAX SQL. This allows a program to accept arbitrary SQL statements at run time, rather than embedding fixed SQL statements directly into the program code. Pre-processed SQL statements are sent to VAX SQL, which applies them to the underlying database.

SQL statements input into SQUALID are passed to a pre-processor which detects any rule references and performs deduction if required. The pre-processed statement is then passed to the SQL interface, which executes the modified statement. Rules are stored in the underlying database using a modified form of the PCN representation. When required, rules are loaded from the database into an HTR-style structure in main memory.

Chapter 6

Summary and Conclusions

6.1 Introduction

This chapter will briefly summarise what has gone before, and draw some conclusions from what has been done.

6.2 A Summary of Logic and Databases

In Chapter 2 we introduced first-order logic, and described the basic concepts and symbols underlying it. In this section, we will briefly summarise how relational databases, deductive or otherwise, can be related to first-order logic. We will also briefly recap the major unsolved problems with deductive databases.

6.2.1 Model Theory and Proof Theory

Model theory and proof theory refer to the two basic methods of evaluating logical formulae. In model theory, we specify truth values for wffs; that is, each formula is assigned a truth value. This is also known as the semantic view. In proof theory, we derive new formulae from a set of existing formulae using a set of well-defined inference rules. This is also known as the syntactic view.

Conventional relational databases have generally been viewed from a model-theoretic point of view. That is, each formula in the database must be either true or false (usually, every formula in the database is true). However, databases may also be viewed as a first-order theory, that is, from a proof-theoretic point of view. Deductive databases are particularly suited to this view.

6.2.2 Problems With Deductive Databases

There are several major problems associated with deductive database which have not yet been fully solved. A brief review of each of these problems follows.

Indefinite and negative data. One of the standard assumptions that is usually made when implementing a database is the Closed World Assumption (CWA), which effectively states: if a fact cannot be found in the database, then assume it to be false. This is fine for definite data, but when we introduce indefinite and negative data into a database, we can also introduce inconsistencies in the data. Several logical formalisms have been proposed to alleviate aspects of this problem. It has not yet been completely solved. (Sections 2.6.1 and 2.6.2)

Recursion. If we have a database which contains a recursive rule, how can we determine when to stop applying the rule? If we are not careful, we may get

caught in an endless loop. Several promising approaches have been developed to handle recursive rules. (Section 2.6.3)

How to treat rules. It is not clear whether a given rule should be used as a deductive law or as an integrity constraint. We know of no effective method for dealing with this problem, though there are some heuristic approaches. (Section 2.6.4)

Functions. If we allow logical functions into databases, we introduce what are known as intensional entities. For example, we may form the description *father*(John), even though we don't know who John's father is. Different descriptions may refer to the same entity, so testing the equality of these entities is difficult. The current state of this problem is unknown. (Section 2.6.5)

6.3 A Summary of SQUALID

SQUALID is an integrated deductive database system, based on Rdb/VMS and VAX SQL. It consists of extensions to VAX SQL to allow the handling of rules. In this section we will give a brief summary of the system.

6.3.1 Aims of the System

SQUALID is designed as an extension to an existing conventional DBMS, in this case, Rdb/VMS. The intention was to extend the DBMS's capabilities with facilities for defining, using and manipulating rules. This was done by creating a separate module, to handle all deduction-related activities; this module was linked to the underlying DBMS by the standard interface provided (VAX SQL module language and dynamic SQL). The existing query language, SQL, was also extended with appropriate deductive capabilities. The main advantage of basing SQUALID on an existing DBMS is that data retrieval should be reasonably efficient, as opposed to systems based on logic programming principles.

The other main objective was to make the deductive process as transparent as possible. That is, the user should not need to know whether a table physically exists or is defined by a rule in order to extract data from it; all they need to know is that a table exists and that they can retrieve data from it. The internal mechanism used to actually retrieve that data is generally not important. To this end, SQUALID has been designed to handle all deductive processing internally. Deduction is only performed when required, without user intervention.

6.3.2 Structure of the System

SQUALID is made up of five main components (see Figure 5-1 on page 62):

- the SQLD parser;
- the SQLD preprocessor;
- the deduction module;

- the dynamic SQL interface; and
- the underlying Rdb/VMS database.

The user interacts with SQUALID as follows:

1. An SQLD statement is entered at the keyboard. This is parsed by the SQLD parser for correctness of syntax. As this is done, a representation of the statement is also built in main memory.

2. If the statement is syntactically correct, it is passed to the SQLD preprocessor, which scans the statement for references to virtual tables.

3. For each virtual table reference found, the SQLD preprocessor calls the deduction module. The result of this is a list of base tables which are equivalent to the original virtual table.

4. The statement is reassembled into text form from the modified internal representation.

5. The preprocessed statement is passed to the dynamic SQL interface, which prepares the statement for execution and sends it to the underlying database for execution.

6. The statement is executed against the database, and the results are passed back to the dynamic SQL interface. The results of the statement, if any, are displayed on the user's terminal.

Rules are stored in the underlying database using a modified form of the PCN representation developed by Cheiney and de Maindreville [1989]. This allows rules to be treated in a similar way to conventional data. It also has the advantage that rules may be retrieved more efficiently using relational methods.

6.3.3 SQUALID in Terms of Bell's Model

This section describes how SQUALID stands in terms of the conceptual model described in Chapter 4 [Bell *et al.* 1990]. We will consider the three levels of integration separately.

Logical Level

SQUALID is tightly integrated at the logical level. A single language is used, which is oriented towards data manipulation. In this case, SQL has been extended with facilities for defining and manipulating rules. No support for recursion has been added yet (which restricts things somewhat), but it should be possible to extend SQL with appropriate operators [Agrawal 1988, Date 1990].

Function Level

SQUALID is loosely integrated at the function level. When a query is made, deduction is performed (if necessary), and the query is rewritten into a form that

can access the EDB. This query is then sent to SQL and processed normally. The deduction module is totally independent of the system as a whole, and could be changed relatively easily—only the interfacing routines between SQUALID and SQL would need to be rewritten.

Physical Level

SQUALID is tightly coupled at the physical level. The EDB and the IDB are both stored in a relational database. Rules are stored in the database using a variant form of the PCN representation introduced in Chapter 3. When required for deduction, they are read into memory and processed there.

6.3.4 Unresolved Issues and Future Extensions

There are many things that could be added to SQUALID. The features which could be added in the near future are, in order of priority:

- full implementation of the SELECT statement.
- implementation of the ALTER and DROP TABLE statements.
- implementation of the DELETE, INSERT and UPDATE statements.

More long-term plans include:

- implementation of the remaining SQL statements.
- support for recursion.
- alternative deduction algorithms.
- support for indefinite data.
- a complete rewrite of the system in C to improve portability.

Note that many of the features listed above are related.

6.4 Conclusions

The main aim of the research has been achieved: a simple integrated deductive DBMS has been developed and implemented, and is now operational. It lacks many of the features of a full-blown DBMS, but is sufficient to demonstrate the principles involved.

The emphasis throughout has been on making the system easy to use from an end-user point of view. This objective has been achieved: the user does not need to know anything about the deductive capabilities of SQUALID in order to be able to take advantage of them. Rather, the deductive process underlies all operations in the system, and is only used when necessary—the user is not forced to decide whether a given operation requires deduction or not.

Future deductive database systems will probably be homogeneous; that is, there will be little distinction between the components of the system. Implementing such a system will be a large and complicated job however. Integrated systems have the advantage that existing technology can be used, without having to rewrite everything from scratch. This is how SQUALID was implemented. An existing product (Rdb/VMS) was used as the underlying DBMS. The deductive extensions were written as separate modules which are totally independent of the underlying DBMS. It would be relatively easy to port SQUALID to a different DBMS; only the interface between SQUALID and the underlying DBMS would have to be rewritten.

Similarly, the deductive module of SQUALID is separate from the main engine of the system. A new deductive module could be incorporated into SQUALID without great difficulty.

What are the advantages and disadvantages of an integrated deductive DBMS? The main disadvantages are:

- An integrated deductive DBMS will probably not be as efficient as a homogeneous system. This is because the deductive portions are “bolted on” to the existing system, rather than being an integral part of it.

- There is not yet any standard for deductive query languages. For example, we have defined some possible extensions to SQL to handle rules. Unfortunately, other researchers have either done something similar, or gone as far as defining completely new languages. The rule language RDL1 could be considered a *de facto* standard, but it is not particularly suited to end-users.

In the short term, however, we feel that these problems are outweighed by the advantages of an integrated system:

- Much of the system is already implemented. Developers of such systems will not have to “reinvent the wheel,” as it were.

- The deductive system, if implemented in a modular fashion with a standard language (e.g. Pascal or C), is very portable. It may be ported to different DBMSs without much difficulty. This assumes, of course, that a standard query language such as SQL is used throughout.

- Users will already know how to use the existing system, so their learning curve will be significantly reduced.

- The deductive mechanisms may be changed without unduly affecting the underlying DBMS.

It remains to be seen whether further integrated systems will be developed in future, and how effective they will be.

References

[Agrawal 1988]

R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, **14**(7), 1988.

[Bancilhon and Ramakrishnan 1986]

F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. SIGMOD '86*, pages 16–52, Washington, 1986.

[Bancilhon *et al.* 1986]

F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15, Cambridge, 1986.

[Bell *et al.* 1990]

D.A. Bell, J. Shao and M.E.C Hull. Integrated deductive database system implementation: A systematic study, *The Computer Journal*, **33**(1):40–48, 1990.

[Boas and Boas 1986]

G. van Emde Boas and P. van Emde Boas. Storing and evaluating Horn-clause rules in a relational database. *IBM Journal of Research and Development*, **30**(1):80–92, 1986.

[Bossu and Siegel 1984]

G. Bossu and P. Siegel. Nonmonotonic reasoning and databases. In H. Gallaire *et al.*, editors, *Advances in Database Theory*, Volume 2, pages 239–284. Plenum, New York, 1984.

[Bossu and Siegel 1985]

G. Bossu and P. Siegel. Saturation, nonmonotonic reasoning and the closed-world assumption. *Artificial Intelligence*, **25**:13–63, 1985.

[Bratko 1986]

I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1986.

[Chamberlin and Boyce 1974]

D.D. Chamberlin and R.F. Boyce. SEQUEL: A structured English query language. In *Proc. ACM SIGMOD Workshop on Data Description, Access and Control,* Ann Arbor, Mich., 1974.

[Chang 1978]

C.L. Chang. DEDUCE 2: Further investigations of deduction in relational databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases,* pages 291–236, Plenum, New York, 1978.

[Cheiney and de Maindreville 1989]

J.-P. Cheiney and C. de Maindreville. Relational storage and efficient retrieval of rules in a deductive DBMS. In *Proc. 5th International Conference on Data Engineering,* pages 644–651, Los Angeles, U.S.A. IEEE Computer Society Press, Washington D.C., 1989.

[Chisholm *et al.* 1987]

P. Chisholm, D. Chen, P. Ferbrache, P. Thanisch and M.H. Williams. Coping with indefinite and negative data in deductive databases: A survey. *Data and Knowledge Engineering,* **2**:259–284, 1987.

[Clark 1978]

K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322, Plenum, New York, 1978.

[Codd 1970]

E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, **13**(6):377–387, 1970.

[Dahl 1982]

V. Dahl. On database systems development through logic. *ACM Transactions on Database Systems,* **7**(1):102–123, 1982.

[Date 1981]

C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., 3rd edition, 1981.

[Date 1986]

C.J. Date. *Relational Database: Selected Writings.* Addison-Wesley, Reading, Mass., 1986.

[Date 1987]

C.J. Date. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Mass., 1987.

[Date 1990]

C.J. Date. *An Introduction to Database Systems*, Volume 1. Addison-Wesley, Reading, Mass., 5th edition, 1990.

[DEC 1989a]

VAX Rdb/VMS Guide to Using SQL. Digital Equipment Corporation, 1989.

[DEC 1989b]

VAX Rdb/VMS SQL Reference Manual. Digital Equipment Corporation, 1989.

[Gabbay and Sergot 1986]

D.M. Gabbay and M.J. Sergot. Negation and inconsistency, I. *Journal of Logic Programming*, **3**(1):1–35, 1986.

[Gallaire and Minker 1978]

H. Gallaire and J. Minker. *Logic and Data Bases.* Plenum, New York, 1978.

[Gallaire *et al.* 1984]

H. Gallaire, J. Minker and J.-M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, **16**(2):153–185, 1984.

[Gardarin 1987]

G. Gardarin. Magic functions: A technique to optimize extended Datalog recursive programs. In *Proc. 13th Very Large Data Bases Conference*, pages 21–30, Brighton, 1987.

[Grant and Minker 1986]

J. Grant and J. Minker. Answering queries in indefinite databases and the null value problem. *Advances in Computing Research*, **3**:247–257, 1986.

[Gurk and Minker 1961]

H. Gurk and J. Minker. The design and simulation of an information processing system. *Journal of the ACM*, **8**(2):260–270, 1961.

[Helman and Veroff 1988]

P. Helman and R. Veroff. Designing deductive databases. *Journal of Automated Reasoning*, **4**:29–68, 1988.

[Henschen and Naqvi 1984]

L. Henschen and S. Naqvi. On compiling queries in recursive first-order databases. *Journal of the ACM*, **31**(1):47–85, 1984.

[Kellogg *et al.* 1978]

C. Kellogg, P. Klahr and L. Travis. Deductive planning and pathfinding for relational data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 179–200, Plenum, New York, 1978.

[Kowalski 1974]

R. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP 4*, pages 569–574, Amsterdam, 1974.

[Kowalski 1978]

R. Kowalski. Logic for data description. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 77–103. Plenum, New York, 1978.

[Lifschitz 1985]

V. Lifschitz. Closed world databases and circumscription. *Artificial Intelligence*, **27**:229–235, 1985.

[Lloyd 1984]

J.W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[Lloyd and Topor 1984]

J.W. Lloyd and R.W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, **1**(3):225–240, 1984.

[Lloyd and Topor 1985]

J.W. Lloyd and R.W. Topor. A basis for deductive databases, I. *Journal of Logic Programming*, **2**(2):93–109, 1985.

[Lloyd and Topor 1986]

J.W. Lloyd and R.W. Topor. A basis for deductive databases, II. *Journal of Logic Programming*, **3**(1):55–67, 1986.

[McCarthy 1980a]

J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, **13**:27–39, 1980.

[McCarthy 1980b]

J. McCarthy. Circumscription and other non-monotonic formalisms. *Artificial Intelligence*, **13**:171–172, 1980.

[Mendelson 1979]

E. Mendelson. *Introduction to Mathematical Logic*. D. van Nostrand, Princeton, 2nd edition, 1979.

[Minker 1978]

J. Minker. An experimental relational database system based on logic. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 107–147. Plenum, New York, 1978.

[Minker 1982]

J. Minker, On indefinite databases and the closed world assumption. In *Proc. 6th International Conference on Automated Deduction*, pages 292–308. Springer-Verlag, 1982. (*Springer-Verlag Lecture Notes in Computer Science*, Vol. 138.)

[Minker 1983]

J. Minker. On deductive relational databases. In *Proc. 5th International Conference on Collective Phenomena*, pages 181–200, 1983. (*Annals of the New York Academy of Science*, Vol. 410.)

[Minker 1988]

J. Minker. Perspectives in deductive databases. *Journal of Logic Programming*,
5(1):33–60, 1988.

[Naish and Thom 1983]

L. Naish and J.A. Thom. The MU-Prolog deductive database. Technical Report 83/10, Dept. of Computer Science, University of Melbourne, 1983. (*Springer-Verlag Lecture Notes in Computer Science*, Vol. 138.)

[Nicolas and Gallaire 1978]

J.-M. Nicolas and H. Gallaire. Data base: Theory vs. interpretation. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 33-54. Plenum, New York, 1978.

[Nicolas and Yazdanian 1982]

J.-M. Nicolas and K. Yazdanian. An outline of BDGEN: A deductive DBMS. In *Proceedings of IFIP 83 Congress*, pages 711–717, North-Holland, Amsterdam, 1983.

[Pratt 1990]

P.J. Pratt. *A Guide to SQL*. Boyd & Fraser, Boston, 1990.

[Przymusinski 1986a]

T.C. Przymusinski. On the semantics of stratified deductive databases. In J. Minker, editor, *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*, pages 433–443, Washington, 1986.

[Przymusinski 1986b]

T.C. Przymusinski. A query answering algorithm for circumscriptive theories. In *Proc. ACM SIGART International Symposium on Methodologies for Intelligent Systems*, pages 85–93, Knoxville, Tenn., 1986.

[Reiter 1978]

R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 33–54. Plenum, New York, 1978.

[Reiter 1982]

R. Reiter. Circumscription implies predicate completion (sometimes). In *Proc. International Joint Conference on Artificial Intelligence*, pages 418–420, 1982.

[Reiter 1984]

R. Reiter. Towards a logical reconstruction of relational database theory. In M.L. Brodie, J.L. Mylopoulos and J.W. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, pages 191–238. Springer-Verlag, 1984.

[Robinson 1965]

J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, **12**(1):23–41, 1965.

[Small 1988]

C. Small. The implementation of the Exegesis system. *The Computer Journal*,
31(2):125–132, 1988.

[Spyratos 1987]

N. Spyratos. The partition model: A deductive database model. *ACM Transactions on Database Systems*, **12**(1):1–37, 1987.

[Tsang 1990]

Personal communication.

[Yahya and Henschen 1985]

H. Yahya and L.J. Henschen. Deduction in non-Horn databases. *Journal of Automated Reasoning*, **1**:141–160, 1985.

Glossary

Anonymous Variable. Placeholders in Prolog that can be used instead of a variable, if that variable would be used only once in a clause. They are represented by a single underscore ("_") character. For example, suppose we have the following Prolog clause [Bratko 1986]:

```
hasachild(X) :- parent(X, Y).
```

This clause says that X has a child if X is the parent of Y. The result of this clause does not depend on the value of Y, so we could replace Y with an anonymous variable:

```
hasachild(X) :- parent(X, _).
```

Each time a single underscore is used in a clause, it represents a new anonymous variable. That is, any two anonymous variables are not equal.

ANSI. An acronym for the American National Standards Institute.

Arity. The number of arguments that a predicate has (i.e. the number of terms in the predicate). For example, the arity of the predicate $P(x, y, z)$ is three.

BNF. An acronym for Backus-Naur Form. A standard notation for defining the syntax of languages and other formal systems.

DBMS. An abbreviation of Database Management System.

DDB. An abbreviation of Deductive Database

EDB. An abbreviation of Extensional Database. That part of a deductive database which is explicitly stored.

Expert System. A rule-based system which can make deductions and decisions in some limited area of expertise. The classic example of an expert system is MYCIN, which could diagnose lung diseases with over 90% certainty, although it was never used in "live" situations. To do this it used a very large rule base to perform inferences on information about the patient's condition. Expert systems are often used as decision *support* systems, rather than as decision *making* systems.

Generalisation. One of the two inference rules of the first-order predicate calculus (the other is MODUS PONENS). It can be stated as:

$$P \rightarrow (\forall x)P,$$

where P is a well-formed formula. That is, if the formula P is true, then it is also true for any value of x.

HTR. An acronym for Hierarchical Tree Representation. A structure for storing rules.

IDB. An abbreviation of Intensional Database. That part of a deductive database which represented by rules, i.e. implicitly stored.

Integrity Constraint. A condition that data in a database must satisfy. From a database perspective, "integrity" refers to the accuracy or correctness of the data in the database. If a user attempts to perform some operation which violates an integrity constraint, the DBMS usually rejects the operation. In some more complicated cases, the DBMS may have to take some action to restore the database to a state which satisfies the integrity constraint. For example, some systems allow *cascaded deletion*, that is, if a particular item is deleted, then all data which depends on that item must also be deleted.

Lattice. A lattice can be defined as follows:

- A *relation* is an operation on a set of objects (e.g. "+" is a relation on numbers).
- A relation ~ is a *partial order* if it is:
  1. *reflexive*, i.e. $a \sim a$.
  2. *antisymmetric*, i.e. $a \sim b \Rightarrow b \sim a$.
  3. *transitive*, i.e. $a \sim b$ and $b \sim c \Rightarrow a \sim c$.

  Partial orders are usually denoted by the symbols $\leq$ and $\geq$.
- A *partially ordered set* (or *poset*) is a set with partial order $\leq$ or $\geq$.
- Let P be a poset. For any subset Y of P, an element $u \in P$ is called an *upper bound* for Y if $u \geq y \ \forall y \in Y$. Similarly, a *lower bound* for Y is an element $l \in P$ such that $l \leq y \ \forall y \in Y$.
- The *least upper bound* (*lub*) for a subset Y of P is an element $u \in P$ that is an upper bound for Y, and there is no $u' \in P$ with $u' \leq u$ and u' an upper bound for Y. Similarly, the *greatest lower bound* (*glb*) for Y is a lower bound l for Y, and there is no l' with $l' \geq l$ which is also a lower bound for Y.

- In any poset P, the *meet* of any two elements x and y (denoted $\wedge$) is the glb of x and y (if it exists). Similarly, the *join* of x and y (denoted $\vee$) is the lub of x and y (if it exists).

- A *lattice* is a poset L in which the meet and the join of every pair of elements exists.

Lattices are an important mathematical concept. For example, a Boolean algebra is just a particular form of lattice, with meet and join corresponding to **and** and **or** respectively.

LHS. An acronym for Left-Hand Side.

LIF. An acronym for Logical Inference Function.

LIU. An acronym for Logical Inference Unit. The part of a deductive database that actually performs the deduction.

LPL. An acronym for Logic Programming Language.

Makefile. A source file for the UNIX utility `make`. This utility automatically recompiles and links files in a project which have changed. The makefile contains a description of the relationships between the source files, which `make` uses to determine which files need to be updated. The VMS version of `make` is the Module Management System, or MMS.

Modus Ponens. One of the two inference rules of the first-order predicate calculus (the other is GENERALISATION). It can be stated as:

$$P \wedge (P \rightarrow Q) \rightarrow Q,$$

where P and Q are well-formed formulae. That is, if P is true, and Q follows from P, then Q is also true.

Non-monotonic Logic. A logic is either *monotonic* or *non-monotonic*. A monotonic logic is defined as follows: suppose we have a theory T (with a set of axioms W) from which a formula w can be proven (i.e. $W \vdash_T w$). Now add a new axiom a to this theory. If we can still prove w from this new theory (i.e. $W \cup \{a\} \vdash_T w$), then the logic is said to be monotonic. A non-monotonic logic does not have this property.

Null. A special marker placed in a field of a tuple when an actual value cannot be supplied. A null can generally be interpreted as meaning either “value not known” (i.e. we cannot enter a value because we do not know what the value is), or “value inapplicable” (e.g. a phone number for someone who has no phone). A null is *not* a data value, rather it is a marker to indicate that the

data is missing. For this reason, the often-used term "null value" should be avoided.

PCN. An acronym for Production Compilation Network. A structure for storing rules.

Predicate. A logical formula which has some truth value, usually true or false, depending on how many truth values the logic being used has.

Prefix Form. An alternate way of writing mathematical or logical expressions. Standard mathematical notation uses what is known as *infix* form: operators appear between the operands, e.g. 3 + 2. The two alternate ways of writing these expressions are *prefix* form, where the operators precede the operands, e.g. +(3, 2); and *postfix* form, where the operators come after the operands, e.g. (3, 2)+. Many scientific calculators use Reverse Polish Notation, which is essentially postfix.

RDBMS. An abbreviation of Relational DBMS.

RDMF. An acronym for Relational Data Manipulation Function.

RDML. An acronym for Relational Data Manipulation Language.

RHS. An acronym for Right-Hand Side.

Robinson Resolution Principle [Robinson 1965]. A rule of inference that allows new clauses to be derived from two given clauses that meet certain criteria. The principle is best explained by an example. From the clauses

$$\neg P(\mathrm{a}, \mathrm{b}, \mathrm{c}) \vee Q(\mathrm{d}, \mathrm{e}), \text{ and} \qquad (1)$$

$$P(x, y, z) \vee R(x, y), \qquad (2)$$

one can derive the clause

$$Q(\mathrm{d}, \mathrm{e}) \vee R(\mathrm{a}, \mathrm{b}). \qquad (3)$$

Clause (3) is found by looking at the literals in the clauses (1) and (2) that have the same predicate name, but one is negated and the other is not. The only literal of this type in clauses (1) and (2) is *P*. We then try and find some substitution of the variables *x*, *y*, and *z* that will make the two literals identical. In this case, we can make the substitution {a/*x*, b/*y*, c/*z*} (i.e. substitute a for *x*, etc.). We then eliminate the literals made identical by the substitution (these are said to be *unified*), form the disjunction of the remaining literals in the two clauses and apply the substitution to the remaining literals to obtain the derived clause (3).

There are several varieties of the resolution rule. Some of these are:

- *SL-Resolution*: Linear resolution with selection function. This is the "standard" form of resolution.
- *SLD-Resolution*: SL-resolution for definite (Horn) clauses [Lloyd 1984].
- *SLDNF-resolution*: SLD-resolution with negation as failure.
- *SLSNF-resolution*: SL-resolution with subsumption based on negation as failure [Przymusinski 1986a, 1986b].

SQL. An acronym for Structured Query Language.

SQLD. An acronym for SQL Deductive. A superset of standard SQL, containing extensions to create and manipulate rules. It is the query language of SQUALID.

SQUALID. An abbreviation of Structured Query And Logical Inference Database.

VAX Message Utility. A standard VMS utility for creating and displaying error messages. A source file is created which contains descriptions of error messages and how severe they are. This file is compiled and linked into a program, which can then call special routines to display the error messages.

YACC. An acronym for Yet Another Compiler-Compiler. YACC is a utility that was developed for UNIX. Given a grammar definition, it attempts to generate a table-driven LR parser. Users must write their own lexical analyser to scan the input and pass tokens to the parser. The original UNIX version produces C code, but it can be modified to support a different language, for example Pascal.

Appendix A

Examples

A.1 An Example of the Interpretive Method of Deduction

Consider a database with the facts

$F(e, b_1)$	$M(c, e)$	$H(a, c)$
$F(e, b_2)$	$M(c, f)$	
$F(e, b_3)$	$M(c, g)$	
$M(c, d)$		

where F, M, and H represent father, mother and husband respectively. The rules for this database are

$\mathbf{R_1}$	$M(x, y) \wedge M(y, z) \rightarrow GM(x, z)$,
$\mathbf{R_2}$	$M(x, y) \wedge F(y, z) \rightarrow GM(x, z)$,
$\mathbf{R_3}$	$GM(z, y) \wedge H(x, z) \rightarrow GF(x, y)$,

where GM and GF stand for grandmother and grandfather. The problem is to answer the query $GF(a, y)$, that is, who are a's grandchildren? We start from the query

$$GF(a, y), \qquad (1)$$

and apply R_3, using resolution, which gives

$$GM(z, y) \wedge H(a, z) \qquad (2)$$

as the subproblem to be solved. At this point we could invoke a selection function, which *should* provide advice to solve $H(a, z)$. We should solve this one first because it contains a constant. Otherwise we would have to arbitrarily search for a tuple that satisfies $GM(z, y)$, and hope that $H(a, z)$ (with appropriate value of z), was in the database. By accessing the database, we find that the fact $H(a, c)$ is in the database, and so z is bound to c. Now, we must solve the subproblem

$$GM(c, y). \qquad (3)$$

Since there are two ways of solving this goal (rules R_1 and R_2), we can make a choice to use rule R_2 (note that we could just as easily have used rule R_1), giving

$$M(\mathrm{c}, y_1) \wedge F(y_1, y), \qquad (4)$$

as the next subproblems to be solved. We can solve these subproblems in two steps, giving the answer b_1. Also, if we backtrack at previous choice points, we get b_2, b_3, d as further answers to the query.

A.2 An Example of the Compiled Method of Deduction

Using the example from the previous section, the deduction process would proceed as follows:

$$GF(\mathrm{a}, y) \qquad (1)$$

is again (using rule R_3) split into

$$GM(z, y) \wedge \mathrm{H}(\mathrm{a}, z), \qquad (2)$$

which is (if we choose rule R_2) reduced to

$$M(z, y_1) \wedge F(y_1, y) \wedge H(\mathrm{a}, z). \qquad (3)$$

Note that this is not the same as what happened with the interpretive approach. In the compiled approach, we look first at those subproblems which contain virtual relations. Since *H* is a base relation, we ignore it, and instead work on *GM*(*z*, *y*). We now see that the entire clause consists of references to base relations, so we can now access the EDB. As before, we select *H*(a, *z*) (because it contains a constant), and retrieve *H*(a, c) from the database, giving us

$$M(\mathrm{c}, y_1) \wedge F(y_1, y) \wedge H(\mathrm{a}, \mathrm{c}). \qquad (4)$$

Now we search the database for a tuple that satisfies $M(\mathrm{c}, y_1)$, and find $M(\mathrm{c}, \mathrm{e})$, giving us

$$M(\mathrm{c}, \mathrm{e}) \wedge F(\mathrm{e}, y) \wedge H(\mathrm{a}, \mathrm{c}). \qquad (5)$$

Finally, we look for tuples that satisfy *F*(e, *y*), which gives us the set of answers b_1, b_2, b_3. Once again, by backtracking at the choice points, we also find the answer d.

Appendix B

A Brief Overview of Standard SQL

B.1 Introduction

This appendix gives a brief overview of SQL. The second section describes the history of the language; the third section briefly covers some of the more important features of interactive SQL; and the fourth section discusses some problems with SQL. For those who are interested, a full BNF grammar for SQL may be found in Date [1987].

B.2 History of SQL

SQL is an acronym for Structured Query Language. It is often pronounced "sequel" for historical reasons. SQL consists of a set of facilities for defining, manipulating and controlling data in relational databases. In recent years it has become a *de facto* standard for relational query languages. An official standard has also been developed.

Relational database theory came into existence in 1970 with the publication of Codd's paper on the relational model [Codd 1970]. This paper led to much research and experimentation into relational database technology. One result of this was the development of a variety of relational languages in the early and mid-1970s. One such language was the "Structured English Query Language" (SEQUEL), developed at the IBM San Jose Research Laboratory in 1974 [Chamberlin and Boyce 1974]. This was the direct precursor of modern-day SQL. The first implementation of this language was in the IBM prototype system SEQUEL-XRM.

In 1976-1977, a new version of the language was defined, based on the experience gained from SEQUEL-XRM. This new version was called SEQUEL/2, later renamed to SQL for legal reasons. Work then began on a new prototype system called System R [Date 1981]. This became operational in 1977, and was tested at many IBM sites. It proved very popular, and several changes were incorporated into the system as a result of user feedback.

Because of the success of System R, it seemed inevitable that IBM would develop relational products based on SQL, so other companies began work on SQL-based systems as well. Indeed, one such system, ORACLE, was released *before* IBM's products, in 1979. IBM eventually did release two SQL products: SQL/DS for DOS/VSE (1981) and VM/CMS (1982) systems; and DB2 for MVS systems (1983).

During the next five years, many other vendors released their own SQL products. There were completely new products like DG/SQL and SYBASE, and

SQL interfaces to existing relational systems such as INGRES. By 1986, there were about fifty products running some dialect of SQL, on everything from microcomputers to mainframes.

An official standard for SQL was first suggested in 1982 by the American National Standards Institute (ANSI). Its Database Committee (X3H2) was chartered to develop proposals for a standard relational language. The X3H2 proposal was ratified as a standard in 1986, and was essentially the IBM dialect of SQL with a few minor changes.

B.3 SQL Features

Using SQL, a relational database is perceived by the user as a collection of *tables* (i.e. unordered collections of rows). For example, consider the database shown in Figure B-1 (this is a "classic" example used by Date). The tables S, P and SP represent, respectively, suppliers, parts and shipments of parts by suppliers.

SQL data manipulation statements (i.e. those that retrieve or update data) can be invoked interactively or from statements within application programs. We will only discuss interactive SQL for reasons of clarity.

B.3.1 Data Definition

Figure B-1 represents the database as it appears at a specific time (i.e. it is an *instance* of the database). The general structure of the database, or *schema*, is shown in Figure B-2. The clause AUTHORIZATION TED specifies that user TED is the creator of this schema. The three CREATE TABLE clauses define three empty tables with the appropriate names and columns. The data in these tables can be manipulated using the data manipulation statements discussed in the next section.

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Figure B-1. The suppliers-parts database (sample values).

```
CREATE SCHEMA AUTHORIZATION TED

CREATE TABLE S  ( SNO    CHAR(5)    NOT NULL,
                  SNAME  CHAR(20),
                  STATUS DECIMAL(3),
                  CITY   CHAR(15),
                  UNIQUE ( SNO ) )

CREATE TABLE P  ( PNO    CHAR(6)    NOT NULL,
                  PNAME  CHAR(20),
                  COLOR  CHAR(6),
                  WEIGHT DECIMAL(3),
                  CITY   CHAR(15),
                  UNIQUE ( PNO ) )

CREATE TABLE SP ( SNO    CHAR(5)   NOT NULL,
                  PNO    CHAR(6)   NOT NULL,
                  QTY    DECIMAL(5),
                  UNIQUE ( SNO, PNO ) )
```

Figure B-2. Schema definition example.

SQL can define two types of table, *base tables* and *views*. A base table could be referred to as a "real" table, that is, it physically exists somewhere in some form or other. A view, on the other hand, can be thought of as an "imaginary" table; it does not exist in physical storage, but appears to the user as if it does. Views are defined in terms of one or more underlying base tables. This technique will be explained further in Section B.3.3.

B.3.2 Data Manipulation

SQL has four data manipulation statements: SELECT for retrieving data; INSERT for adding new data to a table; UPDATE for changing existing data; and DELETE for deleting data. An example of each of these statements can be found in Figure B-3.

INSERT, UPDATE and DELETE may operate directly upon multiple rows; SELECT can only work on a single row. Note that the result of a SELECT statement is another table (this is derived from an existing table, and not stored in the database). This is called the *result table*.

Note that the SELECT statement can be much more involved than the simple SELECT-FROM-WHERE form shown here. You can also do such things as specify the ordering and grouping of the output and join multiple tables together in a single query (this is equivalent to the relational *join* operator). You can also apply aggregate functions to the output: COUNT, SUM, AVG (average), MAX (maximum) and MIN (minimum).

```
SELECT S.CITY                  Result:  CITY
FROM   S                                London
WHERE  S.SNO = 'S4'

INSERT                         Result:  Specified row
INTO SP ( SNO, PNO, QTY )               added to table
                                        SP
VALUES  ('S5','P1',1000 )

UPDATE S                       Result:  STATUS doubled
SET    STATUS = 2 * S.STATUS            for suppliers in
WHERE  S.CITY = 'London'                London (i.e., S1
                                        and S4)

DELETE                         Result:  Rows deleted
                                        from
FROM   P                                table P for
                                        parts
WHERE  P.WEIGHT > 15                    P2, P3 and P6
```

Figure B-3. Examples of data manipulation statements.

B.3.3 Views

Recall that a view does not physically exist in the database, but appears to exist to the user. Views are not directly stored; rather, their *definition* in terms of other tables is specified in the database definition. An example of such a definition is:

```
CREATE VIEW LS ( SNO, SNAME, STATUS )
    AS SELECT S.SNO, S.SNAME, S.STATUS
       FROM   S
       WHERE  S.CITY='London'
```

This creates a view called LS (for “London suppliers”), which is based on the base table S. The view acts as a “window” on the S table, through which the user can see the SNO, SNAME and STATUS values of rows in S for which the CITY value is London. When an operation is performed on a view, the operation and the view definition query are “merged” to give the equivalent operation on the underlying base tables. Note however, that in ANSI SQL updating of views is restricted to views based on subsets of a single table only.

B.3.5 Data Control

SQL also provides facilities for data control. These facilities fall into three basic categories:

Recovery and Concurrency

Standard SQL includes support for *transactions* as a means of controlling concurrent access to databases, and to aid recovery in the event of a system crash. A transaction is a sequence of operations terminated by either a COMMIT (which makes any changes permanent) or a ROLLBACK (which discards any changes).

Security

Security in SQL is provided by two means: views and the GRANT statement. Views let database administrators restrict access to data by creating views which only allow users to see the information that they need to see, and nothing else.

The GRANT statement places *privileges* on tables. To perform any operations at all, a user must hold the appropriate privilege for the combination of operators and the operands used, or the statement will be rejected. The possible privileges are SELECT, DELETE, INSERT and UPDATE. UPDATE can be restricted to certain columns of a table, whereas the others apply to a table as a whole.

Integrity

Integrity refers to the consistency of the data in a database. Standard SQL allows the definition of integrity constraints as part of the CREATE TABLE statement. Any operation which violates these constraints is rejected. The possible constraints are:

- NOT NULL: This column cannot contain nulls. Any attempt to insert a row containing a null in this column will fail.
- UNIQUE: This column (or combination of columns) can only contain unique values. Any attempt to insert a row with the same value(s) in this column or combination of columns will fail. UNIQUE columns must also be NOT NULL.

B.4 Some Problems with Standard SQL

SQL is a very powerful language, and has many strong points. These include [Date 1986]:

- simple data structure
- powerful operators
- improved data independence
- integrated data definition and data manipulation
- compilation and optimisation

However, the language has several weak points as well. This section will briefly cover some of these weak points.

B.4.1 Functions and Nulls

A major anomaly with the built-in functions is their handling of nulls. SUM, AVG, MAX and MIN ignore nulls, whereas COUNT counts *all* rows, including

those with nulls. Thus we can end up with the following counter-intuitive situation:

```
SELECT AVG ( S.STATUS ) FROM S      (Result = x)

SELECT SUM ( S.STATUS)  FROM S      (Result = y)

SELECT COUNT(*)         FROM S      (Result = z)
```

There is no guarantee that $x = y / z$. Similarly, the expression

```
SUM ( F1 + F2 )
```

is *not* equivalent to the expression

```
SUM ( F1 ) + SUM ( F2 )
```

B.4.2 Inconsistent Comparison Syntax

If you want to test (in a WHERE clause) whether a field is null, you must use the special comparison `field IS NULL`. It is not intuitively obvious why this is the case—why couldn't you use `field = NULL` instead, giving a more consistent approach? The only reason why this special comparison has been included is because, in theory, a null should not be equal to *anything*, so `field = NULL` is technically incorrect. This is only really a point of semantics though—from a user's point of view, it does not really matter.

B.4.3 Inconsistent Statement Syntax

Consider the following:

```
SELECT * FROM T …

UPDATE       T …

DELETE   FROM T …

INSERT   INTO T …
```

A more consistent approach would be to define a *table-expressions* (SQL expressions that return a table), and treat SELECT, UPDATE, etc. as operators, one of whose arguments is a table-expression.

Note also, that SQL has whole-record SELECT and INSERT operations, but no whole-record UPDATE operation. (DELETE must be a whole-record operation by definition.)

B.4.4 No Aliases in SELECT Clauses

SQL allows you to define an "alias" for a table in the FROM clause of a SELECT statement, for example `FROM T TX`. This facility is not available for expression in the SELECT clause, for example, `SELECT A + B C, D + E, ….` This would

allow you, for example, to sort the output based on the result of an expression in the SELECT clause. This is currently possible, but you must know the ordering of the columns in the output beforehand.

B.4.5 Legal Statements

Certain INSERT, UPDATE and DELETE statements are illegal. For example, consider the request "Delete all suppliers with a status less than the average." The natural way of expressing this would be:

```
DELETE
FROM   S
WHERE  S.STATUS <
     ( SELECT AVG ( S.STATUS )
       FROM S )
```

This statement is, however, illegal—the FROM clause in the subquery is not allowed to refer to the table against which the DELETE is to be done. Similarly, the statement:

```
UPDATE S
SET    STATUS = 0
WHERE  S.STATUS <
     ( SELECT AVG ( S.STATUS )
       FROM S )
```

is illegal, for much the same reasons.

B.4.6 Redundant Operators

The comparison operators =ANY, >ALL, and so on, are redundant, and can also be misleading to users. For example, consider the following example, which is taken from an IBM DB2 manual:

Query:
"Select employees who are younger than any member of department E21"

Statement:

```
SELECT EMPNO, LASTNAME, WORKDEPT
FROM   TEMPL
WHERE  BRTHDATE <ANY ( SELECT BRTHDATE
                       FROM TMPL
                       WHERE WORKDEPT = 'E21' )
```

This statement does *not* find what is required from the original query, at least as that query would normally be interpreted in colloquial English. What is *really* required from the query are the employees who are younger than *all* members of department E21. What the statement does is find all employees who are younger than *at least one* employee in E21, which is the strict logical definition.

As far as the redundancy is concerned, the WHERE clause

```
WHERE x $ANY ( SELECT y FROM T WHERE p )
```

(where *$* is any one of <, > or =) is equivalent to the WHERE clause

```
WHERE EXISTS ( SELECT * FROM T WHERE (p) AND x $ T.y )
```

Similarly, the WHERE clause

```
WHERE x $ALL ( SELECT y FROM T WHERE p )
```

is equivalent to

```
WHERE NOT EXISTS ( SELECT * FROM T WHERE (p)
                                    AND NOT ( x $ T.y ) )
```

Appendix C

Test Data

C.1 Introduction

This appendix describes the database that the system was tested with. The database encodes the family tree shown in Figure C-1. The second section describes the structure of the database, the third has some sample output from SQUALID and the fourth shows how the rules are actually stored in the database.

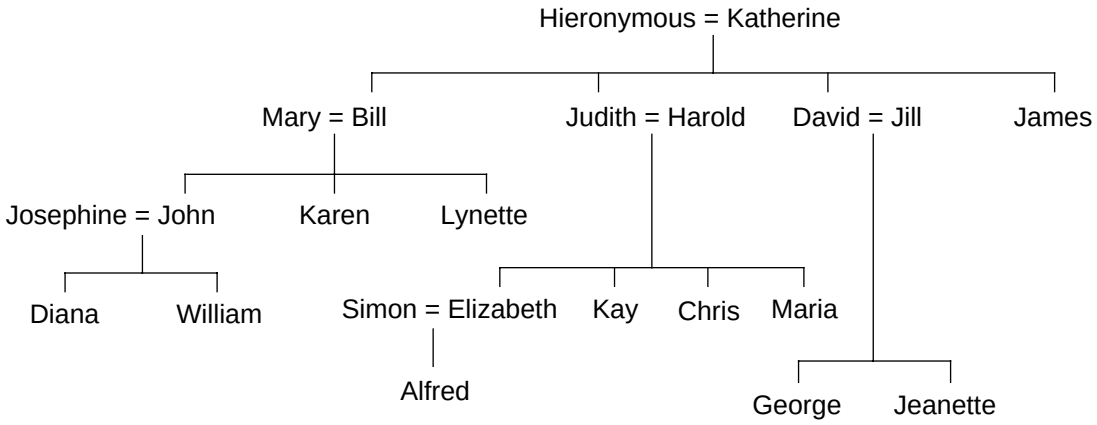


Figure C-1. Family tree which the sample database represents.

C.2 Database Structure

The database consists of three base tables, *Parent*, *Male* and *Female*, and a number of virtual tables, including *Father*, *Mother*, *Grandparent*, and several others. The next three sections list, respectively:

- the original SQL schema definition for the test database (note that this does not include any rule definitions)
- the rule base definitions, in both old SQLD format and first-order logic notation
- the equivalent SQLD schema definition corresponding to the above two items

C.2.1 Database Schema Definition (SQL)

```
!*****************************************************************************
!*                                                                           *
!* TESTDB.SQL (27-JUL-1990)                                                  *
!*                                                                           *
!* Defines a small test database for use with the SQLD system.               *
!* The database consists of three tables which encode a small family tree.   *
!* Facility is VAX Rdb/VMS Version 3.1                                       *
!*                                                                           *
!*****************************************************************************
!
! Note this assumes an already existing schema!
!
!**************************************
! Create PARENT relation
!**************************************
!
create table parent
(
   parent char(20),
   child  char(20),

   primary key ( parent, child )
);
!
!**************************************
! Create MALE relation
!**************************************
!
create table male
(
   name char(20)        primary key
);
!
!**************************************
! Create FEMALE relation
!**************************************
!
create table female
(
   name char(20)        primary key
);
```

C.2.2 Rule Base

SQLD Rule Definition File (Old Format)

```
!*****************************************************************************
!*                                                                           *
!* DDB.RLS (27-JUL-1990)                                                     *
!*                                                                           *
!* Contains rule definitions for the database defined by TESTDB.SQL.         *
!*                                                                           *
!*****************************************************************************
!
!**************************************
! Define Parent rules (mother, father)
!**************************************
!
PARENT(_X:PARENT, _Y:CHILD) ^ MALE(_X:NAME)   -> FATHER(_X:FATHER, _Y:CHILD)
PARENT(_X:PARENT, _Y:CHILD) ^ FEMALE(_X:NAME) -> MOTHER(_X:MOTHER, _Y:CHILD)
```

```
!
!*************************************
! Define Grandparent rules (grandparent, grandfather, grandmother)
!*************************************
!
PARENT(_X:PARENT, _Y:CHILD) ^ PARENT(_Y:PARENT, _Z:CHILD)
                                    -> GRANDPARENT(_X:GPARENT, _Z:GCHILD)
FATHER(_X:FATHER, _Y:CHILD) ^ PARENT(_Y:PARENT, _Z:CHILD)
                                    -> GRANDFATHER(_X:GFATHER, _Z:GCHILD)
MOTHER(_X:MOTHER, _Y:CHILD) ^ PARENT(_Y:PARENT, _Z:CHILD)
                                    -> GRANDMOTHER(_X:GMOTHER, _Z:GCHILD)
!
!*************************************
! Define Child rules (child, son, daughter)
!*************************************
!
PARENT(_X:PARENT, _Y:CHILD)                    -> CHILD(_Y:CHILD, _X:PARENT)
PARENT(_X:PARENT, _Y:CHILD) ^ MALE(_Y:NAME)   -> SON(_Y:SON, _X:PARENT)
PARENT(_X:PARENT, _Y:CHILD) ^ FEMALE(_Y:NAME) -> DAUGHTER(_Y:DAUGHTER,
_X:PARENT)
!
!*************************************
! Define Sibling rules (sibling, brother, sister)
! Note: these are not well formulated!
!   e.g. parent(john, william) ^ parent(john, william) -> sibling(william, william)
!
! What is needed is a "not equal" predicate (among others), i.e.:
!
!       parent(x, y) ^ parent(x, z) ^ (y <> z) -> sibling(y, z)
!
! This is not yet available.
!*************************************
!
PARENT(_X:PARENT, _Y:CHILD) ^ PARENT(_X:PARENT, _Z:CHILD) -> SIBLING(_Y:SIB1, _Z:SIB2)
SIBLING(_X:SIBLINGA, _Y:SIBLINGB) ^ FEMALE(_X:NAME) -> SISTER(_X:SISTER, _Y:SIBLING)
SIBLING(_X:SIBLINGA, _Y:SIBLINGB) ^ MALE(_X:NAME)   -> BROTHER(_X:BROTHER, _Y:SIBLING)
!
!*************************************
! Define Grandchild rules (grandchild, grandson, granddaughter)
!*************************************
!
GRANDPARENT(_X:GPARENT, _Y:GCHILD)      -> GRANDCHILD(_Y:GCHILD, _X:GPARENT)
GRANDPARENT(_X:GPARENT, _Y:GCHILD) ^ MALE(_Y:NAME)
                                    -> GRANDSON(_Y:GSON, _X:GPARENT)
GRANDPARENT(_X:GPARENT, _Y:GCHILD) ^ FEMALE(_Y:NAME)
                                    -> GRANDDAUGHTER(_Y:GDAUGHTER, _X:GPARENT)
!
!*************************************
! Define Great-grandparent rules (ggp, ggf, ggm)
!*************************************
!
PARENT(_X:PARENT, _Y:CHILD) ^ GRANDPARENT(_Y:GPARENT, _Z:GCHILD)
                                        -> GGP(_X:GGPARENT, _Z:GGCHILD)
FATHER(_X:FATHER, _Y:CHILD) ^ GRANDPARENT(_Y:GPARENT, _Z:GCHILD)
                                        -> GGF(_X:GGFATHER, _Z:GGCHILD)
MOTHER(_X:MOTHER, _Y:CHILD) ^ GRANDPARENT(_Y:GPARENT, _Z:GCHILD)
                                        -> GGM(_X:GGMOTHER, _Z:GGCHILD)
!
!*************************************
! Define Cousin rule
! Doesn't work properly: see Sibling rules above
!*************************************
!
PARENT(_X:PARENT, _Y:CHILD) ^ PARENT(_A:PARENT, _B:CHILD) ^ SIBLING(_X:SIB1, _A:SIB2)
                                        -> COUSIN(_Y:COUSIN1, _B:COUSIN2)
```

First-Order Logic Notation

Parent rules:

$parent(x, y) \wedge male(x) \rightarrow father(x, y)$
$parent(x, y) \wedge female(x) \rightarrow mother(x, y)$

Grandparent rules:

$parent(x, y) \wedge parent(y, z) \rightarrow grandparent(x, z)$
$father(x, y) \wedge parent(y, z) \rightarrow grandfather(x, z)$
$mother(x, y) \wedge parent(y, z) \rightarrow grandmother(x, z)$

Child rules:

$parent(x, y) \rightarrow child(y, x)$
$parent(x, y) \wedge male(y) \rightarrow son(y, x)$
$parent(x, y) \wedge female(y) \rightarrow daughter(y, x)$

Sibling rules:

$parent(x, y) \wedge parent(x, z) \wedge (y \neq z) \rightarrow sibling(y, z)$ [Note inclusion of $y \neq z$]
$sibling(x, y) \wedge female(x) \rightarrow sister(x, y)$
$sibling(x, y) \wedge male(x) \rightarrow brother(x, y)$

Grandchild rules:

$grandparent(x, y) \rightarrow grandchild(y, x)$
$grandparent(x, y) \wedge male(y) \rightarrow grandson(y, x)$
$grandparent(x, y) \wedge female(y) \rightarrow granddaughter(y, x)$

Great-grandparent rules:

$parent(x, y) \wedge grandparent(y, z) \rightarrow ggp(x, z)$
$father(x, y) \wedge grandparent(y, z) \rightarrow ggf(x, z)$
$mother(x, y) \wedge grandparent(y, z) \rightarrow ggm(x, z)$

Cousin rules:

$parent(x, y) \wedge parent(a, b) \wedge sibling(x, a) \rightarrow cousin(y, b)$

C.2.3 Database Schema Definition (SQLD)

```
!*****************************************************************************
!*                                                                           *
!* TESTDB.SQLD (27-JUL-1990)                                                 *
!*                                                                           *
!* Defines a small test database for use with the SQLD system.               *
!* The database consists of three base tables and 18 virtual tables, which   *
!* encode a small family tree.                                               *
!* Facility is VAX Rdb/VMS Version 3.1 extended with SQLD                    *
!*                                                                           *
!*****************************************************************************
!
```

```
!**************************************
! Create PARENT relation
!**************************************
!
create table parent
(
   parent char(20),
   child  char(20),

   primary key ( parent, child )
);
!
!**************************************
! Create MALE relation
!**************************************
!
create table male
(
   name char(20)        primary key
);
!
!**************************************
! Create FEMALE relation
!**************************************
!
create table female
(
   name char(20)        primary key
);
!
!**************************************
! Create FATHER relation
!**************************************
!
create table father
(
   father char(20),
   child  char(20),

   primary key ( father, child )
)
from rule
   if   parent(x, y) and male(x)
   then father(x, y);
!
!**************************************
! Create MOTHER relation
!**************************************
!
create table mother
(
   mother char(20),
   child  char(20),

   primary key ( mother, child )
)
from rule
   if   parent(x, y) and female(x)
   then mother(x, y);
!
```

```
!**************************************
! Create GRANDPARENT relation
!**************************************
!
create table grandparent
(
   gparent char(20),
   gchild  char(20),

   primary key ( gparent, gchild )
)
from rule
   if   parent(x, y) and parent(y, z)
   then grandparent(x, z);
!
!**************************************
! Create GRANDFATHER relation
!**************************************
!
create table grandfather
(
   gfather char(20),
   gchild  char(20),

   primary key ( gfather, gchild )
)
from rule
   if   father(x, y) and parent(y, z)
   then grandfather(x, z);
!
!**************************************
! Create GRANDMOTHER relation
!**************************************
!
create table grandmother
(
   gmother char(20),
   gchild  char(20),

   primary key ( gmother, gchild )
)
from rule
   if   mother(x, y) and parent(y, z)
   then grandmother(x, z);
!
!**************************************
! Create CHILD relation
!**************************************
!
create table child
(
   child  char(20),
   parent char(20),

   primary key ( child, parent )
)
from rule
   if   parent(x, y)
   then child (y, x);
!
```

```
!**************************************
! Create SON relation
!**************************************
!
create table son
(
   son    char(20),
   parent char(20),

   primary key ( son, parent )
)
from rule
   if   parent(x, y) and male(y)
   then son(y, x);
!
!**************************************
! Create DAUGHTER relation
!**************************************
!
create table daughter
(
   daughter char(20),
   parent   char(20),

   primary key ( daughter, parent )
)
from rule
   if   parent(x, y) and female(y)
   then daughter(y, x);
!
!**************************************
! Create SIBLING relation
!**************************************
!
create table sibling
(
   siblinga char(20),
   siblingb char(20),

   primary key ( siblinga, siblingb )
)
from rule
   if   parent(x, y) and parent(x, z) and (y <> z)
   then sibling(y, z);
!
!**************************************
! Create SISTER relation
!**************************************
!
create table sister
(
   sister  char(20),
   sibling char(20),

   primary key ( sister, sibling )
)
from rule
   if   sibling(x, y) and female(x)
   then sister(x, y);
!
```

```
!**************************************
! Create BROTHER relation
!**************************************
!
create table brother
(
   brother char(20),
   sibling char(20),

   primary key ( brother, sibling )
)
from rule
   if   sibling(x, y) and male(x)
   then brother(x, y);
!
!**************************************
! Create GRANDCHILD relation
!**************************************
!
create table grandchild
(
   gchild  char(20),
   gparent char(20),

   primary key ( gchild, gparent )
)
from rule
   if   grandparent(x, y)
   then grandchild(y, x);
!
!**************************************
! Create GRANDSON relation
!**************************************
!
create table grandson
(
   gson    char(20),
   gparent char(20),

   primary key ( gson, gparent )
)
from rule
   if   grandparent(x, y) and male(y)
   then grandson(y, x);
!
!**************************************
! Create GRANDDAUGHTER relation
!**************************************
!
create table granddaughter
(
   gdaughter char(20),
   gparent   char(20),

   primary key ( gdaughter, gparent )
)
from rule
   if   grandparent(x, y) and female(y)
   then granddaughter(y, x);
!
```

```
!**************************************
! Create GGP (great-grandparent) relation
!**************************************
!
create table ggp
(
   ggparent char(20),
   ggchild  char(20),

   primary key ( ggparent, ggchild )
)
from rule
   if   parent(x, y) and grandparent(y, z)
   then ggp(x, z);
!
!**************************************
! Create GGF (great-grandfather) relation
!**************************************
!
create table ggf
(
   ggfather char(20),
   ggchild  char(20),

   primary key ( ggfather, ggchild )
)
from rule
   if   father(x, y) and grandparent(y, z)
   then ggf(x, z);
!
!**************************************
! Create GGM (great-grandmother) relation
!**************************************
!
create table ggm
(
   ggmother char(20),
   ggchild  char(20),

   primary key ( ggmother, ggchild )
)
from rule
   if   mother(x, y) and grandparent(y, z)
   then ggm(x, z);
!
!**************************************
! Create COUSIN relation
!**************************************
!
create table cousin
(
   cousina char(20),
   cousinb char(20),

   primary key ( cousina, cousinb )
)
from rule
   if   parent(x, y) and parent(a, b) and sibling(x, a)
   then cousin(y, b);
```

C.3 Sample Output from SQUALID

The following is an example of a session with SQUALID. User input is shown in **bold face**. Comments are in *italics*.

```
$ run squalid
SQLD> declare schema filename ddb;  open database
SQLD> show tables                   list all base and virtual tables
PARENT
MALE
FEMALE
TRANSITION
PTP
FATHER
MOTHER
GRANDPARENT
GRANDFATHER
GRANDMOTHER
CHILD
SON
DAUGHTER
SIBLING
SISTER
BROTHER
GRANDCHILD
GRANDSON
GRANDDAUGHTER
GGP
GGF
COUSIN
SQLD> select *
      from parent;                  select from a base table
SELECT
   PARENT.PARENT                    this is a debugging message
   PARENT.CHILD                     generated by SQUALID to
FROM                                display the modified statement
  PARENT

Hieronymous         Bill
Hieronymous         Judith
Hieronymous         David
Hieronymous         James
Katherine           Bill
Katherine           Judith
Katherine           David
Katherine           James
Mary                John
Mary                Karen
Mary                Lynette
Bill                John
Bill                Karen
Bill                Lynette         continued over...
```

```
John                 William
John                 Diana
Josephine            Diana
Josephine            William
Judith               Elizabeth
Judith               Kay
Judith               Chris
Judith               Maria
Harold               Elizabeth
Harold               Kay
Harold               Chris
Harold               Maria
Simon                Alfred
Elizabeth            Alfred
David                George
David                Jeanette
Jill                 George
Jill                 Jeanette
SQLD> select *
      from grandparent;              select from a virtual table
SELECT
   P1.PARENT
   PARENT.CHILD
FROM
  PARENT
  PARENT P1
WHERE
   PARENT.PARENT = P1.CHILD


Katherine            Lynette
Hieronymous          Lynette
Katherine            John
Hieronymous          John
Katherine            Karen
Hieronymous          Karen
Katherine            Jeanette
Hieronymous          Jeanette
Katherine            George
Hieronymous          George
Judith               Alfred
Harold               Alfred
Bill                 Diana
Mary                 Diana
Bill                 William
Mary                 William
Katherine            Elizabeth
Hieronymous          Elizabeth
Katherine            Maria
Hieronymous          Maria
Katherine            Chris
Hieronymous          Chris
Katherine            Kay
Hieronymous          Kay
```

```
SQLD> select distinct gchild
      from grandmother
      order by gchild;
```

select from a virtual table with modifiers

```
SELECT
   P1.CHILD
FROM
  FEMALE
  PARENT
  PARENT P1
WHERE
   FEMALE.NAME = PARENT.PARENT
ORDER BY
   P1.CHILD ASC

Alfred
Bill
Chris
David
Diana
Elizabeth
George
James
Jeanette
John
Judith
Karen
Kay
Lynette
Maria
William
SQLD> select *
      from ggp;
```

select from another virtual table with a more complex rule definition

```
SELECT
   P2.PARENT
   PARENT.CHILD
FROM
  PARENT
  PARENT P1
  PARENT P2
WHERE
   PARENT.PARENT = P1.CHILD
   AND  P1.PARENT = P2.CHILD


Katherine           William
Hieronymous         William
Katherine           Diana
Hieronymous         Diana
Katherine           Alfred
Hieronymous         Alfred
```

```
SQLD> create table ggm                    create a virtual table
        ( ggmother char(20),              (great-grandmother rule)
          ggchild char(20)  )
      from rule
        if mother(x,y) and grandparent(y,z)
        then ggm(x,z);
SQLD> commit;
SQLD> show tables
PARENT
MALE
FEMALE
TRANSITION
PTP
FATHER
MOTHER
GRANDPARENT
GRANDFATHER
GRANDMOTHER.
CHILD
SON
DAUGHTER
SIBLING
SISTER
BROTHER
GRANDCHILD
GRANDSON
GRANDDAUGHTER
GGP
GGF
COUSIN
GGM                                       ggm table has been created
SQLD> select *
      from ggm;                           have a look at it
SELECT
   FEMALE.NAME
   PARENT.CHILD
FROM
  FEMALE
  PARENT
  PARENT P1
  PARENT P2
WHERE
   FEMALE.NAME = P2.PARENT
   AND  PARENT.PARENT = P1.CHILD
   AND  P1.PARENT = P2.CHILD


Katherine           Alfred
Katherine           Diana
Katherine           William
SQLD> quit                                end of session
```

C.4 How the Rules are Stored

The following two sections show the contents of the TRANSITION and PTP relations for the database used in the example in Section C.3.

C.4.1 TRANSITION Relation

TRANS_ID	TRANS_SCRIPT
PFM14-DEC-1990 14:40:40.48	1.1=2.1=-1.1,1.2=-1.2,
PMF14-DEC-1990 11:45:08.77	1.1=2.1=-1.1,1.2=-1.2,
PPG27-JAN-1991 09:53:46.89	1.1=-1.1,1.2=2.1,2.2=-1.2,
FPG27-JAN-1991 10:20:51.68	1.1=-1.1,1.2=2.1,2.2=-1.2,
MPG27-JAN-1991 10:22:50.89	1.1=-1.1,1.2=2.1,2.2=-1.2,
PC27-JAN-1991 10:25:08.53	1.1=-1.2,1.2=-1.1,
PMS27-JAN-1991 10:26:22.83	1.1=-1.2,1.2=2.1=-1.1,
PFD27-JAN-1991 10:27:06.51	1.1=-1.2,1.2=2.1=-1.1,
GG27-JAN-1991 10:27:47.46	1.1=-1.2,1.2=-1.1,
GMG27-JAN-1991 10:28:36.49	1.1=-1.2,1.2=2.1=-1.1,
GFG27-JAN-1991 10:29:27.77	1.1=-1.2,1.2=2.1=-1.1,
PGG27-JAN-1991 10:30:14.24	1.1=-1.1,1.2=2.1,2.2=-1.2,
FGG27-JAN-1991 10:30:57.67	1.1=-1.1,1.2=2.1,2.2=-1.2,
MGG27-JAN-1991 11:25:36.27	1.1=-1.1,1.2=2.1,2.2=-1.2,

C.4.2 PTP Relation

TRANS_ID	PREPLACE	PREPOSITION	PRECOLUMNS	POSTPLACE
PFM14-DEC-1990 14:40:40.48	PARENT	1	2	MOTHER
PFM14-DEC-1990 14:40:40.48	FEMALE	2	1	MOTHER
PMF14-DEC-1990 11:45:08.77	PARENT	1	2	FATHER
PMF14-DEC-1990 11:45:08.77	MALE	2	1	FATHER
PPG27-JAN-1991 09:53:46.89	PARENT	1	2	GRANDPARENT
PPG27-JAN-1991 09:53:46.89	PARENT	2	2	GRANDPARENT
FPG27-JAN-1991 10:20:51.68	FATHER	1	2	GRANDFATHER
FPG27-JAN-1991 10:20:51.68	PARENT	2	2	GRANDFATHER
MPG27-JAN-1991 10:22:50.89	MOTHER	1	2	GRANDMOTHER
MPG27-JAN-1991 10:22:50.89	PARENT	2	2	GRANDMOTHER
PC27-JAN-1991 10:25:08.53	PARENT	1	2	CHILD
PMS27-JAN-1991 10:26:22.83	PARENT	1	2	SON
PMS27-JAN-1991 10:26:22.83	MALE	2	1	SON
PFD27-JAN-1991 10:27:06.51	PARENT	1	2	DAUGHTER
PFD27-JAN-1991 10:27:06.51	FEMALE	2	1	DAUGHTER
GG27-JAN-1991 10:27:47.46	GRANDPARENT	1	2	GRANDCHILD
GMG27-JAN-1991 10:28:36.49	GRANDPARENT	1	2	GRANDSON
GMG27-JAN-1991 10:28:36.49	MALE	2	1	GRANDSON
GFG27-JAN-1991 10:29:27.77	GRANDPARENT	1	2	GRANDDAUGHTER
GFG27-JAN-1991 10:29:27.77	FEMALE	2	1	GRANDDAUGHTER
PGG27-JAN-1991 10:30:14.24	PARENT	1	2	GGP
PGG27-JAN-1991 10:30:14.24	GRANDPARENT	2	2	GGP
FGG27-JAN-1991 10:30:57.67	FATHER	1	2	GGF
FGG27-JAN-1991 10:30:57.67	GRANDPARENT	2	2	GGF
MGG27-JAN-1991 11:25:36.27	MOTHER	1	2	GGM
MGG27-JAN-1991 11:25:36.27	GRANDPARENT	2	2	GGM

Appendix D

Source Code Listings

D.1 Introduction

This Appendix contains the source code for SQUALID. A brief description of each of the files follows:

SQUALID.PAS: Main program loop (Section D.2).

GLOBAL.PAS: Global declarations (Section D.3).

LISTOPS.PAS: List-related operations (Section D.4).

RULES.PAS: Routines for manipulating rules. Includes core routines for deduction (Section D.5).

SQLPARSE.YAC: YACC description of a SQLD parser (Section D.6).

DYNAMIC.PAS: Pascal routines which handle the interface with Rdb/VMS (Section D.7).

SQLDYN.SQLMOD: SQL module language procedures which interface with Rdb/VMS (Section D.8).

SQUALIDMSG.MSG: VAX MESSAGE UTILITY source file for SQUALID's error messages (Section D.9).

DESCRIP.MMS: "MAKEFILE" for SQUALID (Section D.10).

D.2 SQUALID.PAS

```
[ INHERIT ('GLOBAL',
           'DYNAMIC',
           'SQLPARSE',
           'RULES')]


PROGRAM DEDUCTIVE_SQL (input,output);


(*****************************************************************************)
(* SQLD.PAS - N. Stanger, August 1989                                        *)
(*                                                                           *)
(* Main module for the SQLD system.                                          *)
(*****************************************************************************)


VAR     finished               : boolean;
        stmt                   : st;
        part_stmt              : string;
        begin_pos, end_pos, pos : integer;
        end_of_statement       : boolean;
        last, i                : integer;
        theColumns             : StringPair_ptr;  (* names of columns for output *)


BEGIN
  finished   := false;
  theColumns := nil;

(*
  write('SQLD> ');
  reset(input);
  while ((not finished) and (not eof(input))) do
  begin
    for i := 1 to 1024 do stmt[i] := ' ';
    begin_pos := 1;
    end_of_statement := false;
    while ((not end_of_statement) and (not eof(input))) do
    begin
      readln(part_stmt,error := continue);
      last := length(part_stmt);
      if (last > 0) then
      begin
        end_pos := (begin_pos + last) - 1;
        for i := 1 to last do stmt[i - 1 + begin_pos] := part_stmt[i];
        stmt[end_pos + 1] := ' ';
        begin_pos := (begin_pos + last) + 1;
        if part_stmt[last] = ';' then
        begin
          stmt[end_pos] := ' ';
          end_of_statement := true;
          write('SQLD> ');
        end
        else write('Cont> ');
      end
      else write('SQLD> ');
    end;
    pre_process(stmt);
    send_to_sql(stmt);
  end;
*)

  while (not finished) do
  begin
    write('SQLD> ');
    readln(part_stmt);
    stmt := part_stmt;
    if pre_process(stmt, theColumns) then
      send_to_sql(stmt, theColumns);
  end;
END.    (* DEDUCTIVE_SQL *)
```

D.3 GLOBAL.PAS

```
[ INHERIT ('SYS$LIBRARY:PASCAL$LIB_ROUTINES',
           'SYS$LIBRARY:PASCAL$STR_ROUTINES',
           'SYS$LIBRARY:STARLET'),
  ENVIRONMENT ('GLOBAL') ]


MODULE GLOBAL (input,output);


(*****************************************************************************)
(* GLOBAL.PAS - N. Stanger, August 1989                                      *)
(*                                                                           *)
(* This module contains all globally-defined data types, variables and rou-  *)
(* tines.                                                                    *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  22-MAY-1990 Changed PLIST structure to include both SQL names for columns *)
(*              and rule names for columns.                                  *)
(*****************************************************************************)


CONST  max_strlng = 1024;
       uppercase  = ['A'..'Z'];
       lowercase  = ['a'..'z'];
       alpha      = lowercase + uppercase;
       numeric    = ['0'..'9'];
       alphanum   = alpha + numeric;
       id_chars   = alphanum + ['$','_'];
       quotes     = ['"',''''];
       semicolon  = ';';
       smiley     = ':-)';


TYPE   String = varying[255] of char;

       BigString      = varying[max_strlng] of char;
       StringPair_ptr = ^StringPair;  (* a linked list of pairs of strings *)
       StringPair     = record
                          str1, str2 : string;
                          next       : StringPair_ptr;
                        end;
       (* when used for storing Rule and SQL column names: *)
       (*       str1 = SQL column name                     *)
       (*       str2 = Rule column name                    *)
       (*                                                  *)
       (* when used for storing virtual column names:      *)
       (*       str1 = virtual table name                  *)
       (*       str2 = virtual column name                 *)
       (*                                                  *)
       (* when used for storing identical variables:       *)
       (*       str1 = predicate position                  *)
       (*       str2 = parameter position                  *)

       String_ptr  = ^String_list;
       String_list = record
                       theStr : string;
                       next   : String_ptr;
                     end;

       str255      = packed array[1..255] of char;
       st          = packed array[1..max_strlng] of char;
       varchar     = varying[max_strlng] of char;
       date_string = packed array[1..23] of char;
       $word       = [WORD] -32768..32767;
       $uword      = [WORD] 0..65535;
       $uquad      = [QUAD,UNSAFE] record
                                     l0, l1 : unsigned;
                                   end;
       date_time   = [VOLATILE] $uquad;
       table_ptr   = ^table_list;
       table_list  = record
                       table_name : string;
                       columns    : String_ptr;
                       next       : table_ptr;
                     end;

       (* data structures for storing rules follow *)
       ptypes      = (var_name,const_name);

       value_ptr   = ^valuelist;  (* values are stored in these *)
       valuelist   = record
                       next, prev  : value_ptr;  (* need to search both ways *)
                       pkind       : ptypes;
                       value       : string;
                       purgeable   : boolean;    (* used to purge value list *)
                     end;

       plist_ptr   = ^plist;   (* a linked list of pointers to parameter values *)
       plist       = record
                       next       : plist_ptr;
                       RuleColumn : string;     (* Rule's name for the column *)
                       SQLColumn  : string;     (* SQL name of the column *)
                       valueRef   : value_ptr;  (* ptr to actual value *)
                     end;

       pred_ptr    = ^pred;
       pred        = record
                       next       : pred_ptr;
                       name       : string;
                       arity      : integer;
                       negated    : boolean;
                       parameters : plist_ptr;
                     end;
```

```
       clause_ptr  = ^clause;  (* CLAUSEs store clauses in a linked list *)
       clause      = record
                       next        : clause_ptr;
                       left, right : pred_ptr;
                       tableName   : string;     (* stores SQL "table" name for this rule *)
                     end;

       instant_ptr = ^instant_rec;  (* store variable instantiations *)
       instant_rec = record
                       next                : instant_ptr;
                       old_value, new_value : string;
                       old_kind, new_kind   : ptypes;
                     end;


        PCN_ptr    = ^PCN;
        PCN        = record
                       id       : String;  (* transition id *)
                       prePlace : String;  (* predicate name *)
                       preCols  : integer; (* number of columns *)
                       next     : PCN_ptr;
                     end;


VAR    base_tables       : table_ptr value nil;
       virt_tables       : table_ptr value nil;
       this_rule         : string;               (* current rule *)
       scan_pos          : integer;              (* position of parser in current rule *)
       rulelength        : integer;              (* length of current rule *)
       the_rules         : clause_ptr value nil;  (* the global rule list *)
       the_query         : clause_ptr value nil;  (* stores queries *)
       theValues         : value_ptr value nil;   (* global list of parameter values *)
       rulefile          : text;
       reading_from_file : boolean;
       numVars           : integer value 0;       (* how many variables have been created *)

       DDB_NORULES,      (* Message IDs *)
       DDB_BADFILERULE,
       DDB_BADINPUTRULE,
       DDB_NOTMATCHED,
       DDB_NOARITY,
       DDB_NOCOMBINE,
       DDB_BADINSTANT,
       DDB_TOOMANYVARS,
       DDB_RULEEXIST,
       DDB_NOTABLE,
       DDB_UNEXP,
       DDB_TABNOTFD,
       DDB_BADSQLD,
       DDB_ABORT         : [value, external] integer;


(*****************************************************************************)
(* MESSAGE                                                                   *)
(* Uses the $GETMSG system service to display messages from a message file   *)
(* (refer to VAX/VMS Message utility documentation)                           *)
(*****************************************************************************)
Procedure MESSAGE ( msgid : integer; p1, p2, p3, p4 : string := '' );
Var       ctrstr, msgtxt : string;
          fao_parms      : packed array[1..8] of integer;
Begin
  fao_parms[1] := length(p1);
  fao_parms[2] := IADDRESS(p1.body);
  fao_parms[3] := length(p2);
  fao_parms[4] := IADDRESS(p2.body);
  fao_parms[5] := length(p3);
  fao_parms[6] := IADDRESS(p3.body);
  fao_parms[7] := length(p4);
  fao_parms[8] := IADDRESS(p4.body);
  $getmsg(msgid, ctrstr.length, ctrstr.body, , );
  $faol(ctrstr, msgtxt.length, msgtxt.body, fao_parms);
  writeln(msgtxt);
end;    (* MESSAGE *)


(*****************************************************************************)
(* SCREAM (19-FEB-1990)                                                      *)
(* Fatal error routine.                                                      *)
(*****************************************************************************)
Procedure SCREAM ( msgid : integer; p1, p2, p3, p4 : string := '' );
Begin
  message(msgid, p1, p2, p3, p4);
  $exit(DDB_ABORT);
end; (* SCREAM *)


(*****************************************************************************)
(* NUM2STR (09-FEB-1990)                                                     *)
(* Converts a positive integer into a string.                                *)
(*****************************************************************************)
Function NUM2STR ( num : integer ) : String;
Begin
  if (num < 10) then Num2Str := chr(num + ord('0'))
  else Num2Str := Num2Str(num div 10) + chr(num mod 10 + ord('0'));
end; (* NUM2STR *)


(*****************************************************************************)
(* NUMTOSTRING (09-FEB-1990)                                                 *)
(* Converts any integer into a string.                                       *)
(*****************************************************************************)
Function NumToString ( num : integer ) : String;
Var      tempStr : String;
Begin
  if (num >= 0) then NumToString := Num2Str(abs(num))
  else NumToString := '-' + Num2Str(abs(num));
end; (* NUMTOSTRING *)
```

```
(*************************************************************************)
(* UPCASE (12-MAR-1990)                                                  *)
(* Converts a VARYING string to uppercase.                               *)
(*************************************************************************)
Function UPCASE ( theStr : string ) : string;
Var     i : integer;
Begin
  for i:= 1 to length(theStr) do
  begin
    if (theStr[i] in lowercase) then
      theStr[i] := chr(ord(theStr[i]) - 32);
  end;
  upcase := theStr;
end; (* UPCASE *)


(*************************************************************************)
(* APPEND_STRINGPAIR (11-DEC-1990)                                       *)
(* Given two strings, appends them to the end of a StringPair_ptr list   *)
(*************************************************************************)
Function APPEND_STRINGPAIR ( string1, string2 : String;
                             theList          : StringPair_ptr ) : StringPair_ptr;
Var      temp1, temp2 : StringPair_ptr;
Begin
  if (theList = nil) then
  begin
    new(theList);
    theList^.str1 := string1;
    theList^.str2 := string2;
    theList^.next := nil;
  end
  else
  begin
    temp1 := theList;
    while (temp1^.next <> nil) do temp1 := temp1^.next;
    new(temp2);
    temp2^.str1 := string1;
    temp2^.str2 := string2;
    temp2^.next := nil;
    temp1^.next := temp2;
  end;
  Append_StringPair := theList;
  temp1             := nil;
  temp2             := nil;
end; (* APPEND_STRINGPAIR *)


(*************************************************************************)
(* KILLSTRINGPAIR (13-DEC-1990)                                          *)
(* Destroys a StringPair_ptr list and deallocates all memory allocated to it. *)
(*************************************************************************)
Procedure KILLSTRINGPAIR ( theList : StringPair_ptr );
Begin
  if (theList <> nil) then
  begin
    KillStringPair(theList^.next);
    dispose(theList);
    theList := nil;
  end;
end; (* KILLSTRINGPAIR *)


(*************************************************************************)
(* KILLSTRING (13-DEC-1990)                                              *)
(* Destroys a String_ptr list and deallocates all memory allocated to it.     *)
(*************************************************************************)
Procedure KILLSTRING ( theList : String_ptr );
Begin
  if (theList <> nil) then
  begin
    KillString(theList^.next);
    dispose(theList);
    theList := nil;
  end;
end; (* KILLSTRING *)


(*************************************************************************)
(* KILLTABLES (12-DEC-1990)                                              *)
(* Destroys a Table_ptr list and deallocates all memory allocated to it.      *)
(*************************************************************************)
Procedure KILLTABLES ( var theList : Table_ptr );
Begin
  if (theList <> nil) then
  begin
    KillTables(theList^.next);
    KillString(theList^.columns);
    dispose(theList);
    theList := nil;
  end
end; (* KILLTABLES *)
```

```
(*****************************************************************************)
(* STRINGPTR_TO_STRINGPAIRPTR (18-JUN-1990)                                  *)
(* Converts a String_ptr list into a StringPair_ptr list.                    *)
(*****************************************************************************)
Function STRINGPTR_TO_STRINGPAIRPTR ( inList : String_ptr ) : StringPair_ptr;
Var      tempS          : String_ptr;
         tempP1, tempP2 : StringPair_ptr;
         outList        : StringPair_ptr;
Begin
  tempS   := inList;
  outList := nil;
  while (tempS <> nil) do with tempS^ do
  begin
    if (outList = nil) then
    begin
      new(outList);
      outList^.str1 := theStr;
      outList^.str2 := '';
      outList^.next := nil;
    end
    else
    begin
      tempP1 := outList;
      while (tempP1^.next <> nil) do tempP1 := tempP1^.next;
      new(tempP2);
      tempP2^.str1 := theStr;
      tempP2^.str2 := '';
      tempP2^.next := nil;
      tempP1^.next := tempP2;
    end;
    tempS := tempS^.next;
  end;
  StringPtr_To_StringPairPtr := outList;
  tempS  := nil;
  tempP1 := nil;
  tempP2 := nil;
end; (* STRINGPTR_TO_STRINGPAIRPTR *)


END.    (* MODULE GLOBAL *)
```

D.4 LISTOPS.PAS

```
[ INHERIT ('GLOBAL',
           'SYS$LIBRARY:STARLET'),
  ENVIRONMENT ('LISTOPS') ]


Module LISTOPS (input,output);


(*************************************************************************)
(*                                                                       *)
(* LISTOPS.PAS - N. Stanger, March/April 1989                            *)
(*                                                                       *)
(* Contains list, tree and other assorted pointer operations             *)
(*                                                                       *)
(* 'D+' and 'D-' delimit debugging code                                  *)
(*                                                                       *)
(* MODIFIED:                                                             *)
(*  26-MAY-1989 Major rewrite to incorporate new and improved data structures *)
(*              Most routines rewritten                                  *)
(*  12-MAR-1990 Implemented functions for building the SQL definition list    *)
(*              for rules                                                *)
(*************************************************************************)


CONST  maxVars = maxint - 10; (* maximum number of variables allowed *)


VAR    querying : boolean := false;  (* is this a query or not *)


(*************************************************************************)
(* PRINT_PARAMETERS                                                      *)
(* Diagnostic routine - prints out a predicates parameter list           *)
(*************************************************************************)
Procedure PRINT_PARAMETERS (list : plist_ptr);
Var       sep : string;
Begin
  if (list <> nil) then with list^ do
  begin
    if (next <> nil) then sep := ',' else sep := '';
    write(valueRef^.value, sep);
    print_parameters(next);
  end;
end;    (* PRINT_PARAMETERS *)


(*************************************************************************)
(* DUMP_SIDE (26-MAR-1989)                                               *)
(* Diagnostic routine - prints out LH and RH sides of clauses            *)
(*************************************************************************)
Procedure DUMP_SIDE (side : pred_ptr; LHS : boolean);
Begin
  while (side <> nil) do with side^ do
  begin
    write(name,'(');
    print_parameters(parameters);
    write(')');
    if (next <> nil) then
    begin
      if LHS then write(' ^ ')  (* all predicates on LHS separated by ^'s *)
      else write(' | ');        (*  "       "       " RHS     "      " |'s *)
    end;
    side := side^.next;
  end;
end;    (* DUMP_SIDE *)


(*************************************************************************)
(* DUMP_RULE (26-MAY-1989)                                               *)
(* Diagnostic routine - prints out a single rule                         *)
(*************************************************************************)
Procedure DUMP_RULE (rule : clause_ptr);
Begin
  if (rule <> nil) then with rule^ do
  begin
    dump_side(left,true);
    write(' -> ');
    dump_side(right,false);
  end;
  writeln;
end;    (* DUMP_RULE *)


(*************************************************************************)
(* THESE_ARE                                                             *)
(* Diagnostic routine - prints out all rules in the list                 *)
(*************************************************************************)
Procedure THESE_ARE (the_rules : clause_ptr);
Begin
  while (the_rules <> nil) do with the_rules^ do
  begin
    write('>> ');
    dump_side(left,true);
    write(' -> ');
    dump_side(right,false);
    writeln;
    the_rules := the_rules^.next;
  end;
end;    (* THESE_ARE *)
```

```
(*****************************************************************************)
(* DUMP_INSTANT (05-FEB-1990)                                                *)
(* Prints out an instantiation list.                                         *)
(*****************************************************************************)
Procedure DUMP_INSTANT (instant : instant_ptr);
Var       temp : instant_ptr;
Begin
  temp := instant;
  write('{ ');
  while (temp <> nil) do with temp^ do
  begin
    write(old_value, '/', new_value);
    if (temp^.next <> nil) then write(', ');
    temp := temp^.next;
  end;
  writeln(' }');
end; (* DUMP_INSTANT *)


(*****************************************************************************)
(* DUMP_VALUES (15-FEB-1990)                                                 *)
(* Lists the global value list to the screen.                                *)
(*****************************************************************************)
Procedure DUMP_VALUES (valueList : value_ptr);
Var       temp : value_ptr;
Begin
  writeln('Value list currently contains:');
  temp := valueList;
  while (temp <> nil) do with temp^ do
  begin
    write('> ', value, ' (', pkind, ')');
    if purgeable then writeln(' *')
    else writeln;
    temp := temp^.next;
  end;
end; (* DUMP_VALUES *)


(*****************************************************************************)
(* RENUMBERVALUES (13-FEB-1990)                                              *)
(* Takes the current value list and replaces variable values (e.g. '_x') with *)
(* unique ids (e.g. '_0').                                                   *)
(*****************************************************************************)
Procedure RENUMBERVALUES;
Var       temp : value_ptr;
Begin
  temp := theValues;
  numVars := 0;
  while (temp <> nil) do with temp^ do
  begin
    if (numVars > maxVars) then  (* highly unlikely... *)
      scream(DDB_TOOMANYVARS);
    if (pkind = var_name) then
    begin
      value := '_' + NumToString(numVars);
      numVars := numVars + 1;
    end;
    temp := temp^.next;
  end;
end; (* RENUMBERVALUES *)


(*****************************************************************************)
(* PURGE_VALUES (15-FEB-1990)                                                *)
(* Purges the global value list of copied values that are no longer needed.  *)
(*****************************************************************************)
Procedure PURGE_VALUES;
Var       temp1, temp2 : value_ptr;
Begin
  temp1 := theValues;
  while (temp1 <> nil) do with temp1^ do
  begin
    if purgeable then
    begin
      if (prev = nil) then (* first item *)
      begin
        theValues := next;
        dispose(temp1);
        temp1 := theValues;
      end
      else if (next = nil) then (* last item *)
      begin
        prev^.next := nil;
        dispose(temp1);
        temp1 := nil;
      end
      else
      begin (* in the middle somewhere *)
        temp2      := next;
        prev^.next  := temp2;
        temp2^.prev := prev;
        dispose(temp1);
        temp1      := temp2;
      end;
    end
    else temp1 := temp1^.next;
  end;
end; (* PURGE_VALUES *)
```

```
(*************************************************************************)
(* SET_PURGE_PARAMETER (15-FEB-1990)                                     *)
(* Subroutine of SET_PURGE_SIDE and SET_PURGEABLE.                       *)
(*************************************************************************)
Procedure SET_PURGE_PARAMETER ( thePred : plist_ptr );
Var       temp : plist_ptr;
Begin
  temp := thePred;
  while (temp <> nil) do
  begin
    temp^.valueRef^.purgeable := true;
    temp := temp^.next;
  end;
end; (* SET_PURGE_PARAMETER *)


(*************************************************************************)
(* SET_PURGE_SIDE (15-FEB-1990)                                          *)
(* Subroutine of SET_PURGEABLE.                                          *)
(*************************************************************************)
Procedure SET_PURGE_SIDE ( theSide : pred_ptr );
Var       temp : pred_ptr;
Begin
  temp := theSide;
  while (temp <> nil) do
  begin
    set_purge_parameter(temp^.parameters);
    temp := temp^.next;
  end;
end; (* SET_PURGE_SIDE *)


(*************************************************************************)
(* SET_PURGEABLE (15-FEB-1990)                                           *)
(* Sets the values of a rule to be purgeable.                            *)
(*************************************************************************)
Procedure SET_PURGEABLE ( theRule : clause_ptr );
Begin
  set_purge_side(theRule^.left);
  set_purge_side(theRule^.right);
end; (* SET_PURGEABLE *)


(*************************************************************************)
(* ADD_RULE                                                              *)
(* Adds a rule to the global rule list (the_rules)                       *)
(*                                                                       *)
(* Called from: RULES.PAS : LOADRULE                                     *)
(*************************************************************************)
Procedure ADD_RULE ( rule : clause_ptr );
Var       temp : clause_ptr value nil;
Begin

(* D+
  writeln('In ADD_RULE, rule is: ');
  dump_rule(rule);
  writeln;
   D- *)

  if (rule <> nil) then
  begin
    if (rule^.right <> nil) then rule^.tableName  := rule^.right^.name;
    if (the_rules = nil) then
    begin
      the_rules       := rule;
      the_rules^.next := nil;
    end
    else
    begin
      temp := the_rules;
      while (temp^.next <> nil) do temp := temp^.next;
      temp^.next := rule;
    end;
    RenumberValues;

(* D+
    writeln('The rule list contains: ');
    these_are(the_rules);
   D- *)

  end;  (* if rule <> nil ... *)
  temp := nil;
end;    (* ADD_RULE *)


(*************************************************************************)
(* KILL_PARAMETERS (19-FEB-1990)                                         *)
(* Deletes the parameters of a rule.                                     *)
(*                                                                       *)
(* Called from: KILL_SIDE                                                *)
(*************************************************************************)
Procedure KILL_PARAMETERS ( theParameters : plist_ptr );
Var       temp, del : plist_ptr;
Begin
  if (theParameters <> nil) then
  begin
    temp := theParameters;
    while (temp <> nil) do
    begin
      del  := temp;
      temp := temp^.next;
      if (del^.valueRef <> nil) then del^.valueRef^.purgeable := true;
      dispose(del);
    end;
  end;
  temp := nil;
  del  := nil;
end; (* KILL_PARAMETERS *)
```

```
(*****************************************************************************)
(* KILL_SIDE (19-FEB-1990)                                                   *)
(* Deletes the predicates of a rule.                                         *)
(*                                                                           *)
(* Called from: KILL_RULE                                                    *)
(*****************************************************************************)
Procedure KILL_SIDE ( theSide : pred_ptr );
Var       temp, del : pred_ptr;
Begin
  if (theSide <> nil) then
  begin
    temp := theSide;
    while (temp <> nil) do
    begin
      del  := temp;
      temp := temp^.next;
      kill_parameters(del^.parameters);
      dispose(del);
    end;
  end;
  temp := nil;
  del  := nil;
end; (* KILL_SIDE *)


(*****************************************************************************)
(* KILL_RULE (19-FEB-1990)                                                   *)
(* Deletes a rule from the rule_list and releases the memory allocated to it. *)
(*****************************************************************************)
Procedure KILL_RULE ( theRule : clause_ptr );
Begin
  if (theRule <> nil) then
  begin
    kill_side(theRule^.left);
    kill_side(theRule^.right);
    dispose(theRule);
  end;
end; (* KILL_RULE *)


(*****************************************************************************)
(* REMOVE_LAST_RULE                                                          *)
(* Deletes the last rule from the rule list                                  *)
(*****************************************************************************)
Procedure REMOVE_LAST_RULE;
Var       temp1, temp2 : clause_ptr;
Begin
  if (the_rules <> nil) then
  begin
    temp1 := the_rules;
    temp2 := nil;
    while (temp1^.next <> nil) do
    begin
      temp2 := temp1;
      temp1 := temp1^.next;
    end;
    if (temp2 = nil) then
    begin
      kill_rule(the_rules);
      the_rules := nil;
    end
    else
    begin
      kill_rule(temp1);
      temp2^.next := nil;
    end;
  end;

(* D+
  writeln('In REMOVE_LAST_RULE, rule list contains:');
  these_are(the_rules);
   D- *)

end;    (* REMOVE_LAST_RULE *)


(*****************************************************************************)
(* MAKE_CLAUSE (formerly MAKETREE)                                           *)
(* Creates a CLAUSE node                                                     *)
(*****************************************************************************)
Function MAKE_CLAUSE (leftb, rightb : pred_ptr) : clause_ptr;
Var      temp : clause_ptr;
Begin
  new(temp);
  with temp^ do
  begin
    left       := leftb;
    right      := rightb;
    next       := nil;
  end;
  make_clause := temp;
  temp := nil;
end;    (* MAKE_CLAUSE *)
```

```
(*****************************************************************************)
(* ADD_PRED (27-MAY-1989)                                                    *)
(* Adds a PRED to the LHS or RHS list                                        *)
(*****************************************************************************)
Function ADD_PRED (pred_list, thePred : pred_ptr) : pred_ptr;
Var      temp : pred_ptr;
Begin
  if (pred_list = nil) then pred_list := thePred
  else
  begin
    temp := pred_list;
    while (temp^.next <> nil) do temp := temp^.next;
    temp^.next := thePred;
  end;
  add_pred := pred_list;
end;    (* ADD_PRED *)


(*****************************************************************************)
(* NEGATE (27-MAR-1989)                                                      *)
(* Negates a predicate definition (i.e. adds '~' on front)                   *)
(*****************************************************************************)
Function NEGATE (thePred : pred_ptr) : pred_ptr;
Begin
  if (thePred <> nil) then thePred^.negated := true;
  negate := thePred;
end;    (* NEGATE *)


(*****************************************************************************)
(* MAKE_PRED                                                                 *)
(* Creates a PRED node                                                       *)
(*****************************************************************************)
Function MAKE_PRED (pname  : string;
                    params : plist_ptr;
                    arty   : integer) : pred_ptr;
Var      temp : pred_ptr;
Begin

(* D+
  writeln('In MAKE_PRED.');
  write('Params is: ');
  print_parameters(params);
  writeln;
   D- *)

  new(temp);
  with temp^ do
  begin
    name       := pname;
    parameters := params;
    arity      := arty;
    negated    := false;    (* this is changed later if needed *)
    next       := nil;
  end;
  make_pred := temp;
  temp := nil;
end;    (* MAKE_PRED *)


(*****************************************************************************)
(* MAKE_VALUE                                                                *)
(* Creates a predicate parameter definition and returns a pointer to it. If a *)
(* definition of the parameter already exists, MAKE_VALUE returns a pointer   *)
(* to this instead.                                                          *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  24-APR-1989  Fixed bug with searching the parameter list                 *)
(*  12-FEB-1990  Changed name from MAKE_PARAMETER to MAKE_VALUE              *)
(*               Introduced global value list                                *)
(*  19-FEB-1990  Fixed purging bug                                           *)
(*                                                                           *)
(* Called from: ADD_PARAMETER                                                *)
(*              RULES.PAS : INSERT_COMMONVARS                                *)
(*              RULES.PAS : INSERT_VARS                                      *)
(*****************************************************************************)
Function MAKE_VALUE ( name : string ) : value_ptr;
Var      temp1, temp2, p : value_ptr;
         kind            : ptypes;
         found           : boolean;
         i               : integer;
Begin

(* D+
  writeln('In MAKE_VALUE, name = ',name);
   D- *)

  if (name[1] = '_') then kind := var_name
  else kind := const_name;
  if (theValues = nil) then
  begin

(* D+
    writeln('Parameter list empty - creating new list');
    writeln('- Type = ',kind,', name = ',name);
   D- *)

    new(theValues);
    with theValues^ do
    begin
      next      := nil;
      prev      := nil;
      pkind     := kind;
      value     := name;
      purgeable := false;
    end;
    make_value := theValues;
  end
```

```
  else
  begin
    found := false;
    temp1 := theValues;
    p     := nil;       (* prev pointer *)
    while ((temp1 <> nil) and (not found)) do with temp1^ do
    begin
      found := (value = name);

(* D+
      writeln('Searching parameter list - found = ',found);
      writeln('- Type = ',pkind,', name = ',value);
   D- *)

      p     := temp1;
      temp1 := temp1^.next;
    end;
    if (found and (not querying)) then make_value := p
    else
    begin

(* D+
      writeln('Creating new parameter definition:');
      writeln('- Type = ',kind,', name = ',name);
   D- *)

      new(temp2);
      with temp2^ do
      begin
        next      := temp1;
        prev      := p;
        pkind     := kind;
        value     := name;
        purgeable := false;
      end;
      if (temp1 <> nil) then temp1^.prev := temp2;
      p^.next     := temp2;
      make_value  := temp2;
    end;
  end;
  temp1 := nil;
  temp2 := nil;
  p     := nil;
end;    (* MAKE_VALUE *)


(*****************************************************************************)
(* ADD_PARAMETER                                                             *)
(* Takes a parameter list and adds a parameter definition to the end of it   *)
(*                                                                           *)
(* Called from: RULES.PAS : MAKE_BLANKVARS                                   *)
(*              SQLPARSE.YAC : YYLEX                                         *)
(*****************************************************************************)
Function ADD_PARAMETER ( param (*, SQLName*) : string;
                         theList        : plist_ptr ) : plist_ptr;
Var      temp1, temp2 : plist_ptr;
Begin

(* D+
  writeln('In ADD_PARAMETER, appending ',param);
  write('Parameter list is currently: ');
  print_parameters(theList);
  writeln;
   D- *)

  if (theList = nil) then
  begin
    new(theList);
    theList^.next       := nil;
(*    theList^.SQLColumn  := SQLName; *)
    theList^.RuleColumn := param;
    theList^.valueRef   := make_value(param);
  end
  else
  begin
    temp1 := theList;
    while (temp1^.next <> nil) do temp1 := temp1^.next;
    new(temp2);
    temp2^.next       := nil;
(*    temp2^.SQLColumn  := SQLName; *)
    temp2^.RuleColumn := param;
    temp2^.valueRef   := make_value(param);
    temp1^.next       := temp2;
  end;

(* D+
  write('After append, parameter list is now: ');
  print_parameters(theList);
  writeln;
   D- *)

  add_parameter := theList;
  temp1 := nil;
  temp2 := nil;
end;    (* ADD_PARAMETER *)
```

```
(*****************************************************************************)
(* COUNT_ARITY                                                               *)
(* Takes a parameter list and returns the number of parameters in it         *)
(*                                                                           *)
(* Called from: SQLPARSE.YAC : YYLEX                                         *)
(*              RULE.PAS : SAVERULE                                          *)
(*****************************************************************************)
Function COUNT_ARITY ( theList : plist_ptr ) : integer;
Var      num : integer;
Begin
  num := 0;
  while (theList <> nil) do
  begin
    num  := num + 1;
    theList := theList^.next;
  end;
  count_arity := num;
end;    (* COUNT_ARITY *)


END.    (* MODULE LISTOPS *)
```

D.5 RULES.PAS

```
[ INHERIT ('GLOBAL',
           'LISTOPS',
           'DYNAMIC',
           'SYS$LIBRARY:PASCAL$LIB_ROUTINES',
           'SYS$LIBRARY:STARLET'),
  ENVIRONMENT ('RULES') ]


Module RULES (input,output);


(*****************************************************************************)
(*                                                                           *)
(* RULES.PAS - N. Stanger March/April 1989                                   *)
(*                                                                           *)
(* Contains routines that deal with rule-handling                            *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  27-MAR-1989 Major rewrite to incorporate new data structures & algorithms *)
(*  02-FEB-1990 Implemented rule-copying functions, fixed bugs               *)
(*  28-NOV-1990 Implemented PCN to HTR conversion (and vice versa)           *)
(*****************************************************************************)


(*****************************************************************************)
(* COMPILE                                                                   *)
(* Takes a rule and compiles it using the YACC-generated parser in LOGIC.PAS  *)
(*****************************************************************************)
Function COMPILE ( rule  : string;
                   query : boolean ) : boolean;
Begin
  compile  := true;
  scan_pos := 1;
  querying := query;
  rulelength := length(rule);
  if (rulelength > 0) then
  begin
(*    if (yyparse <> 1) then *)
    begin
      if reading_from_file then message(DDB_BADFILERULE)
      else message(DDB_BADINPUTRULE);
      compile := false;
    end;
  end
  else compile := false;
end;    (* COMPILE *)


(*****************************************************************************)
(* GET_RULES                                                                 *)
(* Reads rules from a text file and compiles them                            *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  24-APR-1989  Rewritten so that there is one generic compile routine, not  *)
(*               two separate ones                                           *)
(*****************************************************************************)
Procedure GET_RULES ( filename : string );
Var       line_no : integer;
Begin
  line_no := 1;
  reading_from_file := true;
  open(rulefile,filename,history := old,error := continue);
  reset(rulefile,error := continue);
  if (status(rulefile) <> 0) then scream(DDB_NORULES)
  else
  begin
    while not eof(rulefile) do
    begin
      readln(rulefile,this_rule);
      if (length(this_rule) > 0) then
        if (this_rule[1] <> '!') then  (* rule commented out *)
          if not compile(this_rule, false) then
            writeln('Line ',line_no:1,' of file DDB.RLS');
    end;
    writeln('OK.');
    writeln('Rule list contains:');
    these_are(the_rules);
    reading_from_file := false;
  end;
end; (* GET_RULES *)


(*****************************************************************************)
(* ADD_INSTANT (29-MAY-1989)                                                 *)
(* Creates a variable instantiation, given the values of two parameters, and  *)
(* tacks it on the end of the instantiation list                             *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  07-FEB-1990 Fixed 'john/john' bug                                        *)
(*****************************************************************************)
Function ADD_INSTANT (oldVal, newVal   : string;
                      oldKind, newKind : ptypes;
                      instant          : instant_ptr) : instant_ptr;
Var      temp1, temp2 : instant_ptr;
Begin
  if (instant = nil) then
  begin
    new(instant);
    instant^.next      := nil;
    instant^.old_value := oldVal;
    instant^.old_kind  := oldKind;
    instant^.new_value := newVal;
    instant^.new_kind  := newKind;
  end
```

```
  else
  begin
    temp1 := instant;
    while (temp1^.next <> nil) do temp1 := temp1^.next;
    new(temp2);
    temp2^.old_value := oldVal;
    temp2^.old_kind  := oldKind;
    temp2^.new_value := newVal;
    temp2^.new_kind  := newKind;
    temp2^.next      := nil;
    temp1^.next      := temp2;
    temp1            := nil;
    temp2            := nil;
  end;
  add_instant := instant;
end;    (* ADD_INSTANT *)


(*****************************************************************************)
(* MATCH_PARAMETERS (28-APR-1989)                                            *)
(* Takes two parameter lists and atttempts to match them (see MATCH_RULES    *)
(* below)                                                                    *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  28-MAY-1989 Rewritten in line with new data structures and algorithms    *)
(*              This routine uses the second half of Bratko's matching algo- *)
(*              rithm (predicate parameters)                                 *)
(*****************************************************************************)
Procedure MATCH_PARAMETERS (list1, list2 : plist_ptr;
                            arity        : integer;
                            var matched  : boolean;
                            var instant  : instant_ptr);
Var      i            : integer;
         temp1, temp2 : value_ptr;
Begin
  matched := true;
  i := 1;
  while ((i <= arity) and matched) do
  begin
    temp1 := list1^.valueRef;
    temp2 := list2^.valueRef;
    if (    (    (temp1^.pkind = const_name)
             and (temp2^.pkind = const_name))
        and (temp1^.value <> temp2^.value)) then matched := false
    else if (    (temp1^.pkind = var_name)
             and (temp2^.pkind = const_name)) then
      instant := add_instant(temp1^.value, temp2^.value, temp1^.pkind, temp2^.pkind, instant)
    else if (    (temp1^.pkind = var_name)
             and (temp2^.pkind = var_name)) then
      instant := add_instant(temp2^.value, temp1^.value, temp2^.pkind, temp1^.pkind, instant)
    else if (    (temp1^.pkind = const_name)
             and (temp2^.pkind = var_name)  ) then
      instant := add_instant(temp2^.value, temp1^.value, temp2^.pkind, temp1^.pkind, instant);
    list1 := list1^.next;
    list2 := list2^.next;
    i := i + 1;
  end;
  temp1 := nil;
  temp2 := nil;
end;    (* MATCH_PARAMETERS *)


(*****************************************************************************)
(* MATCH (28-APR-1989)                                                       *)
(* Takes two rules and attempts to match them (a la Prolog pattern-matching) *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  27-MAY-1989 Rewritten to incorporate new data structures and algorithms. *)
(*              Algorithm taken from Bratko, p.37. This routine uses the     *)
(*              first half of the algorithm (structure/predicates)           *)
(*  29-MAY-1989 Added instantiation code                                     *)
(*****************************************************************************)
Procedure MATCH (rule1, rule2 : pred_ptr;
                 var matched  : boolean;
                 var instant  : instant_ptr);
Begin
  matched := false;
  if (rule1 = rule2) then matched := true
  else
  begin
    if ((rule1^.name = rule2^.name) and (rule1^.arity = rule2^.arity)) then
      match_parameters(rule1^.parameters,rule2^.parameters,rule1^.arity,matched,instant);
  end;
end;    (* MATCH *)
```

```
(*****************************************************************************)
(* INSTANTIATE (29-MAY-1989)                                                 *)
(* Substitutes a variable instantiation into a list of goals                 *)
(*****************************************************************************)
Function INSTANTIATE ( instant   : instant_ptr;
                       goal_list : pred_ptr     ) : pred_ptr;
Var      temp, start : plist_ptr;
         goal        : pred_ptr;
         inst        : instant_ptr;
         found       : boolean;
Begin
  if (goal_list = nil) then instantiate := nil
  else if (goal_list^.arity = 0) then  (* THIS SHOULD NOT HAPPEN *)
    scream(DDB_NOARITY)
  else
  begin
    goal := goal_list;
    while (goal <> nil) do
    begin
      start := goal^.parameters;
      inst  := instant;
      while (inst <> nil) do with inst^ do
      begin
        temp := start;
        found := false;
        while ((temp <> nil) and (not found)) do with temp^.valueRef^ do
        begin
          if (value = old_value) then
          begin
            found := true;
            value := new_value;
            pkind := new_kind;
          end
          else temp := temp^.next;
        end;
        inst := inst^.next;
      end;
      goal := goal^.next;
    end;
    instantiate := goal_list;
  end;
end;    (* INSTANTIATE *)


(*****************************************************************************)
(* UNINSTANTIATE (22-FEB-1990)                                               *)
(* Removes a variable instantiation from a list of goals                     *)
(*****************************************************************************)
Function UNINSTANTIATE ( instant   : instant_ptr;
                         goal_list : pred_ptr     ) : pred_ptr;
Var      temp, start : plist_ptr;
         goal        : pred_ptr;
         inst        : instant_ptr;
         found       : boolean;
Begin
  if (goal_list = nil) then uninstantiate := nil
  else if (goal_list^.arity = 0) then  (* THIS SHOULD NOT HAPPEN *)
    scream(DDB_NOARITY)
  else
  begin
    goal := goal_list;
    while (goal <> nil) do
    begin
      start := goal^.parameters;
      inst  := instant;
      while (inst <> nil) do with inst^ do
      begin
        temp := start;
        found := false;
        while ((temp <> nil) and (not found)) do with temp^.valueRef^ do
        begin
          if (value = new_value) then
          begin
            found := true;
            value := old_value;
            pkind := old_kind;
          end
          else temp := temp^.next;
        end;
        inst := inst^.next;
      end;
      goal := goal^.next;
    end;
    uninstantiate := goal_list;
  end;
end; (* UNINSTANTIATE *)


(*****************************************************************************)
(* COPY_VALUES (20-FEB-1990)                                                 *)
(* Makes a copy of the valueRef of a parameter definition. Subroutine of     *)
(* COPY_PARAMETERS, COPY_SIDE AND COPY_RULE.                                 *)
(*****************************************************************************)
Function COPY_VALUES ( thisValue : value_ptr ) : value_ptr;
Var      temp1, temp2, p : value_ptr;
         tmpVal          : string;
         found           : boolean;
Begin
  if (thisValue <> nil) then
  begin
    if (thisValue^.pkind = var_name) then tmpVal := '_' + thisValue^.value
    else tmpVal := '%$^#*&@*!&%';  (* garbage - will definitely <not> be in the list *)
                                   (* always make a copy of constants               *)
    temp1 := theValues;
    found := false;
    p     := nil;
```

```
    while ((temp1 <> nil) and (not found)) do
    begin
      found := (temp1^.value = tmpVal);
      p     := temp1;
      temp1 := temp1^.next;
    end;
    if found then copy_values := p  (* already made a copy of it *)
    else
    begin
      new(temp2);
      with temp2^ do
      begin
        next      := temp1;
        prev      := p;
        pkind     := thisValue^.pkind;
        purgeable := true;
        if (thisValue^.pkind = var_name) then value := '_' + thisValue^.value
        else value := thisValue^.value;
      end;
      if (temp1 <> nil) then temp1^.prev := temp2;
      p^.next     := temp2;
      copy_values := temp2;
    end;
  end
  else copy_values := nil;
  temp1 := nil;
  temp2 := nil;
  p     := nil;
end; (* COPY_VALUES *)


(*****************************************************************************)
(* COPY_PARAMETERS (02-FEB-1990)                                             *)
(* Takes a parameter list and returns a copy of it. Subsidiary routine of    *)
(* COPY_RULE.                                                                *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  22-MAY-1990 Modified to reflect changes in the rule data structures      *)
(*****************************************************************************)
Function COPY_PARAMETERS ( theList : plist_ptr ) : plist_ptr;
Var      temp : plist_ptr;
Begin
  if (theList <> nil) then
  begin
    new(temp);
    with temp^ do
    begin
      next       := copy_parameters(theList^.next);
      valueRef   := copy_values(theList^.valueRef);
      RuleColumn := theList^.RuleColumn;
      SQLColumn  := theList^.SQLColumn;
    end;
    copy_parameters := temp;
  end
  else copy_parameters := nil;
  temp := nil;
end; (* COPY_PARAMETERS *)


(*****************************************************************************)
(* COPY_PRED (08-MAY-1990)                                                   *)
(* Makes a copy of a predicate and returns it. Similar to COPY_SIDE except   *)
(* that it only handles one predicate (instead of a list of them). Any NEXT  *)
(* elements for the predicates are removed.                                  *)
(*****************************************************************************)
Function COPY_PRED ( srcPred : pred_ptr ) : pred_ptr;
Var      temp : pred_ptr;
Begin
  with srcPred^ do
  begin
    new(temp);
    temp^.next       := nil;  (* truncate any lists *)
    temp^.name       := name;
    temp^.arity      := arity;
    temp^.negated    := negated;
    temp^.parameters := copy_parameters(parameters);
  end;
  copy_pred := temp;
  temp      := nil;
end; (* COPY_PRED *)


(*****************************************************************************)
(* COPY_SIDE (02-FEB-1990)                                                   *)
(* Takes a LHS or RHS of a rule and returns a copy of it. Subsidiary routine *)
(* of COPY_RULE.                                                             *)
(*****************************************************************************)
Function COPY_SIDE (theSide : pred_ptr ) : pred_ptr;
Var      temp : pred_ptr;
Begin
  if (theSide <> nil) then
  begin
    new(temp);
    with temp^ do
    begin
      next       := copy_side(theSide^.next);
      name       := theSide^.name;
      arity      := theSide^.arity;
      negated    := theSide^.negated;
      parameters := copy_parameters(theSide^.parameters);
    end;
    copy_side := temp;
  end
  else copy_side := nil;
  temp := nil;
end; (* COPY_SIDE *)
```

```
(*************************************************************************)
(* COPY_RULE (02-FEB-1990)                                               *)
(* This routine takes a rule and returns a copy of it. This is to prevent the *)
(* original rule base being modified during the deduction process - only the  *)
(* copy is changed.                                                      *)
(*                                                                       *)
(* MODIFIED:                                                             *)
(*  22-MAY-1990 Modified to reflect changes in the rule data structures.      *)
(*************************************************************************)
Function COPY_RULE (theRule : clause_ptr) : clause_ptr;
Var      temp : clause_ptr;
Begin
  if (theRule <> nil) then
  begin
    new(temp);
    with temp^ do
    begin
      next       := nil;
      left       := copy_side(theRule^.left);
      right      := copy_side(theRule^.right);
      tableName  := theRule^.tableName;
(*      columnList := CopyColumns(theRule^.columnList);  *)
    end;
    copy_rule := temp;
  end
  else copy_rule := nil;
  temp := nil;
end; (* COPY_RULE *)


(*************************************************************************)
(* ADD_GOALS (27-MAY-1989)                                               *)
(* Takes the list of current goals and adds any new goals to it          *)
(*************************************************************************)
Function ADD_GOALS (new_goals, old_goals : pred_ptr ) : pred_ptr;
Var      temp : pred_ptr;
Begin
  if (new_goals <> nil) then
  begin
    if (old_goals = nil) then add_goals := new_goals
    else
    begin
      temp := old_goals;
      while (temp^.next <> nil) do temp := temp^.next;
      temp^.next := new_goals;
      add_goals := old_goals;
      temp := nil;
    end;
  end
  else add_goals := old_goals;
end;    (* ADD_GOALS *)


(*************************************************************************)
(* SEARCH_INSTANT (05-FEB-1990)                                          *)
(* Takes an instantiation list and a particular variable instantiation and    *)
(* searches the list for the instantiation. Returns a pointer to the item if  *)
(* found, nil, otherwise.                                                *)
(*************************************************************************)
Function SEARCH_INSTANT (item, inst : instant_ptr) : instant_ptr;
Var      temp  : instant_ptr;
         found : boolean;
Begin
  temp := inst;
  found := false;
  while ((temp <> nil) and (not found)) do
  begin
    found := (    (temp^.old_value = item^.old_value)
              and (temp^.new_value = item^.new_value) );
    if (not found) then temp := temp^.next;
  end;
  search_instant := temp;
end; (* SEARCH_INSTANT *)


(*************************************************************************)
(* COMBINE_INSTANTS (05-FEB-1990)                                        *)
(* Take two instantiation lists and combine them according to the following   *)
(* rules:                                                                *)
(*                                                                       *)
(*      o If one instantiation is empty, return the other (OK if both are     *)
(*        empty).                                                        *)
(*      o If an item in one list refers to the same object as one in the      *)
(*        other list, return constants in preference to variables (if both    *)
(*        are variables, return the `new' one. Example:                  *)
(*                                                                       *)
(*              old = [_x/_x, _y/_y, _z/bill]                            *)
(*              new = [_x/john, _y/bert, _z/_z]                          *)
(*                                                                       *)
(*        The returned list should be: [_x/john, _y/bert, _z/bill]           *)
(*************************************************************************)
Function COMBINE_INSTANTS (oldList, newList : instant_ptr) : instant_ptr;
Var      temp, found_item : instant_ptr;
Begin
  if (oldList = nil) then combine_instants := newList
  else if (newList = nil) then combine_instants := oldList
  else  (* neither is nil *)
  begin
    temp := newList;
    while (temp <> nil) do
    begin
      found_item := search_instant(temp, oldList);
      if (found_item = nil) then  (* not in old list *)
        oldList := add_instant(temp^.old_value, temp^.new_value, temp^.old_kind, temp^.new_kind, oldList)
      else if (   (    (found_item^.new_kind = var_name)
                   and (temp^.new_kind = const_name)    )  (* new one is a const, old one isn't *)
               or (    (found_item^.new_kind = var_name)
                   and (temp^.new_kind = var_name)      )  ) then  (* both variables *)
```

```
      begin
        found_item^.old_value := temp^.old_value;    (* replace value in oldList with value from newList *)
        found_item^.new_value := temp^.new_value;
      end
      else if (    (found_item^.new_kind = const_name)
               and (temp^.new_kind = var_name)       ) then  (* old one is a const, new one isn't - do nothing *)
      else  (* both constants - THIS SHOULD NOT HAPPEN *)
        scream(DDB_NOCOMBINE);
      temp := temp^.next;
    end;
    combine_instants := oldList;
  end;
end; (* COMBINE_INSTANT *)


(*************************************************************************)
(* DEDUCE (28-APR-1989)                                                  *)
(* The 'main deduction routine'. Finds rules and executes their LHS's (rather *)
(* like Prolog).                                                         *)
(*                                                                       *)
(* MODIFIED:                                                             *)
(*  27-MAY-1989 Rewritten to use better algorithm (Bratko, pp. 46-48)    *)
(*  02-FEB-1990 Fixed instantiation bug                                  *)
(*  17-APR-1990 Added code to return expanded rules and interface with the    *)
(*              rest of SQUALID                                          *)
(*  06-DEC-1990 Modified slightly to work with new relational rule storage    *)
(*                                                                       *)
(* Called from: DEDUCE                                                   *)
(*              SQL_PROCESS.PAS : EXPANDRULE                             *)
(*************************************************************************)
Function DEDUCE ( rule_list       : clause_ptr;
                  var goal_list   : pred_ptr;
                  var ret_instant : instant_ptr;
                  var expand_rule : pred_ptr     ) : boolean;
Var      temp, copy                      : clause_ptr;
         this_goal, other_goals, new_goals : pred_ptr;
         satisfied, match_ok             : boolean;
         instant, new_instant            : instant_ptr;
Begin
  match_ok    := false;
  instant     := nil;
  new_instant := nil;
  new_goals   := nil;
  if (goal_list = nil) then deduce := true  (* nothing left - succeed *)
  else  (* keep expanding *)
  begin
    temp        := rule_list;
    this_goal   := goal_list;
    other_goals := goal_list^.next;
    satisfied   := false;
    while ((not satisfied) and (temp <> nil)) do
    begin
      instant := nil;
      new_goals := nil;           (* messy, but it works *)
      copy := copy_rule(temp);

(* D+
      writeln; writeln('*****');
      write('Current goal is ');
      dump_side(this_goal, true);
      writeln; writeln;
      writeln('Remaining goals are:');
      dump_side(other_goals, true);
      writeln; writeln('-----');
      write('Testing rule ');
      these_are(copy);
   D- *)

      if (not RuleReference(this_goal^.name)) then  (* it's a base table *)
      begin
        new_goals   := other_goals;
        satisfied   := deduce(rule_list, new_goals, new_instant, expand_rule);
        expand_rule := add_pred(expand_rule, copy_pred(this_goal));
        if satisfied then
          ret_instant := combine_instants(instant, new_instant);
      end
      else  (* it's a virtual table - attempt to match *)
      begin
        match(this_goal, copy^.right, match_ok, instant);
        if match_ok then
        begin

(* D+
          writeln('Rules matched OK.');
          write('Instant = '); dump_instant(instant);
   D- *)

          new_goals := add_goals(copy^.left, other_goals);
          new_goals := instantiate(instant, new_goals);

(* D+
          writeln('New goals are:');
          dump_side(new_goals, true);
          writeln; writeln('-----');
          writeln('>>> Recursing...');
   D- *)

          satisfied := deduce(rule_list, new_goals, new_instant, expand_rule);

(* D+
          write('New instant = '); dump_instant(new_instant);
   D- *)
```

```
          if satisfied then
            ret_instant := combine_instants(instant, new_instant)
          else
          begin
            ret_instant := nil;
            goal_list   := uninstantiate(instant, goal_list);
          end;

(* D+
          write('Return instantiation  = '); dump_instant(ret_instant);
          writeln('<<< Returned from recursion...');
   D- *)

        end;
      end;
      temp := temp^.next;
    end;
    deduce := satisfied;
  end;
end;    (* DEDUCE *)


(*****************************************************************************)
(* SPLIT (04-DEC-1990)                                                       *)
(* Given a string of the form 'nnn.nnn', extracts the two numbers and returns *)
(* them separately.                                                          *)
(*                                                                           *)
(* Called from: LOADRULE                                                     *)
(*****************************************************************************)
Function SPLIT ( inStr : String ) : StringPair_ptr;
Var       dot, equals : integer; (* positions of '.' and '=' *)
          left, right : String value '';
          tempS       : StringPair_ptr;
Begin
  inStr := inStr + '=';
  while (inStr <> '') do
  begin
    dot    := index(inStr, '.');
    equals := index(inStr, '=');
    readv(substr(inStr, 1, dot - 1), left);
    readv(substr(inStr, dot + 1, equals - dot - 1), right);
    tempS := Append_StringPair(left, right, tempS);
    if (equals = length(inStr)) then inStr := ''
    else inStr := substr(inStr, equals + 1, length(inStr) - equals);
  end;
  Split := tempS;
  tempS := nil;
end; (* SPLIT *)


(*****************************************************************************)
(* MAKE_BLANKVARS (04-DEC-1990)                                              *)
(* Creates n blank parameter holders for a predicate. These will be filled in *)
(* with variables at a later date.                                           *)
(*                                                                           *)
(* Called from: LOADRULE                                                     *)
(*****************************************************************************)
Function MAKE_BLANKVARS ( numVars : integer ) : plist_ptr;
Var      i    : integer value 0;
         temp : plist_ptr value nil;
Begin
  for i := 1 to numVars do
    temp := Add_Parameter(smiley, temp);  (* insert marker  for later identification *)
  Make_BlankVars := temp;
  temp           := nil;
end; (* MAKE_BLANKVARS *)


(*****************************************************************************)
(* NEWVAR (04-DEC-1990)                                                      *)
(* Creates a new, unique variable id.                                        *)
(*                                                                           *)
(* Called from: INSERT_COMMONVARS                                            *)
(*              INSERT_VARS                                                  *)
(*****************************************************************************)
Function NEWVAR : string;
Begin
  NewVar  := '';
  numVars := numVars + 1;
  if (numVars > maxVars) then scream(DDB_TOOMANYVARS)  (* extremely unlikely *)
  else NewVar := '_' + NumToString(numVars);
end; (* NEWVAR *)


(*****************************************************************************)
(* INSERT_COMMONVARS (04-DEC-1990)                                           *)
(* Inserts linked parameter values into the rule for identical parameters    *)
(*                                                                           *)
(* Called from: LOADRULE                                                     *)
(*****************************************************************************)
Procedure INSERT_COMMONVARS ( theList      : StringPair_ptr;
                              lList, rList : pred_ptr        );
Var       tempT       : Pred_ptr value nil;
          tempC       : Plist_ptr value nil;
          i, tab, col : Integer value 0;
          theVar      : Value_ptr value nil;
Begin
  theVar := Make_Value(NewVar);
  while (theList <> nil) do with theList^ do
  begin
    readv(str1, tab);
    readv(str2, col);
    if (tab < 0) then
    begin
      tempT := rList;
      tab   := -tab;
    end
    else tempT := lList;
```

```
    for i := 1 to (tab - 1) do tempT := tempT^.next; (* if = 1 then don't move *)
    tempC := tempT^.parameters;
    for i := 1 to (col - 1) do tempC := tempC^.next; (* ditto for this *)
    with tempC^ do
    begin
      valueRef   := theVar;
      ruleColumn := valueRef^.value;
    end;
    theList := theList^.next;
  end;
  tempT := nil;
  tempC := nil;
end; (* INSERT_COMMONVARS *)


(*****************************************************************************)
(* INSERT_VARS (04-DEC-1990)                                                 *)
(* Creates parameter definitions for the remaining unlinked variables.       *)
(*                                                                           *)
(* Called from: LOADRULE                                                     *)
(*****************************************************************************)
Procedure INSERT_VARS ( theList : pred_ptr );
Var       tempC : plist_ptr;
          tempT : pred_ptr;
Begin
  tempT := theList;
  while (tempT <> nil) do with tempT^ do
  begin
    tempC := parameters;
    while (tempC <> nil) do with tempC^ do
    begin
      if (valueRef^.value = smiley) then valueRef := Make_Value(NewVar);  (* if no value, then create one *)
      ruleColumn := valueRef^.value;
      tempC := tempC^.next;
    end;
    tempT := tempT^.next;
  end;
  tempT := nil;
  tempC := nil;
end; (* INSERT_VARS *)


(*****************************************************************************)
(* LOADRULE (28-NOV-1990)                                                    *)
(* Loads a rule from the database into main memory for deduction purposes.   *)
(* Converts from PCN format to HTR format.                                   *)
(*                                                                           *)
(* Called from: DEDUCE                                                       *)
(*              SQL_PROCESS.PAS : EXPANDRULE                                 *)
(*****************************************************************************)
Procedure LOADRULE ( ruleName : String );
Var      outList   : clause_ptr value nil;
         inList,
         tempPCN   : PCN_ptr value nil;
         script    : String;   (* transition inscription *)
         postCols,
         comma     : Integer;  (* position of first ',' in script *)
         tempP     : pred_ptr;
         identList : StringPair_ptr value nil;
Begin
  inList := SearchPCN(ruleName);
  if (inList <> nil) then  (* found it *)
  begin
    new(outList);
    with outList^ do
    begin
      tableName := ruleName;
      next      := nil;
      left      := nil;
      postCols  := FindArity(ruleName);
      right     := Make_Pred(ruleName, Make_BlankVars(postCols), postCols);  (* parameters will be fixed up shortly ... *)
      tempPCN   := inList;
      while (tempPCN <> nil) do with tempPCN^ do
      begin
        LoadRule(prePlace);  (* load this as well, if it is a rule *)
        left    := Add_Pred(left, Make_Pred(prePlace, Make_BlankVars(preCols), FindArity(prePlace)));  (* ... ditto ... *)
        tempPCN := tempPCN^.next;
      end;
      script := GetInscription(inList^.id); (* note assumption that virtual tables are defined by one rule only *)
      while (script <> '') do  (* scan until no more conditions *)
      begin
        comma     := index(script, ',');
        identList := Split(substr(script, 1, comma - 1));
        Insert_CommonVars(identList, left, right);       (* ... e voila! The parameters are inserted here *)
        if (comma = length(script)) then script := ''     (* (well, some of them anyway ...)               *)
        else script := substr(script, comma + 1, length(script) - comma);
      end;
      Insert_Vars(left);  (* ... create parameters for any that are left *)
      Insert_Vars(right);
    end;  (* with outList^ do ... *)
  end;  (* if inList <> nil ... *)
  Add_Rule(outList);  (* add rule to global rule list *)
  outList := nil;
  inList  := nil;
  tempPCN := nil;
  tempP   := nil;
end;  (* LOADRULE *)
```

```
(*************************************************************************)
(* FINDIDENTICALVARS (11-DEC-1990)                                       *)
(* Given a parameter and a rule, searches the rule for parameters that are    *)
(* identical to the originl parameter. Returns a list of them in transition   *)
(* inscription notation. WARNING: attempting to understand this routine may   *)
(* be hazardous to your mental health!                                        *)
(*                                                                       *)
(* Called from: SAVERULE                                                 *)
(*************************************************************************)
Function FINDIDENTICALVARS ( findParm            : Plist_ptr;  (* parameter to match with *)
                             startPred           : Pred_ptr;   (* predicate to start searching from *)
                             firstPred, firstParm : integer;    (* positions of findParm and it's predicate *)
                             RHS                 : boolean;    (* is this the RHS of the rule? *)
                             script              : String;
                             var found           : boolean    ) : String;
Var      thisPred           : Pred_ptr;            (* current predicate *)
         thisParm           : Plist_ptr;           (* current parameter *)
         firstOne           : boolean value true;
         currPred, currParm : integer value 0;     (* current positions *)
Begin
  thisPred := startPred;
  currPred := firstPred;  (* start at the correct predicate *)
  while (thisPred <> nil) do with thisPred^ do
  begin
    if (firstOne and not RHS) then  (* first time through the loop? *)
    begin
      thisParm := findParm;  (* start at the correct parameter *)
      currParm := firstParm; (* need all this because we might start part-way down the parameter list *)
      firstOne := false;     (* only do it if we're scanning the LHS though ... *)
    end
    else
    begin
      thisParm := parameters;
      currParm := 1;
    end;
    while (thisParm <> nil) do with thisParm^ do
    begin
      if ((thisParm^.ruleColumn = findParm^.ruleColumn) and (thisParm <> findParm)) then (* check it isn't the same one *)
      begin
        if (not found) then  (* found a match - we can now insert an entry into the inscription for findParm *)
        begin
          (* it is reasonably unlikely that there will be identical variables solely on the RHS *)
          (* therefore, don't worry about it as yet ... (TO BE CHANGED) *)
          script := script + NumToString(firstPred) + '.' + NumToString(firstParm) + '=';
          found  := true;
        end;
        if RHS then script := script + NumToString(-currPred) + '.' + NumToString(currParm) + '='
        else script := script + NumToString(currPred) + '.' + NumToString(currParm) + '=';
        valueRef^.value := smiley;  (* eliminate this variable from further consideration *)
      end;
      currParm := currParm + 1;
      thisParm := thisParm^.next;
    end;
    currPred := currPred + 1;
    thisPred := thisPred^.next;
  end;
  FindIdenticalVars      := script;
  thisPred               := nil;
  thisParm               := nil;
end; (* FINDIDENTICALVARS *)


(*************************************************************************)
(* GENTRANSID (12-DEC-1990)                                              *)
(* Generates a unique (hopefully!) transition ID using the system date and    *)
(* time. The ID is formed as follows:                                         *)
(*                                                                       *)
(*      1. String together the first letters of all predicates in the rule    *)
(*      2. Add the current date/time (down to nearest '00th of a second)      *)
(*                                                                       *)
(* Called from: SAVERULE                                                 *)
(*************************************************************************)
Function GENTRANSID ( theRule : Clause_ptr ) : String;
Var      theTime : $uquad;
         theID   : String value '';
         tempP   : Pred_ptr;
         date    : String;
Begin
  (* 1. get first letters of all predicate names *)
  tempP := theRule^.left;
  while (tempP <> nil) do with tempP^ do  (* LHS *)
  begin
    theID := theID + substr(name, 1, 1);
    tempP := tempP^.next;
  end;
  tempP := theRule^.right;
  while (tempP <> nil) do with tempP^ do  (* RHS *)
  begin
    theID := theID + substr(name, 1, 1);
    tempP := tempP^.next;
  end;

  (* 2. get the system date/time and add it on the end *)
  $gettim(theTime);
  $asctim(date.length, date.body, theTime, 0);
  theID := theID + date;

  GenTransID := theID;
  tempP      := nil;
end; (* GENTRANSID *)
```

```
(*****************************************************************************)
(* SAVERULE (11-DEC-1990)                                                    *)
(* Saves a rule from main memory into the database. Converts from HTR format  *)
(* to modified PCN format.                                                   *)
(*                                                                           *)
(* Called from: SQL_PROCESS : EXPAND_CT_STATEMENT                            *)
(*****************************************************************************)
Procedure SAVERULE ( theRule : clause_ptr );  (* NOTE: this can never be nil *)
Var       script    : String value '';  (* transition inscription *)
          transID   : String value '';  (* transition ID *)
          thisPred  : Pred_ptr;
          thisParm  : Plist_ptr;
          predPos,
          parmPos   : integer value 0;
          found     : boolean value false;  (* found a match yet? *)
Begin
  thisPred := theRule^.left;
  while (thisPred <> nil) do with thisPred^ do  (* scan LHS of rule *)
  begin
    predPos := predPos + 1;
    parmPos := 0;
    thisParm := parameters;
    while (thisParm <> nil) do with thisParm^ do
    begin
      parmPos  := parmPos + 1;
      if (valueRef^.value <> smiley) then  (* haven't eliminated this one yet *)
      begin
        (* search for identical variables on LHS and RHS *)
        script   := FindIdenticalVars(thisParm, thisPred, predPos, parmPos, false, script, found);
        script   := FindIdenticalVars(thisParm, theRule^.right, predPos, parmPos, true, script, found);
        script[length(script)] := ',';  (* change last '=' to a comma *)
      end;
      thisParm := thisParm^.next;
      found    := false;
    end;
    thisPred := thisPred^.next;
  end;
  thisPred := theRule^.right;
  predPos  := 0;
  while (thisPred <> nil) do with thisPred^ do  (* scan RHS of rule *)
  begin
    predPos  := predPos + 1;
    parmPos  := 0;
    thisParm := parameters;
    while (thisParm <> nil) do with thisParm^ do
    begin
      parmPos  := parmPos + 1;
      if (valueRef^.value <> smiley) then
      begin
        script   := FindIdenticalVars(thisParm, thisPred, predPos, parmPos, true, script, found);
        script[length(script)] := ',';
      end;
      thisParm := thisParm^.next;
    end;
    thisPred := thisPred^.next;
  end;
  transID := GenTransID(theRule);
  StoreTransition(transID, script);
  thisPred := theRule^.left;
  predPos  := 0;
  while (thisPred <> nil) do with thisPred^ do
  begin
    predPos := predPos + 1;
    StorePTP(transID, name, predPos, Count_Arity(parameters), theRule^.right^.name) ;
    (* NOTE: assume only one predicate on RHS at the moment - makes life much simpler! *)
    thisPred := thisPred^.next;
  end;
  thisPred    := nil;
  thisParm    := nil;
end; (* SAVERULE *)


END. (* MODULE RULES *)
```

D.6 SQLPARSE.YAC

```
%{
[ INHERIT ('GLOBAL',
           'DYNAMIC',
           'LISTOPS',
           'SQL_PROCESS',
           'SYS$LIBRARY:PASCAL$STR_ROUTINES'),
 ENVIRONMENT ('SQLPARSE') ]


MODULE SQL_PARSER (input,output);


(*****************************************************************************)
(* SQL_PARSER.PAS - N. Stanger, August 1989                                  *)
(*                                                                           *)
(* Contains routines for parsing and pre-processing SQLD statements. Any ref- *)
(* erences to rules are expanded (see EXPAND_STATEMENT et al).               *)
(*                                                                           *)
(* The parser is based partly on a BNF grammar for ANSI SQL taken from Appen- *)
(* dix B of "A Guide to the SQL Standard" by C.J. Date, but mainly on the    *)
(* syntax diagrams found in the VAX SQL Reference Manual.                    *)
(*****************************************************************************)


VAR    theStatement  : st;
       stmt_len, len : $uword;
       scan          : integer;
       temp255       : Str255;
       sql_send      : boolean := true;  (* TRUE if this statement is to be sent   *)
                                         (* to SQL for processing, otherwise FALSE *)
       expandIt      : boolean := false;  (* TRUE if the statement might need pro- *)
                                          (* cessing, otherwise FALSE              *)
       currentStmt   : SQLStatement;  (* temp storage for stmt we are *)
                                      (* currently processing         *)


(* Put this in in utter desperation due to inexplicable failure when using *)
(* STR$TRIM with VARYING strings (BODY + LENGTH method).                   *)
[ASYNCHRONOUS, EXTERNAL(str$trim)] FUNCTION str$trim_v (
        VAR destination_string : [VOLATILE] VARYING [max1] OF CHAR;
        source_string : VARYING [max2] OF CHAR;
        VAR resultant_length : [VOLATILE] $UWORD := %IMMED 0) : UNSIGNED; EXTERNAL;


(*****************************************************************************)
(* YYWRITE and YYERROR are required by Yacc                                  *)
(*****************************************************************************)
Procedure YYWRITE (msg : string);  (* needed by YACC *)
Begin
  writeln(msg);
end;    (* YYWRITE *)


Procedure YYERROR (msg : string);  (* needed by YACC *)
Begin
  if (msg <> 'syntax error') then writeln(msg);
end;    (* YYERROR *)


Function YYLEX : integer; forward;


(*****************************************************************************)
(* ADD_SELECT_ITEM (29-MAR-1990)                                             *)
(* Adds an item to a SELECT list.                                            *)
(*****************************************************************************)
Function ADD_SELECT_ITEM ( thePtr, theList : Select_ptr ) : Select_ptr;
Var      temp : Select_ptr;
Begin
  if (theList = nil) then theList := thePtr
  else
  begin
    temp := theList;
    while (temp^.next <> nil) do temp := temp^.next;
    temp^.next := thePtr;
  end;
  add_select_item := theList;
  temp := nil;
end; (* ADD_SELECT_ITEM *)


(*****************************************************************************)
(* ADD_FROM_TABLE (06-APR-1990)                                              *)
(* Adds a table to a FROM list.                                              *)
(*****************************************************************************)
Function ADD_FROM_TABLE ( thePtr, theList : From_ptr ) : From_ptr;
Var      temp : From_ptr;
Begin
  if (theList = nil) then theList := thePtr
  else
  begin
    temp := theList;
    while (temp^.next <> nil) do temp := temp^.next;
    temp^.next := thePtr;
  end;
  add_from_table := theList;
  temp := nil;
end; (* ADD_FROM_TABLE *)
```

```
(*************************************************************************)
(* BUILDSELITEM (07-JUN-1990)                                            *)
(* Builds a SELECT item and returns it.                                  *)
(*************************************************************************)
Function BUILDSELITEM ( tableName, columnName : string ) : Select_ptr;
Var      tempSel : Select_ptr;
Begin
  new(tempSel);
  with tempSel^ do
  begin
    next := nil;
    new(theItem);
    theItem^.nodeType  := cName;
    theItem^.theTable  := tableName;
    theItem^.theColumn := columnName;
  end;
  BuildSelItem := tempSel;
  tempSel      := nil;
end; (* BUILDSELITEM *)


(*************************************************************************)
(* BUILDFROMITEM (07-JUN-1990)                                           *)
(* Builds a FROM item and returns it.                                    *)
(*************************************************************************)
Function BUILDFROMITEM ( tableName, aliasName : string ) : From_ptr;
Var      tempFrom : From_ptr;
Begin
  new(tempFrom);
  tempFrom^.theTable    := tableName;
  tempFrom^.theAlias    := aliasName;
  tempFrom^.ruleColumns := GetTableColumns(tableName);
  tempFrom^.next        := nil;
  BuildFromItem         := tempFrom;
  tempFrom              := nil;
end; (* BUILDFROMITEM *)


(*************************************************************************)
(* CREATESELECTEXPR (15-SEP-1990)                                        *)
(* Creates a SelectStmt_ptr and returns it.                              *)
(*************************************************************************)
Function CREATESELECTEXPR ( dupl  : String;
                            sList : Select_ptr;
                            fList : From_ptr;
                            wList : Group_ptr;
                            gList : Where_ptr;
                            hList : Where_ptr   ) : SelectStmt_ptr;
Var      tempSel : SelectStmt_ptr;
Begin
  new(tempSel);
  with tempSel^ do
  begin
    columnList := nil;
    duplicates := dupl;
    selectList := sList;
    fromList   := fList;
    groupList  := gList;
    havingList := hList;
    whereList  := wList;
    orderList  := nil;
  end;
  CreateSelectExpr := tempSel;
end; (* CREATESELECTEXPR *)


(*************************************************************************)
(* ADDORDERCLAUSE (18-SEP-1990)                                          *)
(* Adds an ORDER BY clause to a select_expr.                             *)
(*************************************************************************)
Function ADDORDERCLAUSE ( theOrder  : Order_ptr;
                          theSelect : SelectStmt_ptr ) : SelectStmt_ptr;
Begin
  theSelect^.orderList := theOrder;
  AddOrderClause       := theSelect;
end; (* ADDORDERCLAUSE *)


(*************************************************************************)
(* CREATESELECTSTATEMENT (18-SEP-1990)                                   *)
(* Creates a SELECT SQLStatement and returns it.                         *)
(*************************************************************************)
Function CREATESELECTSTATEMENT ( theSelect : SelectStmt_ptr ) : SQLStatement;
Var      temp : SQLStatement;
Begin
  with temp do
  begin
    stmtKind := SelectStmt;
    selBody  := theSelect;
  end;
  CreateSelectStatement := temp;
end; (* CREATESELECTSTATEMENT *)


(*************************************************************************)
(* EXPRTOSELECT (17-SEP-1990)                                            *)
(* Takes an Expr_ptr and returns a Select_ptr to it.                     *)
(*************************************************************************)
Function EXPRTOSELECT ( theExpr : Expr_ptr ) : Select_ptr;
Var      temp : Select_ptr;
Begin
  new(temp);
  temp^.next    := nil;
  temp^.theItem := theExpr;
  ExprToSelect  := temp;
  temp          := nil;
end; (* EXPRTOSELECT *)
```

```
(*****************************************************************************)
(* EXPRTOWHERE (31-OCT-1990)                                                 *)
(* Takes an Expr_ptr and returns a Where_ptr to it.                          *)
(*****************************************************************************)
Function EXPRTOWHERE ( theExpr : Expr_ptr ) : Where_ptr;
Var      temp : Where_ptr;
Begin
  new(temp);
  temp^.next    := nil;
  temp^.theItem := theExpr;
  ExprToWhere   := temp;
  temp          := nil;
end; (* EXPRTOWHERE *)


(*****************************************************************************)
(* MAKEOPRNODE (17-SEP-1990)                                                 *)
(* Creates an operator node in an expression tree.                           *)
(*****************************************************************************)
Function MAKEOPRNODE ( theOpr       : String;
                       lExpr, rExpr : Expr_ptr ) : Expr_ptr;
Var      temp : Expr_ptr;
Begin
  new(temp);
  temp^.nodeType := stdOpr;
  temp^.theOp    := theOpr;
  temp^.left     := lExpr;
  temp^.right    := rExpr;
  MakeOprNode    := temp;
  temp           := nil;
end; (* MAKEOPRNODE *)


(*****************************************************************************)
(* MAKESUBSELNODE (17-SEP-1990)                                              *)
(* Creates a subselect node in an expression tree.                           *)
(*****************************************************************************)
Function MAKESUBSELNODE ( theSelect : SelectStmt_ptr ) : Expr_ptr;
Var      temp : Expr_ptr;
Begin
  new(temp);
  temp^.nodeType     := subSel;
  temp^.theSubSelect := theSelect;
  MakeSubselNode     := temp;
  temp               := nil;
end; (* MAKESUBSELNODE *)


(*****************************************************************************)
(* MAKEFNNODE (17-SEP-1990)                                                  *)
(* Creates a function node in an expression tree.                            *)
(*****************************************************************************)
Function MAKEFNNODE ( fnName : String;
                      fnExpr : Expr_ptr ) : Expr_ptr;
Var      temp : Expr_ptr;
Begin
  new(temp);
  temp^.nodeType := dbFn;
  temp^.theFn    := fnName;
  temp^.theArg   := fnExpr;
  MakeFnNode     := temp;
  temp           := nil;
end; (* MAKEFNNODE *)


(*****************************************************************************)
(* MAKECNAMENODE (17-SEP-1990)                                               *)
(* Creates a column name node in an expression tree.                         *)
(*****************************************************************************)
Function MAKECNAMENODE ( tableName, columnName : String ) : Expr_ptr;
Var      temp : Expr_ptr;
Begin
  new(temp);
  temp^.nodeType  := cName;
  temp^.theTable  := tableName;
  temp^.theColumn := columnName;
  MakeCnameNode   := temp;
  temp            := nil;
end; (* MAKECNAMENODE *)


(*****************************************************************************)
(* MAKELITNODE (17-SEP-1990)                                                 *)
(* Creates a literal node in an expression tree.                             *)
(*****************************************************************************)
Function MAKELITNODE ( theStr : String ) : Expr_ptr;
Var      temp : Expr_ptr;
Begin
  new(temp);
  temp^.nodeType := lit;
  temp^.theLit   := theStr;
  MakeLitNode    := temp;
  temp           := nil;
end; (* MAKELITNODE *)
```

```
(*************************************************************************)
(* ADDEXPRITEM (06-NOV-1990)                                             *)
(* Adds an expression to an expression list.                             *)
(*************************************************************************)
Function ADDEXPRITEM ( theItem, theList : Expr_ptr ) : Expr_ptr;
Var      temp1, temp2 : Expr_ptr;
Begin
  if (theList = nil) then
  begin
    new(theList);
    theList^.nodeType := exprList;
    theList^.theExpr  := theItem;
    theList^.next     := nil;
  end
  else
  begin
    temp1 := theList;
    while (temp1^.next <> nil) do temp1 := temp1^.next;
    new(temp2);
    temp2^.nodeType := exprList;
    temp2^.theExpr  := theItem;
    temp2^.next     := nil;
    temp1^.next     := temp2;
  end;
  AddExprItem := theList;
  temp1       := nil;
  temp2       := nil;
end; (* ADDEXPRITEM *)


(*************************************************************************)
(* ADDORDERITEM (19-NOV-1990)                                            *)
(* Appends an item to an ORDER clause.                                   *)
(*************************************************************************)
Function ADDORDERITEM ( theItem, theList : Order_ptr ) : Order_ptr;
Var      temp : Order_ptr;
Begin
  if (theList = nil) then theList := theItem
  else
  begin
    temp := theList;
    while (temp^.next <> nil) do temp := temp^.next;
    temp^.next := theItem;
  end;
  AddOrderItem := theList;
  temp         := nil;
end; (* ADDORDERITEM *)


(*************************************************************************)
(* MAKEORDERITEM (19-NOV-1990)                                           *)
(* Creates an item for an ORDER clause list.                             *)
(*************************************************************************)
Function MAKEORDERITEM ( column   : Expr_ptr;
                         ordering : String   ) : Order_ptr;
Var      temp : Order_ptr;
Begin
  new(temp);
  temp^.theColumn := column;
  if (ordering = '') then temp^.theOrder := 'ASC'
  else temp^.theOrder := ordering;
  temp^.next    := nil;
  MakeOrderItem := temp;
  temp          := nil;
end; (* MAKEORDERITEM *)


(*************************************************************************)
(* ADDGROUPCOLUMN (19-NOV-1990)                                          *)
(* Adds a column name to a list of columns for a GROUP BY clause.        *)
(*************************************************************************)
Function ADDGROUPCOLUMN ( groupColumn : Expr_ptr;
                          theList     : Group_ptr ) : Group_ptr;
Var      temp1, temp2 : Group_ptr;
Begin
  if (theList = nil) then
  begin
    new(theList);
    theList^.theColumn := groupColumn;
    theList^.next      := nil;
  end
  else
  begin
    temp1 := theList;
    while (temp1^.next <> nil) do temp1 := temp1^.next;
    new(temp2);
    temp2^.theColumn := groupColumn;
    temp2^.next      := nil;
    temp1^.next      := temp2;
  end;
  AddGroupColumn := theList;
  temp1          := nil;
  temp2          := nil;
end; (* ADDGROUPCOLUMN *)
```

```
(*****************************************************************************)
(* CREATECTPTR (10-DEC-1990)                                                 *)
(* Builds a structure for storing a CREATE TABLE statement and returns a     *)
(* pointer to it.                                                            *)
(*                                                                           *)
(* Called from: YYLEX                                                        *)
(*****************************************************************************)
Function CREATECTSTATEMENT ( theTable   : String;
                             columnList : CTColumn_ptr;
                             theRule    : Clause_ptr    ) : SQLStatement;
Var      temp : SQLstatement;
Begin
  with temp do
  begin
    stmtKind := CreateTabl;
    new(creBody);
    with creBody^ do
    begin
      tableName := theTable;
      colDefs   := columnList;
      ruleDef   := theRule;
    end;
  end;
  CreateCTStatement := temp;
end; (* CREATECTSTATEMENT *)


(*****************************************************************************)
(* APPENDCTCOLUMN (10-DEC-1990)                                              *)
(* Appends a CREATE TABLE column definition to the list.                     *)
(*                                                                           *)
(* Called from: YYLEX                                                        *)
(*****************************************************************************)
Function APPENDCTCOLUMN ( theColumnDef, theList : CTColumn_ptr) : CTColumn_ptr;
Var      temp : CTColumn_ptr;
Begin
  if (theList = nil) then theList := theColumnDef
  else
  begin
    temp := theList;
    while (temp^.next <> nil) do temp := temp^.next;
    temp^.next := theColumnDef;
  end;
  AppendCTColumn := theList;
  temp           := nil;
end; (* APPENDCTCOLUMN *)


(*****************************************************************************)
(* CREATECTCOLUMN (10-DEC-1990)                                              *)
(* Creates a CTColumn_ptr and returns it.                                    *)
(*                                                                           *)
(* Called from: YYLEX                                                        *)
(*****************************************************************************)
Function CREATECTCOLUMN ( colName, dataType : String;
                          theRest           : BigString ) : CTColumn_ptr;
Var      temp : CTColumn_ptr;
Begin
  new(temp);
  temp^.theColumn  := colName;
  temp^.theType    := dataType;
  temp^.otherStuff := theRest;
  temp^.next       := nil;
  CreateCTColumn   := temp;
  temp             := nil;
end; (* CREATECTCOLUMN *)


%}


%start statement


%union
{
  ( stringValue     : String );
  ( bigStringValue  : BigString );
  ( integerValue    : integer );
  ( selectPtrValue  : Select_ptr );
  ( selectStmtValue : SelectStmt_ptr );
  ( SQLStmtValue    : SQLStatement );
  ( fromPtrValue    : From_ptr );
  ( wherePtrValue   : Where_ptr );
  ( groupPtrValue   : Group_ptr );
  ( orderPtrValue   : Order_ptr );
  ( exprPtrValue    : Expr_ptr );
  ( CTPtrValue      : CTColumn_ptr );
  ( clausePtrValue  : Clause_ptr );
  ( predPtrValue    : Pred_ptr );
  ( plistPtrValue   : Plist_ptr );
}


%token  CREATE  SCHEMA  AUTHORIZATION   DEDUCTION       OFF     ON      TABLE
%token  NOT_W   NULL    UNIQUE  VIEW    AS      WITH_W  ALL     SELECT
%token  INSERT  DELETE  UPDATE  RULE    FROM    COMMIT  WORK    WHERE   INTO
%token  VALUES  ROLLBACK        SET_W   DISTINCT        UNION   GROUP   BY
%token  HAVING  ORDER   ASC     DESC    BETWEEN LIKE    IS      AND_W   OR_W
%token  IN_W    ANY     SOME    EXISTS  USER    COUNT   AVG     MAX_W   MIN_W
%token  SUM     CHRTYPE LONG    VCHARTYPE       SMLINTTYPE      INTTYPE
%token  QUADTYPE        DECTYPE NUMTYPE         FLTTYPE REALTYPE        SYSTEM
%token  DATETYPE        NUMBER  FOR_W   FILENAME        DECLARE CONTAINING
%token  STARTING        UNKNOWN TABLES  SHOW    RULES   NOT_W   KEY
%token  BASE    VIRTUAL ONLY    IF_W    THEN_W  DOUBLETYPE      PRIMARY
%token <stringValue> IDENTIFIER CHAR_LITERAL NUMBER
```

```
%type <stringValue>     alias fn_name literal numeric_literal exponential
%type <stringValue>     relop optional_not multiplicity dup
%type <stringValue>     direction data_type whichtables
%type <bigStringValue>  col_constraint
%type <selectPtrValue>  sel_list select_item
%type <fromPtrValue>    from_list from_item
%type <wherePtrValue>   where_cl having_cl
%type <exprPtrValue>    value_expr term term_element function count_function
%type <exprPtrValue>    other_function column_name predicate_expr predicate
%type <exprPtrValue>    basic_pred between_pred containing_pred exists_pred
%type <exprPtrValue>    in_pred like_pred null_pred quantified_pred
%type <exprPtrValue>    starting_with_pred unique_pred in_list
%type <selectStmtValue> col_select select_expr
%type <SQLStmtValue>    select_statement definition create_table
%type <orderPtrValue>   order_cl order_list order_item
%type <groupPtrValue>   group_cl g_column_list
%type <CTPtrValue>      table_element_list table_element column_def
%type <clausePtrValue>  from_rule rule_def
%type <predPtrValue>    and_list or_list table_literal table_def
%type <plistPtrValue>   rule_column_list


%left   OR_W
%left   AND_W
%left   NOT_W
%nonassoc       '='     '<'     '>'     LEQ    GEQ    NEQ
%left   '+'     '-'
%left   '*'     '/'
%left   UNARY


%%

statement: schema
           {
             expandIt := false;
           }
           | definition
           {
             begin
               currentStmt := $1;
               expandIt    := true;  (* not strictly correct, but who cares *)
             end;
           }
           | manipulative
           {
             expandIt := true;
           }
           | select_statement
           {
             begin
               currentStmt := $1;
               expandIt    := true;
             end;
           }
           | misc
           {
             expandIt := false;
           }
           ;


/* Schema statements */

schema: create_schema
        | declare_schema
        ;

create_schema: CREATE SCHEMA auth root_file ';';

auth: | AUTHORIZATION IDENTIFIER;

root_file: | FILENAME file_spec;

declare_schema: DECLARE SCHEMA auth FILENAME file_spec ';'
                {
                  read_in_tables := true;
                }
                | DECLARE SCHEMA auth FOR_W FILENAME file_spec ';'
                {
                  read_in_tables := true;
                }
                ;


/* Definition statements */

definition: create_table
            {
              $$ := $1;
            }
            | create_view
            ;

create_table: CREATE TABLE IDENTIFIER '(' table_element_list ')' from_rule ';'
              {
                $$ := CreateCTStatement($3, $5, $7);
              }
              ;

table_element_list: table_element_list ',' table_element
                    {
                      $$ := AppendCTColumn($3, $1);
                    }
                    | table_element
                    {
                      $$ := $1;
                    }
                    ;
```

```
table_element: column_def  /* also includes table-constraint (this will be added at a later date) */
               {
                 $$ := $1;
               }
               ;

column_def: IDENTIFIER data_type col_constraint
            {
              $$ := CreateCTColumn($1, $2, $3);
            }
            ;

col_constraint: {
                  $$ := '';
                }
                | NOT_W NULL            /* also includes CHECK and REFERENCES - will be added later */
                {
                  $$ := 'NOT NULL';
                }
                | UNIQUE
                {
                  $$ := 'UNIQUE';
                }
                | PRIMARY KEY
                {
                  $$ := 'PRIMARY KEY';
                }
                ;

from_rule: {
             $$ := nil;
           }
           | FROM RULE rule_def
           {
             $$ := $3;
           }
           ;

rule_def: IF_W and_list THEN_W or_list
          {
            $$ := Make_Clause($2, $4);
          }
          ;

and_list: {
            $$ := nil;
          }
          | and_list AND_W table_literal
          {
            $$ := Add_Pred($1, $3);
          }
          | table_literal
          {
            $$ := $1;
          }
          ;

table_literal: NOT_W table_def
               {
                 $$ := Negate($2);
               }
               | table_def
               {
                 $$ := $1;
               }
               ;

table_def: IDENTIFIER '(' rule_column_list ')'
           {
             $$ := Make_Pred(Upcase($1), $3, count_arity($3));
           }
           ;

rule_column_list: rule_column_list ',' IDENTIFIER
                  {
                    $$ := Add_Parameter($3, $1);
                  }
                  | IDENTIFIER
                  {
                    $$ := Add_Parameter($1, nil);
                  }
                  ;

or_list: {
           $$ := nil;
         }
         | or_list OR_W table_def
         {
           $$ := Add_Pred($1, $3);
         }
         | table_def
         {
           $$ := $1;
         }
         ;

create_view: CREATE VIEW IDENTIFIER columns AS select_expr ';'
             {
(*             insert_tablename($3) *);
             }
             ;

columns: | '(' column_list ')';
```

```
/* Manipulative statements */

manipulative: delete_statement
              | insert_statement
              | update_statement
              ;

delete_statement: DELETE FROM IDENTIFIER alias where_cl base_cl ';' ;

base_cl: | BASE ONLY
         | VIRTUAL ONLY
         ;

insert_statement: INSERT INTO IDENTIFIER columns insert_def opt_base ';' ;

insert_def: VALUES '(' insert_list ')'
            | select_expr
            ;

insert_list: insert_list ',' insert_item
             | insert_item
             ;

insert_item: literal
             | '-' literal %prec UNARY
             | '+' literal %prec UNARY
             | NULL
             ;

opt_base: | BASE ONLY;

update_statement: UPDATE IDENTIFIER alias SET_W ass_lst where_cl opt_base ';' ;

ass_lst: ass_lst ',' assignment
         | assignment
         ;

assignment: IDENTIFIER '=' value_expr
            | IDENTIFIER '=' NULL
            ;


/* Query statements */

select_statement: select_expr order_cl ';'
                  {
                    begin
                      $$ := CreateSelectStatement(AddOrderClause($2, $1));
                    end;
                  }
                  ;


/* Misc statements */

misc: showtables
      {
        sql_send := false;
      }
      | showrules
      {
        sql_send := false;
      }
      | commit_statement
      | rollback_statement
      ;

showtables: SHOW whichtables TABLES
            {
              if ($2 = '') then ShowTables(false, false, false)
              else if ($2 = 'ALL') then ShowTables(false, false, true)
              else if ($2 = 'SYSTEM') then ShowTables(false, true, false);
            }
            | SHOW whichtables TABLES ';'
            {
              if ($2 = '') then ShowTables(false, false, false)
              else if ($2 = 'ALL') then ShowTables(false, false, true)
              else if ($2 = 'SYSTEM') then ShowTables(false, true, false);
            }
            ;

showrules: SHOW RULES
           {
             ShowTables(true, false, false);
           }
           | SHOW RULES ';'
           {
             ShowTables(true, false, false);
           }
           ;

whichtables: {
               $$ := '';
             }
             | ALL
             {
               $$ := 'ALL';
             }
             | SYSTEM
             {
               $$ := 'SYSTEM';
             }
             ;

commit_statement: COMMIT ';'
                  | COMMIT WORK ';'
                  ;
```

```
rollback_statement: ROLLBACK ';'
                  | ROLLBACK WORK ';'
                  ;

/* value_expr definitions */

value_expr: value_expr '+' term
            {
              $$ := MakeOprNode('+', $1, $3);
            }
            | value_expr '-' term
            {
              $$ := MakeOprNode('-', $1, $3);
            }
            | value_expr '*' term
            {
              $$ := MakeOprNode('*', $1, $3);
            }
            | value_expr '/' term
            {
              $$ := MakeOprNode('/', $1, $3);
            }
            | term
            {
              $$ := $1;
            }
            ;

term: '+' term_element %prec UNARY
      {
        $$ := MakeOprNode('+', nil, $2);
      }
      | '-' term_element %prec UNARY
      {
        $$ := MakeOprNode('-', nil, $2);
      }
      | term_element
      {
        $$ := $1;
      }
      ;

term_element: '(' col_select ')'
              {
                $$ := MakeSubselNode($2);
              }
              | '(' value_expr ')'
              {
                $$ := MakeFnNode('', $2);
              }
              | column_name
              {
                $$ := $1;
              }
              | function
              {
                $$ := $1;
              }
              | literal
              {
                $$ := MakeLitNode($1);
              }
              ;

function: count_function
          {
            $$ := $1;
          }
          | other_function
          {
            $$ := $1;
          }
          ;

count_function: COUNT '(' '*' ')'
                {
                  $$ := MakeFnNode('COUNT', MakeCnameNode('', '*'));
                }
                | COUNT '(' DISTINCT column_name ')'
                {
                  $$ := MakeFnNode('COUNT DISTINCT', $4);
                }
                ;

other_function: fn_name '(' DISTINCT column_name ')'
                {
                  $$ := MakeFnNode($1, MakeOprNode('DISTINCT', nil, $4));
                }
                | fn_name '(' ALL value_expr ')'
                {
                  $$ := MakeFnNode($1, MakeOprNode('ALL', nil, $4));
                }
                | fn_name '(' value_expr ')'
                {
                  $$ := MakeFnNode($1, $3);
                }
                ;
```

```
fn_name: SUM
         {
           $$ := 'SUM';
         }
         | AVG
         {
           $$ := 'AVG';
         }
         | MAX_W
         {
           $$ := 'MAX';
         }
         | MIN_W
         {
           $$ := 'MIN';
         }
         ;

/* Predicate definitions */

predicate_expr: predicate
                {
                  $$ := $1;
                }
                | '(' predicate_expr ')'
                {
                  $$ := MakeFnNode('', $2);
                }
                | NOT_W predicate_expr
                {
                  $$ := MakeOprNode('NOT', nil, $2);
                }
                | predicate_expr AND_W predicate_expr
                {
                  $$ := MakeOprNode('AND', $1, $3);
                }
                | predicate_expr OR_W predicate_expr
                {
                  $$ := MakeOprNode('OR', $1, $3);
                }
                ;

predicate: basic_pred
           {
             $$ := $1;
           }
           | between_pred
           {
             $$ := $1;
           }
           | containing_pred
           {
             $$ := $1;
           }
           | exists_pred
           {
             $$ := $1;
           }
           | in_pred
           {
             $$ := $1;
           }
           | like_pred
           {
             $$ := $1;
           }
           | null_pred
           {
             $$ := $1;
           }
           | quantified_pred
           {
             $$ := $1;
           }
           | starting_with_pred
           {
             $$ := $1;
           }
           | unique_pred
           {
             $$ := $1;
           }
           ;

basic_pred: value_expr relop value_expr
            {
              $$ := MakeOprNode($2, $1, $3);
            }
            ;
```

```
relop: '='
       {
         $$ := '=';
       }
       | NEQ
       {
         $$ := '<>';
       }
       | '<'
       {
         $$ := '<';
       }
       | '>'
       {
         $$ := '>';
       }
       | LEQ
       {
         $$ := '<=';
       }
       | GEQ
       {
         $$ := '>=';
       }
       ;

between_pred: value_expr optional_not BETWEEN value_expr AND_W value_expr
              {
                if ($2 = '') then $$ := MakeOprNode('BETWEEN', $1, MakeOprNode('AND', $4, $6))
                else $$ := MakeOprNode('NOT BETWEEN', $1, MakeOprNode('AND', $4, $6));
              }
              ;

containing_pred: value_expr optional_not CONTAINING value_expr
                 {
                   if ($2 = '') then $$ := MakeOprNode('CONTAINING', $1, $4)
                   else $$ := MakeOprNode('NOT CONTAINING', $1, $4);
                 }
                 ;

like_pred: value_expr optional_not LIKE CHAR_LITERAL
           {
             if ($2 = '') then $$ := MakeOprNode('LIKE', $1, MakeLitNode($4))
             else $$ := MakeOprNode('NOT LIKE', $1, MakeLitNode($4));
           }
           ;

null_pred: value_expr IS optional_not NULL
           {
             if ($3 = '') then $$ := MakeOprNode('IS NULL', $1, nil)
             else $$ := MakeOprNode('IS NOT NULL', $1, nil);
           }
           ;

in_pred: value_expr optional_not IN_W value_expr
         {
           if ($2 = '') then $$ := MakeOprNode('IN', $1, $4)
           else $$ := MakeOprNode('NOT IN', $1, $4);
         }
         | value_expr optional_not IN_W '(' in_list ')'
         {
           if ($2 = '') then $$ := MakeOprNode('IN', $1, $5)
           else $$ := MakeOprNode('NOT IN', $1, $5);
         }
         ;

in_list: in_list ',' col_select
         {
           $$ := AddExprItem(MakeSubselNode($3), $1);
         }
         | in_list ',' value_expr
         {
           $$ := AddExprItem($3, $1);
         }
         | col_select
         {
           $$ := AddExprItem(MakeSubselNode($1), nil);
         }
         | value_expr
         {
           $$ := AddExprItem($1, nil);
         }
         ;

quantified_pred: value_expr relop multiplicity '(' col_select ')'
                 {
                   $$ := MakeOprNode($2 + $3, $1, MakeSubselNode($5));
                 }
                 ;

multiplicity: ALL
              {
                $$ := ' ALL';
              }
              | ANY
              {
                $$ := ' ANY';
              }
              | SOME
              {
                $$ := ' SOME';
              }
              ;

exists_pred: EXISTS '(' col_select ')'
             {
               $$ := MakeOprNode('EXISTS', nil, MakeSubselNode($3));
             }
             ;
```

```
starting_with_pred: value_expr optional_not STARTING WITH_W value_expr
                    {
                      if ($2 = '') then $$ := MakeOprNode('STARTING WITH', $1, $5)
                      else $$ := MakeOprNode('NOT STARTING WITH', $1, $5);
                    }
                    ;

unique_pred: UNIQUE '(' col_select ')'
             {
               $$ := MakeOprNode('UNIQUE', nil, MakeSubselNode($3));
             }
             ;


/* select_expr definitions */

select_expr: SELECT dup sel_list FROM from_list where_cl group_cl having_cl
             {
               $$ := CreateSelectExpr($2, $3, $5, $6, $7, $8);
             }

dup: {
       $$ := '';
     }
     | ALL
     {
       $$ := 'ALL';
     }
     | DISTINCT
     {
       $$ := 'DISTINCT';
     }
     ;

sel_list: sel_list ',' select_item
          {
            $$ := add_select_item($3, $1);
          }
          | select_item
          {
            $$ := add_select_item($1, nil);
          }
          ;

select_item: value_expr
             {
               $$ := ExprToSelect($1);
             }

/*           (* this may be added back in at some stage *)
             | IDENTIFIER '.' '*'
             {
               begin
                 str$trim_v($1, $1,);
                 $$ := BuildSelItem($1, '*');
               end;
             }
*/

             | '*'
             {
               $$ := BuildSelItem('', '*');
             }
             ;

from_list: from_list ',' from_item
           {
             $$ := add_from_table($3, $1);
           }
           | from_item
           {
             $$ := add_from_table($1, nil);
           }
           ;

from_item: IDENTIFIER alias
           {
             begin
               str$trim_v($1, $1,);
               $$ := BuildFromItem($1, '');  (* ignore aliases at the moment... *)
             end;
           }
           ;

where_cl: {
            $$ := nil;
          }
          | WHERE predicate_expr
          {
            $$ := ExprToWhere($2);
          }
          ;

group_cl: {
            $$ := nil;
          }
          | GROUP BY g_column_list
          {
            $$ := $3;
          }
          ;
```

```
having_cl: {
             $$ := nil;
           }
           | HAVING predicate_expr
           {
             $$ := ExprToWhere($2);
           }
           ;

order_cl: {
             $$ := nil;
          }
          | ORDER BY order_list
          {
            $$ := $3;
          }
          ;

order_list: order_list ',' order_item
            {
              $$ := AddOrderItem($3, $1);
            }
            | order_item
            {
              $$ := AddOrderItem($1, nil);
            }
            ;

order_item: column_name direction
            {
              $$ := MakeOrderItem($1, $2);
            }
/*          | NUMBER direction  (* to be put back later - TO BE CHANGED *)
            {
              $$ := MakeOrderItem($1, $2);
            } */
            ;

direction: {
             $$ := '';
           }
           | ASC
           {
             $$ := 'ASC';
           }
           | DESC
           {
             $$ := 'DESC';
           }
           ;


/* col_select_expr definitions */

col_select: SELECT dup select_item FROM from_list where_cl group_cl having_cl
            {
              $$ := CreateSelectExpr($2, $3, $5, $6, $7, $8);
            }
            ;


/* Column definitions */

g_column_list: g_column_list ',' column_name
               {
                 $$ := AddGroupColumn($3, $1);
               }
               | column_name
               {
                 $$ := AddGroupColumn($1, nil);
               }
               ;

column_list: column_list ',' column_name
             | column_name
             ;

column_name: IDENTIFIER
             {
               $$ := MakeCnameNode('', $1);
             }
             | IDENTIFIER '.' IDENTIFIER
             {
               $$ := MakeCnameNode($1, $3);
             }
             ;


/* Data type definitions */

data_type: CHRTYPE optional_size_1
           | VCHARTYPE '(' NUMBER ')'
           | LONG VCHARTYPE
           | SMLINTTYPE optional_size_1
           | INTTYPE optional_size_1
           | QUADTYPE optional_size_1
           | DECTYPE optional_size_2
           | NUMTYPE optional_size_2
           | FLTTYPE optional_size_1
           | REALTYPE
           | DOUBLETYPE
           | DATETYPE
           ;

optional_size_1: | '(' NUMBER ')';

optional_size_2: | '(' NUMBER ')'
                 | '(' NUMBER ',' NUMBER ')'
                 ;
```

```
/* Literal definitions */

literal: numeric_literal
         {
           $$ := $1;
         }
         | CHAR_LITERAL
         {
           $$ := $1;
         }
         ;

numeric_literal: NUMBER exponential
                 {
                   $$ := $1 + $2;
                 }
                 | NUMBER '.' NUMBER exponential
                 {
                   $$ := $1 + '.' + $3 + $4;
                 }
                 | '.' NUMBER exponential
                 {
                   $$ := '.' + $2 + $3;
                 }
                 ;

exponential: {
               $$ := '';
             }
             | 'E' NUMBER
             {
               $$ := 'E' + $2;
             }
             | 'E' '+' NUMBER
             {
               $$ := 'E+' + $3;
             }
             | 'E' '-' NUMBER
             {
               $$ := 'E-' + $3;
             }
             ;


/* Misc */

optional_not: {
                $$ := '';
              }
              | NOT_W
              {
                $$ := 'NOT';
              }
              ;

file_spec: | IDENTIFIER;

alias: | IDENTIFIER;


%%


(*****************************************************************************)
(* PRE_PROCESS (5-SEP-1989)                                                  *)
(* Takes a candidate SQLD statement and parses it. If it finds references to  *)
(* rules, it expands them until they refer only to base tables or views. The  *)
(* modified statement is then returned and processed as a conventional SQL    *)
(* statement by dynamic SQL.                                                 *)
(*****************************************************************************)
Function PRE_PROCESS ( var stmt : st; var theColumns : StringPair_ptr ) : boolean;
Var      success  : boolean;
         tempStmt : SQLStatement;
         semiPos  : integer;
Begin
  success := true;
  scan := 1;
  str$trim(theStatement, stmt, stmt_len);
  if (yyparse <> 1) then  (* KABOOM - syntax error *)
  begin
    message(DDB_BADSQLD);
    success := false;
  end
  else if expandIt then
  begin
    if (tempStmt.stmtKind = SelectStmt) then
      theColumns := tempStmt.selBody^.columnList;  (* TO BE CHANGED *)
    tempStmt   := expand_statement(currentStmt);
    build_statement(tempStmt, stmt);  (* build the new SQL statement *)
  end
  else  (* leave statement as it is, but remove semicolon for dynamic SQL *)
  begin
    semipos := index(stmt, ';');
    if (semipos <> 0) then stmt[semipos] := ' ';
  end;
  pre_process := (success and sql_send);
  sql_send    := true;
end;    (* PRE_PROCESS *)
```

```
(*****************************************************************************)
(* FIND_TYPE (6-SEP-1989)                                                    *)
(* A simple routine, when given a string, figures out which token it is.     *)
(*****************************************************************************)
Function FIND_TYPE (var str : string) : integer;
Begin
  str := upcase(str);
  if (str = 'CREATE') then find_type := CREATE
  else if (str = 'SCHEMA') then find_type := SCHEMA
  else if (str = 'AUTHORIZATION') then find_type := AUTHORIZATION
  else if (str = 'DEDUCTION') then find_type := DEDUCTION
  else if (str = 'OFF') then find_type := OFF
  else if (str = 'ON') then find_type := ON
  else if (str = 'TABLE') then find_type := TABLE
  else if (str = 'NOT') then find_type := NOT_W
  else if (str = 'NULL') then find_type := NULL
  else if (str = 'UNIQUE') then find_type := UNIQUE
  else if (str = 'VIEW') then find_type := VIEW
  else if (str = 'AS') then find_type := AS
  else if (str = 'WITH') then find_type := WITH_W
  else if (str = 'ALL') then find_type := ALL
  else if (str = 'SELECT') then find_type := SELECT
  else if (str = 'INSERT') then find_type := INSERT
  else if (str = 'DELETE') then find_type := DELETE
  else if (str = 'UPDATE') then find_type := UPDATE
  else if (str = 'RULE') then find_type := RULE
  else if (str = 'FROM') then find_type := FROM
  else if (str = 'COMMIT') then find_type := COMMIT
  else if (str = 'WORK') then find_type := WORK
  else if (str = 'WHERE') then find_type := WHERE
  else if (str = 'INTO') then find_type := INTO
  else if (str = 'VALUES') then find_type := VALUES
  else if (str = 'ROLLBACK') then find_type := ROLLBACK
  else if (str = 'DISTINCT') then find_type := DISTINCT
  else if (str = 'SET') then find_type := SET_W
  else if (str = 'UNION') then find_type := UNION
  else if (str = 'GROUP') then find_type := GROUP
  else if (str = 'BY') then find_type := BY
  else if (str = 'HAVING') then find_type := HAVING
  else if (str = 'ORDER') then find_type := ORDER
  else if (str = 'ASC') then find_type := ASC
  else if (str = 'DESC') then find_type := DESC
  else if (str = 'OR') then find_type := OR_W
  else if (str = 'AND') then find_type := AND_W
  else if (str = 'BETWEEN') then find_type := BETWEEN
  else if (str = 'LIKE') then find_type := LIKE
  else if (str = 'IS') then find_type := IS
  else if (str = 'IN') then find_type := IN_W
  else if (str = 'ANY') then find_type := ANY
  else if (str = 'SOME') then find_type := SOME
  else if (str = 'EXISTS') then find_type := EXISTS
  else if (str = 'USER') then find_type := USER
  else if (str = 'COUNT') then find_type := COUNT
  else if (str = 'AVG') then find_type := AVG
  else if (str = 'MAX') then find_type := MAX_W
  else if (str = 'MIN') then find_type := MIN_W
  else if (str = 'SUM') then find_type := SUM
  else if (str = 'COUNT') then find_type := COUNT
  else if (str = 'CHAR') then find_type := CHRTYPE
  else if (str = 'LONG') then find_type := LONG
  else if (str = 'VARCHAR') then find_type := VCHARTYPE
  else if (str = 'SMALLINT') then find_type := SMLINTTYPE
  else if (str = 'INTEGER') then find_type := INTTYPE
  else if (str = 'QUADWORD') then find_type := QUADTYPE
  else if (str = 'DECIMAL') then find_type := DECTYPE
  else if (str = 'NUMERIC') then find_type := NUMTYPE
  else if (str = 'FLOAT') then find_type := FLTTYPE
  else if (str = 'REAL') then find_type := REALTYPE
  else if (str = 'DOUBLE_PRECISION') then find_type := DOUBLETYPE
  else if (str = 'DATE') then find_type := DATETYPE
  else if (str = 'NUMBER') then find_type := NUMBER
  else if (str = 'FOR') then find_type := FOR_W
  else if (str = 'FILENAME') then find_type := FILENAME
  else if (str = 'DECLARE') then find_type := DECLARE
  else if (str = 'CONTAINING') then find_type := CONTAINING
  else if (str = 'STARTING') then find_type := STARTING
  else if (str = 'SHOW') then find_type := SHOW
  else if (str = 'TABLES') then find_type := TABLES
  else if (str = 'RULES') then find_type := RULES
  else if (str = 'BASE') then find_type := BASE
  else if (str = 'VIRTUAL') then find_type := VIRTUAL
  else if (str = 'ONLY') then find_type := ONLY
  else if (str = 'IF') then find_type := IF_W
  else if (str = 'THEN') then find_type := THEN_W
  else if (str = 'PRIMARY') then find_type := PRIMARY
  else if (str = 'KEY') then find_type := KEY
  else if (str = 'SYSTEM') then find_type := SYSTEM
  else find_type := IDENTIFIER;
end;    (* FIND_TYPE *)
```

```
(*****************************************************************************)
(* YYLEX (6-SEP-1989)                                                        *)
(* Lexical analyser for Yacc-generated parser routine (YYPARSE)              *)
(*****************************************************************************)
Function YYLEX;
Var      fini        : boolean;
         temp        : string;
         start_quote : char;
Begin
  if (scan <= stmt_len) then
  begin
    fini := false;
    temp := '';
    while ((scan < stmt_len) and (theStatement[scan] = ' ')) do scan := scan + 1;
    if (theStatement[scan] in alpha) then
    begin
      while ((scan <= stmt_len) and (not fini)) do
        if (theStatement[scan] in id_chars) then
        begin
          temp := temp + theStatement[scan];
          scan := scan + 1;
        end
        else fini := true;
        yylex             := find_type(temp);
        yylval.stringValue := temp;
(*
        writeln('Found token ',yytoks[find_type(temp) - CREATE].t_name,' ending at position ',scan:1);
*)
    end
    else if (theStatement[scan] in quotes) then
    begin
      start_quote := theStatement[scan];
      scan := scan + 1;
      while ((scan <= stmt_len) and (not fini)) do
        if (theStatement[scan] = start_quote) then fini := true
        else
        begin
          temp := temp + theStatement[scan];
          scan := scan + 1;
        end;
      if (theStatement[scan] = start_quote) then
      begin
        yylex             := CHAR_LITERAL;
        yylval.stringValue := '"' + temp + '"';
(*
        writeln('Found token CHAR_LITERAL ending at position ',scan:1);
*)
      end
      else
      begin
        yylex := UNKNOWN;
(*
        writeln('Found token UNKNOWN ending at position ',scan:1);
*)
      end;
      scan := scan + 1;
    end
    else if (theStatement[scan] in numeric) then
    begin
      while ((scan <= stmt_len) and (not fini)) do
      begin
        if not (theStatement[scan] in numeric) then fini := true
        else
        begin
          temp := temp + theStatement[scan];
          scan := scan + 1;
        end;
      end;
      yylex             := NUMBER;
      yylval.stringValue := temp;
(*
      writeln('Found token NUMBER ending at position ',scan:1);
*)
    end
    else if (theStatement[scan] = '<') then
    begin
      scan := scan + 1;
      if (theStatement[scan] = '=') then
      begin
        yylex := LEQ;
(*
        writeln('Found token LEQ ending at position ',scan:1);
*)
      end
      else if (theStatement[scan] = '>') then
      begin
        yylex := NEQ;
(*
        writeln('Found token NEQ ending at position ',scan:1);
*)
      end
      else
      begin
        yylex := ord('<');
(*
        writeln('Found token < ending at position ',scan:1);
*)
      end;
      scan := scan + 1;
    end
    else if (theStatement[scan] = '>') then
```

```
    begin
      scan := scan + 1;
      if (theStatement[scan] = '=') then
      begin
        yylex := GEQ;
(*
        writeln('Found token GEQ ending at position ',scan:1);
*)
      end
      else
      begin
        yylex := ord('>');
(*
        writeln('Found token > ending at position ',scan:1);
*)
      end;
      scan := scan + 1;
    end
    else
    begin
      yylex := ord(theStatement[scan]);
(*
      writeln('Found token ',theStatement[scan],' ending at position ',scan:1);
*)
      scan := scan + 1;
    end;
  end
  else yylex := -1; (* end-of-input *)
end;    (* YYLEX *)


END.    (* SQL_PARSER *)
```

D.7 DYNAMIC.PAS

```
[ INHERIT ('GLOBAL',
           'SYS$LIBRARY:STARLET',
           'SYS$LIBRARY:PASCAL$STR_ROUTINES'),
  ENVIRONMENT ('DYNAMIC') ]


MODULE DYNAMIC_SQL (input,output);


(******************************************************************************)
(* DYNAMIC.PAS - N. Stanger, July 1989.                                       *)
(*                                                                            *)
(* This module contains the interface between SQLD and SQL. It uses dynamic   *)
(* SQL as described in Chapter 15 of the VAX SQL User's Guide. This module    *)
(* was converted from the original sample program written in Ada.             *)
(******************************************************************************)


CONST   maxparms      = 20;     (* arbitrary = max. no. of variables in SQLDA *)

        SQL_SUCCESS   = 0;      (* SQL return status codes *)
        STREAM_EOF    = 100;
        DEADLOCK      = -913;
        LOCK_CONFLICT = -1003;

        quoteComma    = '", "';


TYPE    indbuf_ptr = ^$word;
        buf_ptr    = ^buf_rec;
        buf_rec    = record case integer of
                       0 : (chabuf    : str255;);
                       1 : (chvbuf    : varchar;);
                       2 : (intbuf    : integer;);
                       3 : (smlintbuf : $word;);   (* I think... *)
                       4 : (fltbuf    : real;);
                       5 : (datbuf    : date_time;);
                     end;
        sqlvar_rec = record
                       sqltype, sqllen : $word;
                       sqldata         : buf_ptr;
                       sqlind          : indbuf_ptr;
                       sqlname         : varying[30] of char;
                     end;

        sqlda_ptr  = ^sqlda_rec;
        sqlda_rec  = record
                       sqldaid    : packed array[1..8] of char; (* = 'SQLDA   ' *)
                       sqldabc    : integer;
                       sqln, sqld : $word;
                       sqlvar     : array[1..maxparms] of sqlvar_rec;
                     end;


VAR     sql_return_status       : integer;
        buffer                  : buf_ptr;
        indbuf                  : indbuf_ptr;
        read_in_tables          : boolean := false;


[EXTERNAL] Procedure STCVTIS ( var dest : varying [m1] of char;  (* convert integer to string *)
                               val, len : integer ); external;


[EXTERNAL] Procedure STCVTRS ( var dest : varying [m1] of char;  (* convert real to string *)
                               val      : real;
                               len, dec : integer); external;


[EXTERNAL] Procedure SQL$GET_ERROR_TEXT (var text : [CLASS_S] str255;
                                         var len  : $word); external;


[EXTERNAL] Procedure SQL$SIGNAL; external;


Procedure PREPARE_STMT (var sqlcode : integer;
                        var sqlda   : sqlda_rec;
                        stmt        : st); external;


Procedure DESCRIBE_PARM (var sqlcode : integer;
                         var sqlda   : sqlda_rec); external;


Procedure EXECUTE_STMT (var sqlcode : integer;
                        var sqlda   : sqlda_rec); external;


Procedure OPEN_CURSOR (var sqlcode : integer;
                       var sqlda   : sqlda_rec); external;


Procedure FETCH_ROW (var sqlcode : integer;
                     var sqlda   : sqlda_rec); external;


Procedure CLOSE_CURSOR (var sqlcode : integer); external;


Procedure RELEASE_STMT (var sqlcode : integer ); external;
```

```
Procedure COMMIT_TRANSACTION (var sqlcode : integer); external;


Procedure ROLLBACK_TRANSACTION (var sqlcode : integer); external;


Procedure ERROR;
Var       error_text : str255;
          error_len  : $word;
Begin
  sql$get_error_text(error_text,error_len);
  writeln(error_text:error_len);
  writeln;
end;    (* ERROR *)


(*****************************************************************************)
(* INIT_SQLDA (03-DEC-1990)                                                  *)
(* Creates, initialises and returns an sqlda_ptr variable.                   *)
(*****************************************************************************)
Function INIT_SQLDA : sqlda_ptr;
Var      temp : sqlda_ptr;
Begin
  new(temp);
  temp^.sqln    := maxparms;
  temp^.sqldaid := 'SQLDA   ';
  Init_Sqlda    := temp;
  temp          := nil;
end; (* INIT_SQLDA *)


(*****************************************************************************)
(* KILL_SQLDA (14-DEC-1990)                                                  *)
(* Destroys a Sqlda_ptr and deallocates all memory allocated to it.          *)
(*****************************************************************************)
Procedure KILL_SQLDA ( var theSqlda : Sqlda_ptr );
Var       i : Integer;
Begin
  if (theSqlda <> nil) then
  begin
    for i := 1 to maxparms do with theSqlda^.sqlvar[i] do
    begin
      if (sqldata <> nil) then dispose(sqldata);
      if (sqlind <> nil) then dispose(sqlind);
    end;
    dispose(theSqlda);
    theSqlda := nil;
  end;
end;


(*****************************************************************************)
(* GET_IN_PARMS                                                              *)
(*                                                                           *)
(* GET_IN_PARMS allocates storage for parameter markers in the statement     *)
(* string supplied by the user.  It also prompts the user for values to place *)
(* in that storage, assigns the values to the storage, and places addresses  *)
(* of the storage in SQLDA_IN.                                               *)
(*****************************************************************************)
Procedure GET_IN_PARMS ( var sqlda_in, sqlda_out : sqlda_ptr );
Var       parm          : integer;
          i             : integer;
          temp_str      : string;
          value_entered : boolean;
          date          : date_string;
Begin
  for parm := 1 to sqlda_in^.sqld do with sqlda_in^.sqlvar[parm] do
  begin
    value_entered := false;
    case sqltype of
      449 : repeat                  (* VARCHAR variable *)
              writeln('Enter value for ',sqlname);
              write('Maximum length is ',sqllen:4,' characters> ');
              new(buffer);
              readln(buffer^.chvbuf);
              writeln;
              if (length(buffer^.chvbuf) > 0) then
              begin
                sqldata := buffer;
                value_entered := true;
              end
              else writeln('Value required.  Please re-enter.');
            until value_entered;
      453 : repeat                  (* CHAR variable *)
              writeln('Enter value for ',sqlname);
              write('Maximum length is ',sqllen:4,' characters> ');
              new(buffer);
              readln(temp_str);
              writeln;
              if (length(temp_str) > 0) then with buffer^ do
              begin
                for i := 1 to length(temp_str) do chabuf[i] := temp_str[i];
                for i := length(temp_str) to 255 do chabuf[i] := ' ';
                sqldata := buffer;
                value_entered := true;
              end
              else
                writeln('Value required.  Please re-enter.');
            until value_entered;
```

```
      481 : repeat                  (* FLOAT variable *)
              writeln('Enter value for ',sqlname);
              write(' Floating-point value> ');
              new(buffer);
              readln(buffer^.fltbuf);
              writeln;
              if (buffer^.fltbuf <> 0.0) then
              begin
                sqldata := buffer;
                value_entered := true;
              end
              else writeln('Value required.  Please re-enter.');
            until value_entered;
      497 : repeat                  (* INTEGER variable *)
              writeln('Enter value for ',sqlname);
              write(' Integer value> ');
              new(buffer);
              readln(buffer^.intbuf);
              writeln;
              if (buffer^.intbuf <> 0) then
              begin
                sqldata := buffer;
                value_entered := true;
              end
              else writeln('Value required.  Please re-enter.');
            until value_entered;
      501 : repeat                  (* SMALLINT variable *)
              writeln('Enter value for ',sqlname);
              write(' Smallint value> ');
              new(buffer);
              readln(buffer^.smlintbuf);
              writeln;
              if (buffer^.smlintbuf <> 0) then
              begin
                sqldata := buffer;
                value_entered := true;
              end
              else writeln('Value required.  Please re-enter.');
            until value_entered;
      503 : begin                   (* DATE variable *)
              writeln('Enter value for ',sqlname);
              write(' Date value in dd-MMM-yyyy:hh:mm:ss.hh format> ');
              readln(date);
              writeln;
              new(buffer);
              $bintim(date,buffer^.datbuf);
              sqldata := buffer;
            end;
      otherwise begin
                  write('An unexpected error occurred (GET_IN_PARMS).  The value ');
                  writeln(sqlda_out^.sqlvar[parm].sqltype:4);
                  writeln('was returned for SQL type.');
                end;
    end;
  end;
end;    (* GET_IN_PARMS *)


(*****************************************************************************)
(* ALLOCATE_BUFFERS                                                          *)
(*                                                                           *)
(* ALLOCATE_BUFFERS allocates storage for select list items in the statement  *)
(* string supplied by the user.  It also allocates storage for indicator     *)
(* parameters associated with the select list items.                         *)
(*****************************************************************************)
Procedure ALLOCATE_BUFFERS (var sqlda_out : sqlda_ptr );
Var       parm, i : integer;
Begin
  for parm := 1 to sqlda_out^.sqld do with sqlda_out^.sqlvar[parm] do
  begin
    new(indbuf);
    sqlind := indbuf;
    case sqltype of
      449 : begin                   (* VARCHAR variable *)
              new(buffer);
              buffer^.chvbuf := '';
              sqldata := buffer;
            end;
      453 : begin                   (* CHAR variable *)
              new(buffer);
              for i := 1 to 255 do buffer^.chabuf[i] := ' ';
              sqldata := buffer;
            end;
      481 : begin                   (* FLOAT variable *)
              new(buffer);
              buffer^.fltbuf := 0.0;
              sqldata := buffer;
            end;
      497 : begin                   (* INTEGER variable *)
              new(buffer);
              buffer^.intbuf := 0;
              sqldata := buffer;
            end;
      501 : begin                   (* SMALLINT variable *)
              new(buffer);
              buffer^.smlintbuf := 0;
              sqldata := buffer;
            end;
      503 : begin                   (* DATE variable *)
              new(buffer);
              buffer^.datbuf := zero;
              sqldata := buffer;
            end;
```

```
      otherwise begin
                  write('An unexpected error occurred (ALLOCATE_BUFFERS).  The value ');
                  writeln(sqltype:4);
                  writeln('was returned for SQL type.');
                end;
    end;
  end;
end;    (* ALLOCATE_BUFFERS *)


(*****************************************************************************)
(* DISPLAYCOLUMNTITLES (14-JUL-1990)                                         *)
(* Displays the 'original' (i.e. virtual) names of the columns in the query.  *)
(*****************************************************************************)
Procedure DISPLAYCOLUMNTITLES ( theColumns : StringPair_ptr );
Var       lineLen : integer value 0;
          outStr  : string;
Begin
  while (theColumns <> nil) do with theColumns^ do
  begin
    if (str1 <> '') then outStr := str1 + '.' + str2
    else outStr := str2;
    if (length(outStr) > 20) then write(substr(outStr, 1, 19), ' ')
    else write(pad(outStr, ' ', 20));
    lineLen := lineLen + 20;
    if (lineLen > 255) then writeln;  (* safety measure *)
    theColumns := theColumns^.next;
  end;
  writeln;
end; (* DISPLAYCOLUMNTITLES *)


(*****************************************************************************)
(* DISPLAY_ROW                                                               *)
(*                                                                           *)
(* DISPLAY_ROW reads from SQLDA_OUT the addresses for storage allocated in   *)
(* the ALLOCATE_BUFFERS procedure. It displays the name and value of each    *)
(* column on the terminal.                                                   *)
(*****************************************************************************)
Procedure DISPLAY_ROW ( var sqlda_out : sqlda_ptr );
Const     nullStr = 'NULL';
Var       parm   : integer value 0;
          st_buf : st;
          date   : date_string value 'DD-MMM-YYYY';
          outStr : string value '';
Begin
  for parm := 1 to sqlda_out^.sqld do with sqlda_out^.sqlvar[parm] do
  begin
    case sqltype of
      449 : begin                   (* VARCHAR variable *)
              if (sqlind^ < 0) then outStr := nullStr  (* Null value *)
              else outStr := sqldata^.chvbuf;
            end;
      453 : begin                   (* CHAR variable *)
              if (sqlind^ < 0) then outStr := nullStr
              else
              begin
                st_buf := sqldata^.chabuf;
                if (sqllen > 255) then outStr := substr(st_buf, 1, 255)
                else outStr := substr(st_buf, 1, sqllen);
              end;
            end;
      481 : begin                   (* FLOAT variable *)
              if (sqlind^ < 0) then outStr := nullStr
              else stcvtrs(outStr, sqldata^.fltbuf, 19, 10);
            end;
      497 : begin                   (* INTEGER variable *)
              if (sqlind^ < 0) then outStr:= nullStr
              else stcvtis(outStr, sqldata^.intbuf, 19);
            end;
      501 : begin                   (* SMALLINT variable *)
              if (sqlind^ < 0) then outStr := nullStr
              else stcvtis(outStr, sqldata^.smlintbuf, 19);
            end;
      503 : begin                   (* DATE variable *)
              if (sqlind^ < 0) then outStr := nullStr
              else
              begin
                $asctim(timbuf := date, timadr := sqldata^.datbuf);
                outStr := date;
              end;
            end;
      otherwise begin
                  write('An unexpected error occurred (DISPLAY_ROW). The value ');
                  writeln(sqltype:4);
                  writeln('was returned for SQL type.');
                end;
    end; (* CASE *)
    if (length(outStr) > 20) then write(substr(outStr, 1, 19), ' ')
    else write(pad(outStr, ' ', 20));
  end;
  writeln;
end;    (* DISPLAY_ROW *)
```

```
(*************************************************************************)
(* SHOWTABLES (13-DEC-1990)                                              *)
(* Implements the SHOW TABLE and SHOW RULE statements. Reads information from *)
(* the system catalog tables.                                            *)
(*                                                                       *)
(* Called from: SQLPARSE.YAC : YYLEX                                     *)
(*************************************************************************)
Procedure SHOWTABLES ( showVirtual, showSystem, showAll : Boolean );
Var       the_stmt           : st;
          temp               : table_ptr;
          sqlda_in, sqlda_out : sqlda_ptr;
Begin
  if showAll then the_stmt := 'SELECT RDB$RELATION_NAME FROM RDB$RELATIONS'  (* show all tables *)
  else if showSystem then
    the_stmt := 'SELECT RDB$RELATION_NAME FROM RDB$RELATIONS WHERE RDB$SYSTEM_FLAG = 1'  (* show system tables only *)
  else if showVirtual then the_stmt := 'SELECT DISTINCT POSTPLACE FROM PTP'  (* show virtual tables only *)
  else the_stmt := 'SELECT RDB$RELATION_NAME FROM RDB$RELATIONS WHERE RDB$SYSTEM_FLAG = 0';
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  allocate_buffers(sqlda_out);
  open_cursor(sql_return_status, sqlda_in^);
  fetch_row(sql_return_status, sqlda_out^);
  while (sql_return_status = SQL_SUCCESS) do with sqlda_out^.sqlvar[1] do
  begin
    case sqltype of
      453 : writeln(substr(sqldata^.chabuf, 1, sqllen));
      otherwise begin
                  write('An unexpected error occurred (SHOWTABLES). The value ');
                  writeln(sqltype:4);
                  writeln('was returned for SQL type.');
                end;
    end;
    fetch_row(sql_return_status, sqlda_out^);
    if (    (sql_return_status <> SQL_SUCCESS)
        and (sql_return_status <> STREAM_EOF)) then error;
  end;
  close_cursor(sql_return_status);
  release_stmt(sql_return_status);
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end; (* SHOWTABLES *)


(*************************************************************************)
(* INSERT_PCN (03-DEC-1990)                                              *)
(* Inserts a PCN descriptor into the PCN list.                           *)
(*                                                                       *)
(* Called from: SEARCHPCN                                                *)
(*************************************************************************)
Function INSERT_PCN ( theID, preP : String;
                      nCols       : integer;
                      theList     : PCN_ptr ) : PCN_ptr;
Var      temp1, temp2 : PCN_ptr;
Begin
  if (theList = nil) then
  begin
    new(theList);
    with theList^ do
    begin
      id       := theID;
      prePlace := preP;
      preCols  := nCols;
      next     := nil;
    end;
  end
  else
  begin
    temp1 := theList;
    while (temp1^.next <> nil) do temp1 := temp1^.next;
    new(temp2);
    with temp2^ do
    begin
      id       := theID;
      prePlace := preP;
      preCols  := nCols;
      next     := nil;
    end;
    temp1^.next := temp2;
  end;
  Insert_PCN := theList;
  temp1      := nil;
  temp2      := nil;
end; (* INSERT_PCN *)


(*************************************************************************)
(* SEARCHPCN (03-DEC-1990)                                               *)
(* Searches the PTP relation for all occurences of a rule name. Returns a     *)
(* list of all occurrences found.                                        *)
(*                                                                       *)
(* Called from: RULES.PAS : LOADRULE                                     *)
(*************************************************************************)
Function SEARCHPCN ( ruleName : String ) : PCN_ptr;
Var      sqlda_in, sqlda_out : sqlda_ptr value nil;
         the_stmt            : st;
         pPlace, theID       : String value '';
         inLabel, outLabel   : String value '';
         thePCNs             : PCN_ptr value nil;
         pCols               : integer;
         temp                : Str255;
         out_len             : $uword value 0;
Begin
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  the_stmt  := 'SELECT TRANS_ID, PREPLACE, PRECOLUMNS FROM PTP WHERE POSTPLACE = "' + ruleName
               + '" ORDER BY PREPOSITION';  (* so that predicates will be in the correct order *)
```

```
  (* note the assumption here that all virtual tables are referred to by only one rule *)

  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  allocate_buffers(sqlda_out);
  open_cursor(sql_return_status, sqlda_in^);
  fetch_row(sql_return_status, sqlda_out^);
  while (sql_return_status = SQL_SUCCESS) do with sqlda_out^ do
  begin
    temp := sqlvar[1].sqldata^.chabuf;
    str$trim(temp, temp, out_len);
    theID := substr(temp, 1, out_len);
    temp  := sqlvar[2].sqldata^.chabuf;
    str$trim(temp, temp, out_len);
    pPlace  := substr(temp, 1, out_len);
    pCols   := sqlvar[3].sqldata^.intbuf;
    thePCNs := Insert_PCN(theID, pPlace, pCols, thePCNs);
    fetch_row(sql_return_status, sqlda_out^);
    if (    (sql_return_status <> SQL_SUCCESS)
        and (sql_return_status <> STREAM_EOF)) then error;
  end;
  close_cursor(sql_return_status);
  release_stmt(sql_return_status);
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
  SearchPCN := thePCNs;
  thePCNs   := nil;
end; (* SEARCHPCN *)


(*****************************************************************************)
(* GETINSCRIPTION (04-DEC-1990)                                              *)
(* Gets the inscription of a transition.                                     *)
(*****************************************************************************)
Function GETINSCRIPTION ( theID : String ) : String;
Var      sqlda_in, sqlda_out : sqlda_ptr value nil;
         the_stmt            : st;
         temp                : Str255;
         out_len             : $uword;
Begin
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  the_stmt  := 'SELECT TRANS_SCRIPT FROM TRANSITION WHERE TRANS_ID = "' + theID + '"';
  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  allocate_buffers(sqlda_out);
  open_cursor(sql_return_status, sqlda_in^);
  fetch_row(sql_return_status, sqlda_out^); (* there should be only one *)
  temp := sqlda_out^.sqlvar[1].sqldata^.chabuf;
  str$trim(temp, temp, out_len);
  GetInscription := substr(temp, 1, out_len);
  close_cursor(sql_return_status);
  release_stmt(sql_return_status);
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end; (* GETINSCRIPTION *)


(*****************************************************************************)
(* FINDARITY (04-DEC-1990)                                                   *)
(* Finds the arity of a table in the database (virtual or base)              *)
(*****************************************************************************)
Function FINDARITY ( theTable : String ) : integer;
Var      sqlda_in, sqlda_out : sqlda_ptr value nil;
         the_stmt            : st;
Begin
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  the_stmt  := 'SELECT COUNT(*) FROM RDB$RELATION_FIELDS WHERE RDB$RELATION_NAME = "' + theTable + '"';
  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  allocate_buffers(sqlda_out);
  open_cursor(sql_return_status, sqlda_in^);
  fetch_row(sql_return_status, sqlda_out^);
  FindArity := sqlda_out^.sqlvar[1].sqldata^.intbuf;  (* there should be only one *)
  close_cursor(sql_return_status);
  release_stmt(sql_return_status);
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end; (* FINDARITY *)


(*****************************************************************************)
(* STORETRANSITION (11-DEC-1990)                                             *)
(* Stores a transition in the TRANSITION relation.                           *)
(*                                                                           *)
(* Called from: RULES.PAS : SAVERULE                                         *)
(*****************************************************************************)
Procedure STORETRANSITION ( theID, theScript : String );
Var       sqlda_in, sqlda_out : sqlda_ptr;
          the_stmt            : st;
Begin
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  the_stmt := 'INSERT INTO TRANSITION VALUES ("' + theID + quoteComma + theScript + '")';
  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  execute_stmt(sql_return_status, sqlda_in^);
  release_stmt(sql_return_status);
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end; (* STORETRANSITION *)
```

```
(*****************************************************************************)
(* STOREPTP (11-DEC-1990)                                                    *)
(* Stores an entry in the PTP relation.                                      *)
(*                                                                           *)
(* Called from: RULE.PAS : SAVERULE                                          *)
(*****************************************************************************)
Procedure STOREPTP ( theID, LHName   : String;
                     preCols, prePos : integer;
                     RHName          : String  );
Var       sqlda_in, sqlda_out : sqlda_ptr;
          the_stmt            : st;
Begin
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  the_stmt := 'INSERT INTO PTP VALUES ("' + theID + quoteComma + LHName + quoteComma
              + NumToString(preCols) + quoteComma + NumToString(prePos) + quoteComma + RHName + '")';
  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  execute_stmt(sql_return_status, sqlda_in^);
  release_stmt(sql_return_status);
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end; (* STOREPTP *)


(*****************************************************************************)
(* RULEREFERENCE (14-DEC-1990)                                               *)
(* Checks whether a given table name refers to a base or a virtual table.    *)
(*****************************************************************************)
Function RULEREFERENCE ( theTable : String ) : Boolean;
Var      sqlda_in, sqlda_out : Sqlda_ptr;
         the_stmt            : st;
Begin
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  the_stmt  := 'SELECT DISTINCT POSTPLACE FROM PTP WHERE POSTPLACE = "' + theTable + '"';
  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  allocate_buffers(sqlda_out);
  open_cursor(sql_return_status, sqlda_in^);
  fetch_row(sql_return_status, sqlda_out^);            (* will return STREAM_EOF if none are found *)
  RuleReference := (sql_return_status = SQL_SUCCESS);
  close_cursor(sql_return_status);
  release_stmt(sql_return_status);
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end; (* RULEREFERENCE *)


(*****************************************************************************)
(* GETTABLECOLUMNS (18-JUN-1990)                                             *)
(* Given a table name, returns a list of its columns.                        *)
(*                                                                           *)
(* Called from: SQL_PROCESS.PAS : EXPANDSTAR                                 *)
(*              SQL_PROCESS.PAS : INSERTTABLECOLUMNS                         *)
(*              SQL_PROCESS.PAS : EXPANDRULE                                 *)
(*                                                                           *)
(* MODIFIED:                                                                 *)
(*  14-DEC-1990 Shifted from SQL_PROCESS.PAS to DYNAMIC.PAS and rewritten to  *)
(*              get info direct from the database                            *)
(*****************************************************************************)
Function GETTABLECOLUMNS ( theTable : string ) : StringPair_ptr;
Var       sqlda_in, sqlda_out : Sqlda_ptr;
          the_stmt            : st;
          theColumns          : StringPair_ptr value nil;
          temp                : Str255;
          len                 : $uword value 0;
Begin
  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;
  the_stmt  := 'SELECT RDB$FIELD_NAME FROM RDB$RELATION_FIELDS WHERE RDB$RELATION_NAME = "' + theTable
               + '" ORDER BY RDB$FIELD_POSITION';
  prepare_stmt(sql_return_status, sqlda_out^, the_stmt);
  describe_parm(sql_return_status, sqlda_in^);
  allocate_buffers(sqlda_out);
  open_cursor(sql_return_status, sqlda_in^);
  fetch_row(sql_return_status, sqlda_out^);
  while (sql_return_status = SQL_SUCCESS) do with sqlda_out^.sqlvar[1] do
  begin
    temp := sqldata^.chabuf;
    str$trim(temp, temp, len);
    theColumns := Append_StringPair(substr(temp, 1, len), '', theColumns);
    fetch_row(sql_return_status, sqlda_out^);
    if (    (sql_return_status <> SQL_SUCCESS)
        and (sql_return_status <> STREAM_EOF)) then error;
  end;
  close_cursor(sql_return_status);
  release_stmt(sql_return_status);
  GetTableColumns := theColumns;
  theColumns      := nil;
  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end; (* GETTABLECOLUMNS *)
```

```
(*****************************************************************************)
(* SEND_TO_SQL                                                               *)
(*                                                                           *)
(* SEND_TO_SQL takes the string which was entered from the keyboard and sends *)
(* it to SQL for processing (see below).                                     *)
(*****************************************************************************)
Procedure SEND_TO_SQL ( sql_stmt : st; theColumns : StringPair_ptr );
Var       sqlda_in, sqlda_out : sqlda_ptr;
Begin

(* Allocate separate SQLDAs for parameter markers (SQLDA_IN) and select list  *)
(* items (SQLDA_OUT).  Assign the value of the constant MAXPARMS (set in the  *)
(* declarations section of this module) to the SQLN field of both SQLDA       *)
(* structures.  SQLN specifies to SQL the maximum size of the SQLDA.          *)

  sqlda_in  := Init_Sqlda;
  sqlda_out := Init_Sqlda;

(* Call an SQL module language procedure, PREPARE_STMT, that contains a       *)
(* PREPARE...SELECT LIST statement to prepare the dynamic statement and write *)
(* information about any select list items in it to SQLDA_OUT:                *)

  prepare_stmt(sql_return_status, sqlda_out^, sql_stmt);
  if (sql_return_status <> SQL_SUCCESS) then error;

(* Call an SQL module language procedure, DESCRIBE_PARM, that contains a      *)
(* DESCRIBE...MARKERS statement to write information about any parameter      *)
(* markers in the dynamic statement to SQLDA_IN:                              *)

  describe_parm(sql_return_status, sqlda_in^);
  if (sql_return_status <> SQL_SUCCESS) then error;

(* Check to see if the prepared dynamic statement contains any parameter      *)
(* markers by looking at the SQLD field of SQLDA_IN.  SQLD contains the       *)
(* number of parameter markers in the prepared statement. If SQLD is          *)
(* positive, the prepared statement contains parameter markers.  The program  *)
(* executes a local procedure, GET_IN_PARMS, that prompts the user for        *)
(* values, allocates storage for those values, and updates the SQLDATA field  *)
(* of SQLDA_IN:                                                               *)

  if (sqlda_in^.sqld) > 0 then get_in_parms(sqlda_in, sqlda_out);

(* Check to see if the prepared dynamic statement is a SELECT by looking at   *)
(* the SQLD field of SQLDA_OUT^. SQLD contains the number of select list      *)
(* items in the prepared statement. If SQLD is positive, the prepared state-  *)
(* ment is a SELECT statement. The program allocates storage for the rows,    *)
(* calls SQL module language procedures to open and fetch from a cursor, and  *)
(* displays the rows on the terminal:                                         *)

  if (sqlda_out^.sqld > 0) then
  begin
    allocate_buffers(sqlda_out);
    open_cursor(sql_return_status, sqlda_in^);
    if (sql_return_status <> SQL_SUCCESS) then error;
    DisplayColumnTitles(theColumns);
    fetch_row(sql_return_status, sqlda_out^);
    if (sql_return_status = STREAM_EOF) then writeln('No records found.')
    else if (sql_return_status <> SQL_SUCCESS) then error;
    while (sql_return_status = SQL_SUCCESS) do
    begin
      display_row(sqlda_out);
      fetch_row(sql_return_status, sqlda_out^);
      if (    (sql_return_status <> SQL_SUCCESS)
          and (sql_return_status <> STREAM_EOF)) then error;
    end;
    close_cursor(sql_return_status);
    if (sql_return_status <> SQL_SUCCESS) then error;
  end
  else
  begin

(* If SQLD in SQLDA_OUT is zero, then the prepared statement is not a SELECT  *)
(* statement and it only needs to be executed.  Call an SQL module language   *)
(* procedure to execute the statement, using the information about parameter  *)
(* markers stored in SQLDA_IN by the local procedure GET_IN_PARMS:            *)

    execute_stmt(sql_return_status, sqlda_in^);
    if (sql_return_status <> SQL_SUCCESS) then error;
  end;  (* else ... *)

(* Once the program is done with the statement, call an SQL module language   *)
(* procedure that contains a RELEASE statement to release resources used by   *)
(* the prepared statement.                                                    *)
(*                                                                           *)
(* This is not strictly necessary.  If the statement is not released          *)
(* explicitly, it will be released the next time the program prepares the     *)
(* same statement name. However, if you know you will not want to use the     *)
(* statement again, it is good practice to release the statement:             *)

  release_stmt(sql_return_status);
  if (sql_return_status <> SQL_SUCCESS) then error;

  Kill_Sqlda(sqlda_in);
  Kill_Sqlda(sqlda_out);
end;    (* SEND_TO_SQL *)


END.    (* DYNAMIC_SQL *)
```

D.8 SQLDYN.SQLMOD

```
-- SQL$DYNAMIC.SQLMOD
--
-- This SQL module provides the SQL procedures needed by the SQL$DYNAMIC.ADA
-- program.
-- The module illustrates how to use SQL module language to process
-- SQL statements generated dynamically by a VAX Ada program.

-----------------------------------------------------------------------------
-- Header Information Section
-----------------------------------------------------------------------------
MODULE          SQL_DYNAMIC     -- Module name
LANGUAGE        PASCAL          -- Language of calling program
AUTHORIZATION   RDB$DBHANDLE    -- Provides default authorization id

-----------------------------------------------------------------------------
-- DECLARE Statements Section
-----------------------------------------------------------------------------

-- Declare a cursor to process dynamic SELECT statements

DECLARE SEL CURSOR FOR DYN_STMT

-----------------------------------------------------------------------------
-- Procedure Section
-----------------------------------------------------------------------------

-- This procedure prepares a statement for dynamic execution from the string
-- passed to it.  It also writes information about the number and data type of
-- any select list items in the statement to an SQLDA (specifically,
-- the sqlda_out SQLDA passed to the procedure by the calling program).
--
-- Note that the PREPARE statement in this procedure could have prepared
-- the statement without writing to an SQLDA.  Instead, a separate
-- DESCRIBE...SELECT LIST statement would have written information
-- about any select list items to an SQLDA.

PROCEDURE PREPARE_STMT
    SQLCODE
    SQLDA
    STMT  CHAR(1024);

    PREPARE DYN_STMT SELECT LIST INTO SQLDA FROM STMT;

-- This procedure writes information to an SQLDA (specifically,
-- the sqlda_in SQLDA passed to the procedure by the calling program)
-- about the number and data type of any parameter markers in the
-- prepared dynamic statement.  Note that SELECT statements may also
-- have parameter markers.

PROCEDURE DESCRIBE_PARM
    SQLCODE
    SQLDA;

    DESCRIBE DYN_STMT MARKERS INTO SQLDA;


-- This procedure dynamically executes a non-SELECT statement.
-- SELECT statements are processed by DECLARE CURSOR, OPEN CURSOR,
-- and FETCH statements.
--
-- The EXECUTE statement specifies an SQLDA (specifically,
-- the sqlda_in SQLDA passed to the procedure by the calling program)
-- as the source of addresses for any parameter markers in the dynamic
-- statement.
--
-- Note that the EXECUTE statement with the USING DESCRIPTOR clause
-- also handles statement strings that contain no parameter markers.
-- If a statement string contains no parameter markers, SQL sets
-- the SQLD field of the SQLDA to zero.


PROCEDURE EXECUTE_STMT
    SQLCODE
    SQLDA;

    EXECUTE DYN_STMT USING DESCRIPTOR SQLDA;


-- This procedure opens the cursor SEL already declared.  It specifies
-- an SQLDA (specifically, the sqlda_in SQLDA passed to the procedure
-- by the calling program) as the source of addresses for any parameter
-- markers in the cursor's SELECT statement.

PROCEDURE OPEN_CURSOR
    SQLCODE
    SQLDA;

    OPEN SEL USING DESCRIPTOR SQLDA;


-- This procedure fetches a row from the opened cursor and writes it to
-- the addresses specified in an SQLDA (specifically, the sqlda_out SQLDA
-- passed to the procedure by the calling program).

PROCEDURE FETCH_ROW
    SQLCODE
    SQLDA;

    FETCH SEL USING DESCRIPTOR SQLDA;
```

```
-- This procedure closes the cursor.

PROCEDURE CLOSE_CURSOR
    SQLCODE;

    CLOSE SEL;


-- The procedure releases the prepared statement.  It frees all resources
-- for the statement.

PROCEDURE RELEASE_STMT
    SQLCODE;

    RELEASE DYN_STMT;


-- This procedure commits the transaction.

PROCEDURE COMMIT_TRANSACTION
    SQLCODE;

    COMMIT;


-- This procedure rolls back the transaction.

PROCEDURE ROLLBACK_TRANSACTION
    SQLCODE;

    ROLLBACK;
```

D.9 SQUALIDMSG.MSG

```
!*****************************************************************************
!*                                                                           *
!* SQUALIDMSG.MSG - N. Stanger, 26-MAR-1990                                  *
!*                                                                           *
!* Message file for SQUALID                                                  *
!* Uses the VAX/VMS Message utility                                          *
!*                                                                           *
!*****************************************************************************


.FACILITY       SQUALID,30/PREFIX=DDB_


.SEVERITY       FATAL
NOTABLE         <rule name not found in SQLD table list>
NOARITY         <predicate with zero arity found>
TOOMANYVARS     <too many variables in rule base>
TABNOTFD        <base table !AD not found in database>
ABORT           <Aaaarrggh! (CRUNCH)>


.SEVERITY       ERROR
NOCOMBINE       <two constants in instantiation: COMBINE_INSTANTS>
BADINSTANT      <two constants in instantiation: ADD_INSTANT>
NORULES         <rules file not found or empty>
UNEXP           <an unexpected error has occurred>
BADSQLD         <syntax error detected in SQLD statement>


.SEVERITY       WARNING
BADINPUTRULE    <entered rule is invalid - reenter>
RULEEXIST       <definition of rule !AD already exists>


.SEVERITY       INFORMATIONAL
BADFILERULE     <invalid rule in rules file - ignored>
NOTMATCHED      <no matching clause found>


.SEVERITY       SUCCESS


.END
```

D.10 DESCRIP.MMS

```
.INCLUDE USER_E:[COSC521.COMMAND.SOURCE]SUFFIXES.MMS

PASCAL  = PASCAL /DEBUG/NOOPT
LINK    = LINK /DEBUG

LISTOPS.OBJ : LISTOPS.PAS, GLOBAL.OBJ

RULES.OBJ : RULES.PAS, GLOBAL.OBJ, LISTOPS.OBJ, DYNAMIC.OBJ

SQL_PROCESS.OBJ : SQL_PROCESS.PAS, GLOBAL.OBJ, LISTOPS.OBJ, -
                  RULES.OBJ, DYNAMIC.OBJ

SQLPARSE.OBJ : SQLPARSE.PAS, GLOBAL.OBJ, SQL_PROCESS.OBJ, DYNAMIC.OBJ

DYNAMIC.OBJ : DYNAMIC.PAS, GLOBAL.OBJ

SQUALID.OBJ : GLOBAL.OBJ, DYNAMIC.OBJ, SQLPARSE.OBJ, RULES.OBJ

SQUALID : SQUALID.OBJ, GLOBAL.OBJ, DYNAMIC.OBJ, SQLDYN.OBJ, SQLPARSE.OBJ, -
          RULES.OBJ, LISTOPS.OBJ, SQUALIDMSG.OBJ, SQL_PROCESS.OBJ
          $(LINK) $(LINKFLAGS) SQUALID, GLOBAL, DYNAMIC, SQLDYN, -
                               SQLPARSE, RULES, LISTOPS, SQUALIDMSG, SQL_PROCESS
```