695

A RELATIONAL DATABASE SYSTEM IN PROLOG

DISSERTATION SUBMITTED TO THE JAWAHARLAL NEHRU UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE AWARD OF THE DEGREE OF
MASTER OF TECHNOLOGY
(COMPUTER SCIENCE)

KUO PAHAI

SCHOOL OF COMPUTERS & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067

1987

To my parents

CERTIFICATE

This work, embodied in the dissertation titled, "A RELATIONAL DATABASE SYSTEM IN PROLOG", has been carried out by Mr.Kuo Pahai, a bonafide student of School of Computers and Systems Sciences, Jawaharlal Nehru University, New Delhi - 110 067.

This work is original and has not been submitted for any other degree or diploma in any other university or institute.

Mr. C.Ravi Shankar,
Systems Specialist,
R & D Centre,
CMC Ltd.,
115, S.D. Road,
Secunderabad - 500 003.

Dr. K.K.Nambiar,
Professor,
School of Computers &
Systems Sciences,
Jawaharlal Nehru University,
New Delhi - 110 067.

Professor Karmeshu,
Dean, School of Computers & Systems Sciences,
Jawaharlal Nehru University,
New Delhi - 110 067.

ACKNOWLEDGEMENTS

This project would not have materialized without the kind consent of my guide Prof. K.K.Nambiar. I am grateful for the help and advice given by him during the tenure of my project, and many more things.

My sincere thanks are due to Mr. C.Ravi Shankar, my guide at CMC Secunderabad, whose foresight and understanding of the time-bound nature of the project helped me choose an appropriate topic of work. I am also grateful to him for the help and advice offered whenever I needed it.

I am thankful to Mr. S.Kapoor, Systems Manager, for giving me this opportunity to work at the CMC, R&D Centre, in Secunderabad.

Many thanks are due to some members of PACE (Parallel Architecture Computation Environment), notably, U.Bhaskar and Siva Prasad for clearing many of my nagging doubts about Prolog.

I am grateful to Mr. Narasimha Rao, who patiently sat through a query session on the supplier-part-department database, and offered many helpful suggestions on what the shortcomings were and how they could be removed. Many of his suggestions, have given insight in the area of integrity constraints, and also other features offered on various DBMS's.

My thanks also go to the 'Operations' staff who bore my long hours at the terminal pleasantly and for booting VAX promptly, despite vagaries in power supply, particularly during the end of my project.

Sincere thanks go to all friends sitting near my cluster in Parklane, for having accomodated me and also for many stimulating and informative discussions on computers and otherwise.

The list of all persons whom I should acknowledge is very long. Any oversight on my part in this aspect, I hope, is forgiven.

JAN 1988

CONTENTS

PROLOGUE

The following chapters describe in detail the creation of a relational database system in Prolog and some aspects of the formalism of logic and relational algebra.

The idea of implementing a relational database system in Prolog is as old as the language itself, and is in fact an obvious consequence of the "database" nature of Prolog. I shall not offer a detailed argument to support this view - suffice to say that, Prolog is an inherent tuple searching (database searching) language and backtracking ensures exhaustive database search till all possible solutions to a given query are found.

Prolog is a recursive language and coding can be very compact. More importantly it is a highly descriptive language in that - specification regarding "what to do" and not "how to do" is enough. In other words one need not give specific control statements for program execution. Consequently, it not only cuts down program development time but also makes a program more understandable and amenable to changes. The Prolog program itself executes like a database search where each sub-goal to be satisfied is searched and on matching with the appropriate head of the clause enters it and executes its body, which may be further sub-goals to be satisfied. In fact one might say here that the distinction between data and program is lost.

The data manipulation language is very similar in syntax to the query language SQL. The database is created and stored as Prolog facts. The relational schema are stored as Prolog structures.

Chapters 1 and 2 discuss some preliminaries of logic, predicate calculus, relational algebra and some important definitions involved in this algebra.

Chapters 3 and 4 form the core of the project, where the query processor is discussed with the help of a complicated query. The functions of the top level predicates involved in this processor are explained.

Chapter 5 deals with some aspects of representation of negative information in the database, some strategies involved in the construction of aggregation operators and other useful predicates.

Chapter 6 discusses the implementation, its capabilities, its performance compared to other RDBMS systems its shortcomings and how one could remove them, and other pointers for future work.

The appendices at the end contain a sample database, sample queries and their solutions, the syntax of the query language, internal representation of relational schema and integrity constraints and the source listing of the program.

CHAPTER 1

INTRODUCTION TO LOGIC, RESOLUTION AND PROLOG

A brief introduction to logic, resolution and Prolog is given here.

1.1 PREDICATE LOGIC

Logic studies the relationship between assumptions and conclusions. It was originally devised as a way of representing arguments formally, so that it would be possible to check in a mechanical way whether or not they were valid. Thus we can use logic to express propositions, the relation between propositions and how one can validly infer some propositions from others (theorem proving). From the logic point of view, a generalised DBMS is just a general-purpose question-answering system in which the set of facts necessary for question-answering (or problem solving) can be viewed as axioms of a theorem, and the question (or the problem) can be viewed as the conclusion of the theorem. Therefore the emphasis of such a database management system is on its deductive power, which corroborates the assertion that logic plays a significant role in DBMSs.

The predicate calculus is a formal language whose essential purpose is to symbolize logical arguments in mathematics. The sentences in this language are called well-formed formulas (wffs). By "interpreting" the symbols in a wff we obtain a statement which is either true or false. We can associate many different interpretations with the same wff and therefore obtain a class of statements where each statement is either true or false. However our interest is mainly in a very restricted subclass of the wffs, those that yield a true value for every possible interpretation.

1.2 BASIC NOTATIONS

The symbols from which our statements are constructed are listed below.

1.2.1 Quantifiers:

The two quantifiers used are: $\forall$ (universal quantifier) and $\exists$ (existential quantifier).

1.2.2 A term is recursively defined as:

1. A constant is a term.

2. A variable is a term.

3. If f is an n-ary function symbol, and t1,t2,...,tn are terms, then f(t1,...,tn) is a term.

4. All terms are generated by applying the above rules.

1.2.3 Atomic formulas are defined as:

If P is an n-ary predicate symbol, and t1,t2,...,tn are terms, then P(t1,...,tn) is an atomic formula. No other expression can be an atomic formula.

1.2.4 Well formed formulas (WFFs) are recursively defined as:

1. An atomic formula is a WFF.

2. If F and G are WFFs then:
~F, F & G, F # G, F -> G and F <-> G are WFFs, where ~, &, #, -> and <-> are 'negation', 'conjunction', 'disjunction', 'implication' and 'equivalence' logical connectives respectively.

3. If F is a WFF, x is a free variable, then (∀x)F and (∃x)F are WFFs, where ∀x and ∃x are universal and existential quantifiers (quantity connectives) respectively.

4. WFFs are generated only by a finite number of applications of (1), (2) and (3).

In a WFF a connective can be expressed in terms of the other connectives. For instance :

$$F \leftrightarrow G \quad == \quad (F \rightarrow G) \ \& \ (G \rightarrow F)$$

$$F \rightarrow G \quad == \quad {\sim}F \ \# \ G$$

$${\sim}{\sim}P \quad == \quad P$$

$${\sim}(F \ \& \ G) \quad == \quad {\sim}F \ \# \ {\sim}G$$

$${\sim}(F \ \# \ G) \quad == \quad {\sim}F \ \& \ {\sim}G$$

$$(\exists x)P \quad == \quad {\sim}((\forall x)({\sim}P))$$

$$(\forall x)P \quad == \quad {\sim}((\exists x)({\sim}P))$$

A functionally complete set of connectives in Predicate Calculus, which is sufficient to express any formula in an equivalent form, could be:

$$\{\&, {\sim}, \forall x\} \text{ or } \{F, \rightarrow, \forall x\}.$$

As a result of the redundancy, there are many ways to express the same formula in an equivalent form (a similar situation to that in database query languages; different queries may express the same request). If we wish to carry out formal manipulations on Predicate Calculus formulas, this turns out to be very inconvenient. It would be much nicer if everything we wanted to say could only be expressed in one way.

A special form called the Clausal Form alleviates this problem to a certain extent.

1.2.5 The Clausal Form -

A clause is an expression of the form :

B1,B2,...,Bm <- A1,A2,...,An

where B1,B2,...,Bm,A1,A2,...,An are atomic formulas, n >= 0 and m >= 0. The atomic formulas A1,A2,...,An are joint conditions of the clause; and B1,B2,...,Bm are the alternative conclusions. If the clause contains the variables x1,x2,...,xk, B1 or B2 or ... or Bm holds if A1 and A2 and ... and An hold. Alternatively a clause may be defined as a finite disjunction of literals, a literal being an atomic formula or negation of an atomic formula. A wff is said to be in a clausal form if it is a finite conjunction of clauses.

There are three special cases of clausal forms:

1. If n=0, that is,

   B1,B2,...,Bm <-

   then: for all x1,x2,...,xk, B1 or B2 or ... or Bm is unconditionally true.

2. If m=0, that is,

   <- A1,A2,...,An

   then: for all x1,x2,...,xk, it is not the case that A1 and A2 and ... and An are true.

3. If m=n=0, that is the empty clause:

   <-

   then: this is a formula which is always false (a contradiction).

1.2.6 Horn Clauses

Clauses containing at most one conclusion are called Horn Clauses, (first investigated by logician Alfred Horn). For many applications of logic, it is sufficient to restrict the form of clauses to Horn Clauses. There are two types of Horn Clauses:

1. Headed Horn Clauses.

   B <- A1,A2,...,An

   or

   B <-

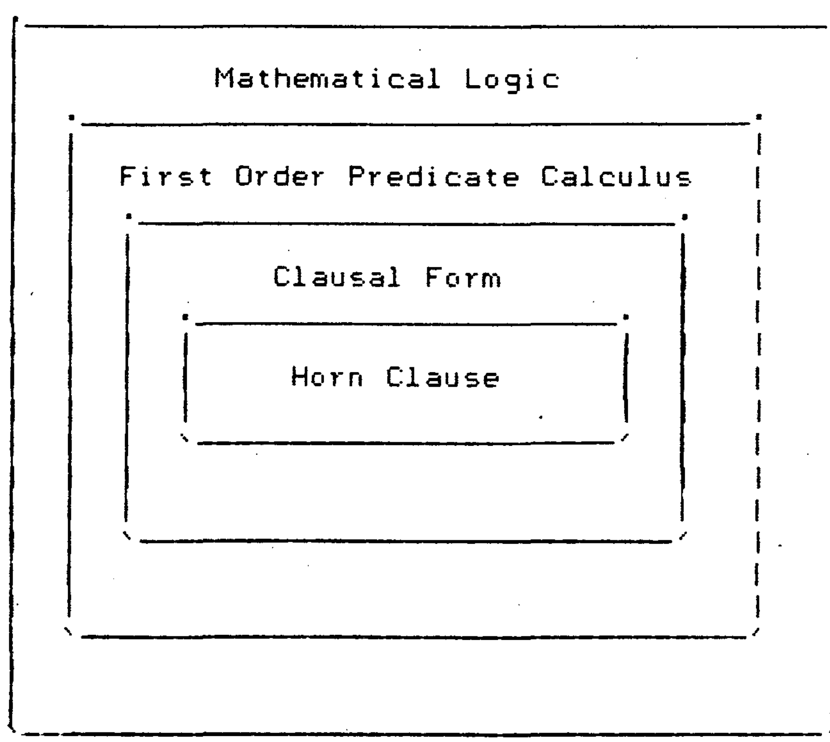
2. Headless Horn Clauses.

<- A1,A2,...,An

or an empty clause:

<-

Any solvable Horn Clauses can be expressed in such a way that, there is one headless clause, and all the rest of the clauses are headed.



Figure(1.1). The onion layer of logic programming.

Thus, we can decide to view the headless clause as the goal which must be present for a problem to be solvable and the other clauses as the hypotheses (axioms). Because resolution with Horn clauses is relatively simple, they are an obvious choice as the basis of a theorem prover which provides a practical programming system.

1.3 RESOLUTION

Proving the satisfiability of wffs in predicate calculus can be done easily by a method known as resolution. Resolution techniques can be applied only to wffs which are in clausal form and all wffs can be reduced to the clausal form. Prolog also incorporates the idea of resolution for proving a theorem.

1. (Robinson) A given wff S in clause form is unsatisfiable if and only if the empty clause can be derived eventually by repeated application of the resolution rule.

2. Resolution: To prove the satisfiability of a wff S in clausal form, it is equivalent to prove that its negation ~S is unsatisfiable. Therefore we repeatedly apply resolution rule to ~S, until an empty clause is obtained. Resolution method says that if we have a clause A and its negation ~A, then "resolvent" of A and ~A yields the empty clause. In Prolog such clauses may be interpreted

as the sub-goals of a main goal, thereby reducing the problem of proving the main goal to that of resolving the sub-goals. If all the sub-goals are resolvable the the main goal is true.

1.4 PROLOG AS A LOGIC PROGRAMMING LANGUAGE

The language Prolog based on Horn Clauses can be seen as the first step towards the ultimate goal of programming in logic [1]. In fact Prolog can be summed up as:

PROLOG = Horn CLauses + Extralogical facilities

Prolog, based on Horn clauses has to provide extralogical facilities in order to solve many problems. In fact the presence of these extralogical facilities may mean the difference between solving the problem or not solving it.

Despite these "shortcomings" Prolog is a language with many outstanding features. It is a highly descriptive language - where the logic of the program is coded, and a programmer is not bogged down by the syntax and creation of data structures as in other more conventional procedural languages. Also Prolog is the most widely available logic programming language.

The version of Prolog I have used is C-Prolog which incorporates all features of the de facto standard - Edinburgh Prolog apart from which it also provides many other facilities in the form of built-in predicates.

A few of the bare essentials of Prolog are given here. The objects that are present in Prolog are:

1.4.1 Terms

The data objects of the language are called terms. A term may be a constant, a variable or a structure.

1. Constants: A constant is thought of naming a specific object or a specific relationship. The constants include integers such as: 0, 1.3e3 and atoms which normally begin with a lower case such as: apple, john or special symbols such as ?-, :-, -> etc. To improve readability the underscore character "_" may be used such as book_name

2. Variables: Variables are like atoms except that they begin with an uppercase letter or with an underscore e.g. London, _val. When we do not need to know what a variable is instantiated to while running a Prolog program the underscore "_" is used and is referred to as the anonymous variable. A variable may thus be thought

of as standing for some definite but unspecified object.

3. Structures : A structure comprises of a functor and a sequence of one or more terms called arguments. A functor is characterized by its name, which is an atom, and its arity being the number of arguments. e.g. book(Title,Author_name,Language) may be thought of as a structure in Prolog where book is the functor of this structure with three arguments called Title,Author_name and Language.

1.4.2 Headed Clauses

There are two kinds of headed clauses - facts and rules.

1. Facts: A fact is an assertion which is an axiom, such as; likes(john,mary) stating that john likes mary. When defining relationships about objects the order may be arbitrary but we must be consistent about its interpretation and order. For instance, likes(john,mary) is not the same as likes(mary,john).

2. Rules: A rule is a conditional assertion which consists of a head and a body. The head and the body are connected by the symbol ":-" which may be read as "if" e.g. Head :- Goal1, Goal2, ... Goaln. For instance john likes anyone who likes wine in Prolog would become:

```
likes(john,X) :- likes(X,wine).
```

The head of this rule describes what fact the rule is intended to define. The body describes the conjunction of goals that must be satisfied, one after the other, for the head to be true. Suppose john likes any female who likes wine, we have:

```
likes(john,X) :- female(X), likes(X,wine).
```

Here unless the sub-goals female(X) and likes(X,wine) are true the head likes(john,X) will not succeed. The database search is done left to right depth-first i.e. not until sub-goal female(X) is satisfied, will likes(X,wine) be attempted.

1.4.3 Headless Clauses

Headless clauses in Prolog are questions. A special symbol "?-" is always put before the question by the system. When a question is asked of Prolog, it will search through the database, and look for facts and rules that match the question. Two facts match if their functors are the same and if each of their corresponding arguments are the same. Such a match is called unification of two terms. Unification algorithm is discussed well in [2]. In logic this method of matching may be interpreted as finding a proof of the question in the database by resolution, where the symbol "?-"

stands for negation of the question, and hence if negation of the question is resolvable then the question is true- the basis for proof by resolution.

For a more detailed account of the syntax and semantics of Prolog I suggest the reader to peruse Programming in Prolog - W.F.Clocksin and C.S.Mellish [3]. Chapters one through three of Mathematical Theory of Computation by Zohar Manna give an excellent introduction to predicate calculus and resolution [4].

CHAPTER 2

RELATIONAL ALGEBRA - DEFINITIONS, THEORY

This chapter deals with relational algebra, some important definitions and concepts.

One of the most critical problems facing database installations today is the rapidly increasing cost of developing and maintaining programs.Two fundamental problems in the use of files are:

1. The data dependence problem: In nondatabase systems an application is data-dependent. It is impossible to change the storage (how the data is physically recorded) or access strategy (how it is accessed) without affecting the application.

2. The data redundancy problem: The fact that each application has its own private files can often lead to considerable redundancy in stored data, with resultant waste in storage space and a high risk of inconsistency.

The relational approach grounded in a well-established mathematical discipline, is a significant approach to the logic description and manipulation of data. Briefly speaking it views the logical database as a time-varying collection of normalized relations which are flat in that no repeating groups are involved.

A relational database is manipulated by powerful operators for extracting columns and joining them, and involves no consideration of positional, pointer or access path aspects as in the case of the network and hierarchical approaches.

2.1 TERMINOLOGY OF RELATIONAL MODEL.

The relational approach carries a terminology of its own and a tendency to use terms which are rather mathematically oriented as it has a good theoretical foundation. A few important terms are defined below for convenience:

1. RELATION : Given a collection of sets D1,D2,...,Dn, R is a relation on those n sets if it is a set of ordered n-tuples <d1,d2,...,dn> such that d1 belongs to D1, d2 belongs to D2,...,dn belongs to Dn. It is normal to display relations as tables where the attributes head the columns and the rows are tuples. Hence the name table and relation are used interchangeably.

R			
AA	BB	CC	DD
a1	b1	c1	d1
a1	b2	c2	d2
a1	b1	c1	d2
a1	b2	c2	d1
a2	b3	c1	d1
a2	b3	c1	d2

Figure(2.1). A Relation (or table) 'R' .

	TERM	CORRESPONDING NAMES	EXAMPLES
1.	relation	table/file	R
2.	relational scheme	table heading	<AA,BB,CC,DD>
3.	attribute	column name/field type	AA
4.	attribute values	underlying domain	a1,a2
5.	component	value	c2
6.	arity	degree	4
7.	tuple	row/entity/segment/record	<a1,b2,c2,d2>
8.	cardinality		6
9.	FD		BB --> CC
10.	MVD		AA ->-> <BB,CC>

Figure(2.2). Terminology.

2. RELATIONAL SCHEME: This is a set of attributes, which are names of columns for a relation. A relational scheme is assumed to remain constant over time, while the relation corresponding to it changes frequently.

3. ARITY: The arity of a relation is the number of columns in the relation. Relations of arity two are binary, ..., and relations of arity n are n-ary.

4. ATTRIBUTE: An attribute of a relation is a column name in the relation; whereas the attribute values are the contents of the column.

5. DOMAIN: A domain is the set of values from which the set of attribute values of a relation may be taken; that is, from which a column of a table may be formed. It is important to appreciate the difference between a domain and columns; a column represents the use of a domain within a relation.

6. TUPLE: We may consider a relation as being a mathematical set of tuples, a tuple is a row in the table. Each tuple in a row is unique since it is an element in a mathematical set.

7. CARDINALITY: The cardinality of a relation R, denoted by |R|, is the number of tuples in R.

8. INTEGRITY CONSTRAINTS: Database consistency is enforced by integrity constraints which are assertions that database instances are compelled to obey. Data dependencies are special cases of integrity constraints.

9. FUNCTIONAL DEPENDENCY (FD): A functional dependency X-->Y is an assertion about a relation scheme. It asserts of any "legal" relation that two of its tuples t1 and t2 that agree on a set of attributes X, also agree on Y, that is, if t1[X] = t2[X] then t1[Y] = t2[Y].

10. MULTIVALUED DEPENDENCY (MVD): A multivalued dependency X->->Y is a statement that if t1 and t2 are two tuples of a relation that satisfies X->->Y and X-->Y, then the relation must have a third tuple t3 that takes the Y-value from one of the tuples, say t1, and its value everywhere from the other. In other words, X->->Y means that the set of Y-values associated with a particular X-value must be independent of the values of the rest of the attributes.

11. KEYS: In terms of FD and MVD of a relation, a minimum collection of attributes that can function as a unique identifier is called a candidate key. A relation may have more than one candidate key.

12. NORMALIZATION: The theory of normalization relates dependency types to desirable properties of relation schemes. By normalization there can be no redundancy caused by dependencies of the given type in the relations for this scheme.

13. NORMALIZED RELATION: A normalized relation is a table with non-decomposable values, which sometimes is called a base table, or a base relation.

14. DATABASES: A database is a collection of relations, and their relation schemes collectively form the database scheme.

To sum up a relational database DB is a collection R = {Ri}, describing certain objects of the world having certain attributes D, and the relationships among D, i.e. a set of dependencies F. Each relation Ri is characterised by a set of attributes Si = {Dj|Dj belongs-to D} called its scheme, and consists of a set of tuples. Each tuple is a map from the attributes of the relation scheme to their domains that satisfy all the dependencies of F.

A relational database is accessed by a set of requests submitted by the user in order to retrieve, delete, insert or modify any subset of the data. In general, retrieval is an essential part of these processes. Users submit their requests for information in the form of queries, specifying the data which must be retrieved and the conditions which must be satisfied by the desired data. The task of query processing is to determine the set of data, the proper order in which the data should be accessed and the types of manipulations that must be performed on the data. This processing is referred to by different authors as query translation, access path finding, or optimisation.

2.2 THE RELATIONSHIP BETWEEN LOGIC AND DATABASES

Logic has been found to be highly useful both for describing static databases as well as for processing databases which change. On the other hand, the evolution in database technology, particularly in relational model, has been drifting more and more toward the use of logic. A considerable amount of programming logic theory can be transferred to databases. The use of logic for data description abolishes the distinction between databases and programs. Strategies which apply to the execution of programs apply also to the retrieval of answers to database queries. Methods for proving properties of programs apply to verification of integrity constraints. Procedures for maintaining dynamically evolving databases apply to the evolutionary development.

As far as relational query languages and user-interfaces are concerned, there are many problems that can be conveniently transformed into logic, some of which are:

1. From the viewpoint of theorem proving, a database system can be considered as a question-answering system where facts are represented as logic formulas, and answering a question from facts may be regarded as proving that a formula corresponding to the answer is derivable from the formulas representing the facts; and therefore deductive search becomes important [5].

2. If a query is written in a very high level non-procedural query language the execution of such a query is to perform tasks that would be called intelligent. The database system can be viewed as a knowledge-base system. Starting with an initial state, one tries to find a sequence of operators that will transform the initial state into the desired state which is called knowledge information processing. In this case, we can describe the states and the state transition rules by logic formulas. In other words, logic programming languages, such as Prolog, can be applied to the task of writing compilers for high level query languages [6,7]. The major advantage of Prolog as a language for writing compilers is that it specifies algorithms in a human-oriented way.

Such specification can be interpreted as a uniform resolution theorem prover. It is not a great exaggeration to say that specification is the implementation. In fact, logic programming can extend any query language to a programming language.

3. Since a database may be viewed as a logic program, retrieval is automatically taken care of through resolution. For instance, given a predicate definition for an n-ary predicate p, the retrieval of the jth argument corresponding to specific values of the others is simply obtained by posing the query:

   | ?- p(v1,...,vj-1,X,vj+1,...,vn).

   where v1 are those specific values and X is a variable. Because of nondeterminism, more than one value for X may be retrieved. Because of the nondeterminism between input and output, any argument or combination of arguments can be chosen for retrieval. Thus retrieval operations that are usually needed in relational databases. For the above example, projection and selection become so simple with logic programming.

4. The link with logic was not only found at the language level and query evaluation. For instance, an equivalence between dependencies (FDs and MVDs) and a fragment of propositional logic has been found [8,9].

r1

aa	bb	cc
a	b	c
d	a	f
c	b	d

s1

aa	bb	cc
b	g	a
d	a	f

r2

aa	bb	cc	dd
a	b	c	d
a	b	e	f
b	c	e	f
e	d	c	d
e	d	e	f
a	b	d	e

s2

cc	dd
c	d
e	f

r3

aa	bb	cc
1	2	3
4	5	6
7	8	9

s3

dd	ee
3	1
6	2

r4

aa	bb	cc
a	b	c
b	b	f
c	a	d

s4

bb	cc	dd
b	c	d
b	c	e
a	d	b

Figure(2.3). MATHEMATICAL DATABASE

A wider use of logic should have a positive effect on the database field, as it provides not only a conceptual framework for formulating various database concepts, but also (in the form of logic programming) a tool for implementing them.

2.3 OPERATIONS OF RELATIONAL ALGEBRA:

Some of the basic operations of relational algebra which are provided by standard DBMS's are given below with illustrative examples.

To give an example of these operations the mathematical database given in Figure(2.3). is used. There are five basic operations that serve to define relational algebra. These are: union, set difference, extended cartesian product, projection and selection.

1. Union. The union of relations R and S, denoted R++S, is the set of tuples that are in R or S or both. R and S must be union compatible, that is to say that they have the same arity. For example r1++s1 is:

r1 ++ s1

aa	bb	cc
a	b	c
d	a	f
c	b	d
b	g	a

2. Set difference. The difference of relations R and S, denoted R--S, is the set of tuples in R but not in S. R and S are required to be union-compatible. For example r1 -- s1 is:

r1 -- s1

aa	bb	cc
a	b	c
c	b	d

3. Cartesian product. Let R and S be relations of arity m and n, respectively. Then R**S, the cartesian product of R and S, is the set of (m+n) tuples t's such that t is the concatenation of a tuple r belonging to R and a tuple S belonging to S. The concatenation of a tuple r = [r1,r2,...,rm] and a tuple s = [s1,s2,...,sn], in that order, is a tuple t = [r1,r2,...,rm,s1,s2,...,sn]. For example r1**s1 is:

r1 ** s1

11	22	33	44	55	66
a	b	c	b	g	a
a	b	c	d	a	f
d	a	f	b	g	a
d	a	f	d	a	f
c	b	d	b	g	a
c	b	d	d	a	f

4. Projection. The projection operator transforms one relation into a new one consisting of selected attributes of the first, including duplicate elimination. Project relation r1 on attributes cc and aa, denoted r1[cc,aa],

is formed by taking each tuple in r1 and forming a new tuple from the third and first components of r1 in that order. For example r1[cc,aa] is:

r1 [cc,aa]

cc	aa
c	a
f	d
d	c

5. Selection. Let F be a formula involving:

   1. operands that are constants or attributes,

   2. the arithmetic comparison: =,\=,<,>,=<,>=,

   3. the arithmetic operators: +,-,*,/,

   then R:F is the set of tuples t in R such that when all attributes occurring in F are substituted by the corresponding components of t, the formula F becomes true. For example, to select tuples from r3 where bb>2, denoted r3: bb>2, we obtain:

r3 : bb>2

aa	bb	cc
4	5	6
7	8	9

In addition, there are a number of useful algebraic operations that can be expressed in terms of the previously mentioned operations, but which have been given names in the literature and sometimes used as primitive algebraic operations. These are intersection, quotient, join, and natural join.

6. Intersection. The intersection of two (union-compatible) relations R and S, denoted R::S, is the set of all tuples t belonging to both R and S. We have:

R::S == R -- (R -- S)

For example the intersection of r1 and s1, r1::s1 is:

r1 :: s1

aa	bb	cc
d	a	f

7. Quotient. Let R and S be relations of arity m and n respectively where m>n. Then R %% S is the set of (m-n) tuples t such that for all s-tuples u in S, the tuple tu is in R. For example r2%%s2 is:

r2 %% s2

aa	bb
a	b
e	d
b	c

8. Join. The O-join of R and S on columns r1,r2,...,ri,s1,s2,...,sj written as R**S:F where F is a formula involved with these columns and defined as the same as that in selection operation, is a set of tuples t in the cartesian product of R**S such that when all attributes occurring in F are substituted by corresponding components of t the formula F becomes true. For example r3**s3 : dd>bb [aa,bb,cc,dd,ee] is:

r3 ** s3 : dd > bb [aa,bb,cc,dd,ee]

a	b	c	d	e
1	2	3	3	1
1	2	3	6	2
4	5	6	6	2

9. Natural join. If O is "=" in an O-join operation, it is called an equi-join denoted by R <=> S. For example R <=> S is:

r4 <=> s4:

aa	bb	cc	dd
a	b	c	d
a	b	c	e
c	a	d	b

Many other functions like updating and deleting tuples, the ability to group tuples using the key of the relation, finding aggregation of certain attributes, like CNT, SUM, AVG, MAX, MIN are also possible. The existential quantifier in clausal form is implicitly assumed. A lot of structural compatibility between the logic form and existing high level query languages is achieved. Thus logic representation appears to be enough for implementation of many realistic and useful query languages.

Additional information on relational algebra and other RDBMS's are found in [10,11].

CHAPTER 3

SYNTAX, INTERNAL REPRESENTATION AND PARSING THE QUERY

This chapter deals with the syntax of the query language, some of its important features and the first two steps involved in the processing of the query.

The language which resembles SQL is simple in its structure, so that users without prior experience are able to learn a usable subset on their first sitting. At the same time when taken as a whole, the language provides the query power of the first-order predicate calculus combined with operators for grouping, arithmetic, and other aggregation functions such as AVG, MAX, MIN. One of the good qualities of this language, as expressed by the users was the uniformity of its syntax across environments of application programs, ad hoc queries and definition of views. Languages like SQL provide a common core of features that will be required by all high level query languages.

For these reasons, therefore, it is necessary that an understanding of the features provided by the query language be looked at. The query language chosen for this implementation is very similar to SQL (Structured Query Language), developed by IBM, on System R. For all practical purposes I shall call the language SQL itself.

The following section illustrates some of the important features of SQL, based on the supplier-parts-department database given in the appendix.

3.1 THE LANGUAGE SQL

The fundamental operation for data manipulation in SQL is the SELECT-FROM-WHERE block. The SELECT clause specifies the attributes of the target result; the FROM clause specifies the names of relations which are involved in getting the result both in conditioning selection and in actually supplying result value; the WHERE clause specifies the condition(s) or the constraint(s) based on which the results are to be obtained. For example the user might like to see a display of supplier names and their status for suppliers in Paris. The query Q(3.1) which does this is:

```
select [sname,status]
from   [supplier]
where  [city=paris].
```

Q(3.2) Get full details of all employees.

```
select *
from   [employee].
```

The asterisk is a shorthand for an ordered list of all attribute names in the FROM table. At the moment this implementation allows only in "SELECT-FROM" queries. For instance the above query could be equivalently written as:

```
select [ename,age,salary,dno]
from   [employee].
```

Qualified retrieval. Q(3.3). Get supplier numbers for suppliers in Paris with status >20.

```
select [sno]
from   [supplier]
where  [status>20, city=paris].
```

The conditions following WHERE may include comparison operators like =, \=, >, >=, < and =<. The comma specified in the WHERE list is implicitly assumed to be the AND operator.

Join. Joining in SQL is described by placing attributes from more than one relation in the SELECT clause, placing multiple relation names in the FROM clause, and including a join criterion in the WHERE clause. For e.g. Q(3.4). For each part supplied, get the part number and names of all cities supplying the part.

```
select [pno,city]
from   [shipment,supplier]
where  [shipment@sno = supplier@sno].
```

At present this implementation does not allow using the relation names in the select clause which is used to resolve any ambiguities in the query.

To get the less join of some relation we have Q(3.5). List all suppliers names and locations for suppliers whose status is less than Smith's.

```
select [sname,city]
from   [supplier]
where  [[status] <
                    select [status]
                    from   [supplier]
                    where  [sname = smith]].
```

Retrieval using equi-join. Q(3.6) Get supplier names for suppliers who supply part p2:

```
select [sname]
from   [supplier,shipment]
where  [supplier@sno = shipment@pno,
        shipment@pno = p2].
```

Since the result in the above query is entirely extracted from supplier, though use of two relations is made to determine the result, it should therefore be possible to express the query in the form:

```
select [sname]
from   [supplier]
where  [[sno] =
                  select [sno]
                  from   [shipment]
                  where  [pno = p2]].
```

The expression in the brackets is a sub-query. In general the condition

```
f = [SELECT Sth FROM ...]
```

evaluates to true if and only if the value of f is equal to at least one value in the result of evaluating the "SELECT Sth FROM ...". Similarly the condition

```
f < [SELECT Sth FROM ...]
```

evaluates to true if and only if the value of f is less than at least one value in the result of evaluating the "SELECT Sth FROM ...". The operators >, >=, \= and =< are analogously defined.

Of the various comparison conditions the most useful is:

f = [SELECT Sth FROM ...]

which may be equivalently (and more clearly) written as

f in [SELECT Sth FROM ...]

Retrievals with multiple levels of nesting. Q(3.7) Get supplier names for suppliers who supply at least one red part.

```
select [sname]
from   [supplier]
where  [[sno] in
                select [sno]
                from   [shipment]
                where  [[pno] in
                                select [pno]
                                from   [part]
                                where  [color = red]]].
```

Sub-queries can be nested to any depth. It is also possible to retrieve with a sub-query where the same relation name is involved in both blocks. Refer example Q(3.5).

Retrieval with negation. Q(3.8) Get supplier names for suppliers who do not supply part p2.

```
select [sname]
from   [supplier]
where  [[sno] not_in
                     select [sno]
                     from   [shipment]
                     where  [pno = p2]].
```

The condition in the where clause:

f not_in [select Sth from ...]

evaluates to true if and only if for all values V's in the result of evaluating "SELECT Sth FROM ..." none of them is f.

SQL uses the usual set operators - UNION, INTERSECT, and MINUS - to combine the results of independent SELECT-blocks. These operators are algebraic in nature, and are procedural components. In order to reduce the procedurality, SQL dispenses with INTERSECT and MINUS by using IN and NOT_IN combined with a set of attributes instead.

For example for the mathematical database given. The intersection of relations r1 and s1 can be queried as Q(3.9):

```
select [aa,bb,cc]
from   [r1]
where  [[aa,bb,cc] in
                        select [aa,bb,cc]
                        from   [s1]].
```

To get the difference of relations r1 and s1, we ask Q(3.10)

```
select [aa,bb,cc]
from   [r1]
where  [[aa,bb,cc] not_in
                            select [aa,bb,cc]
                            from   [s1]].
```

Retrieval using union: Q(3.11). Get part number for parts that either weigh more than 18 units, or are currently supplied by Jones or both.

```
        select [pno]
        from   [part]
        where  [wt > 18]
    union
        select [pno]
        from   [shipment]
        where  [[sno] in
```

```
select [sno]
from   [supplier]
where  [sname = jones]].
```

3.2 USAGE OF AGGREGATION OPERATORS (BUILT-IN FUNCTIONS)

The retrieval power of the basic language can be easily extended by provision of built-in functions. The functions currently supported by SQL are: CNT, SUM, MAX, AVG, MIN and grouped_by.

Function in the SELECT clause. Q(3.12) Get the number of shipments for part p2.

```
select [cnt(sno)]
from   [shipment]
where  [pno = p2].
```

Function in a sub-query. Q(3.13). Get employee names for employees who earn more than the average of all employees in the department d1.

```
select [ename]
from   [employee]
where  [[salary] >
                    select [avg(salary)]
                    from   [employee]
                    where  [dno = d1]].
```

Use of grouped_by. Q(3.14) For each part supplied, get the part number and the total quantity supplied of that part.

```
select     [pno,sum(qty)]
from       [shipment]
grouped_by [pno].
```

The GROUPED_BY operator conceptually rearranges the FROM relation into partitions or groups, such that within any one group all tuples have the same value for the GROUPED_BY attribute. In the above example relation "shipment" is grouped such that the first group contains the tuples for part p1, the second contains the tuples for part p2 and so on. The SELECT clause is then applied to each group of the partitioned relation. Each expression in the SELECT clause must be single-valued for each group; that is it can be the GROUPED_BY attribute itself, or a function such as SUM that operates on all values of a given attribute within a group and reduces those values to a single value. Q(3.15) For each department, get the total number of employees with their average age and their average salary.

```
select      [dno,cnt(ename),avg(age),avg(salary)]
from        [employee]
grouped_by  [dno].
```

Use of HAVING and GROUPED_BY. Q(3.16) Get part number for all parts supplied by more than two suppliers.

```
select      [pno]
from        [shipment]
grouped_by  [pno]
having      [cnt(sno) > 2].
```

The HAVING clause is only used to restrict a partitioned relation so HAVING is always used with GROUPED_BY. The expression in a HAVING clause must be single-valued for each group.

A comprehensive example. Q(3.17). How many people earn less than $8000 in each department ?

```
select      [dno,cnt(ename)]
from        [employee]
grouped_by  [dno]
where       [salary < 8000].
```

It is important to make the distinction between HAVING and WHERE clauses. In comparison with Q(3.16), both the WHERE clause and the HAVING clause are followed by GROUPED_BY, but they differ in that the expression in a HAVING clause involves a built-in function showing the proprerty of each group whereas a WHERE clause only specifies a simple condition in a grouped relation.

3.3 STORAGE OPERATIONS

The three storage operations are: updation, deletion and insertion of tuples.

Update operation. Q(3.18) Change the color of part p2 to yellow.

```
update [part]
set    [color = yellow]
where  [pno = p2].
```

Within a set clause, any reference to an attribute on the right hand side of an equal sign refers to the value of that attribute before the updating has been done. Hence the query Q(3.19). Double the status of all the suppliers in Paris can be written as following.

```
update [supplier]
set    [status = status*2]
where  [city = paris].
```

Update with complex conditions. Q(3.20) Set the quantity to zero for all suppliers in London.

```
update [shipment]
set    [qty = 0]
where  [[sno] in
                 select [sno]
                 from   [supplier]
                 where  [city = london].
```

Single tuple insertion. Q(3.21) Add part p7 (name "WHEEL",color "GREY", weight 27, city = "Hyderabad") to table "PART".

```
insert_into [part] : [p7,wheel,grey,27,hyderabad].
```

Single tuple deletion. Q(3.22) Delete all part records in which the weight is greater than 17.

```
delete [part]
where  [wt > 17].
```

Multiple relations deletion. Q(3.23) Delete those parts from relation "part" whose color is red, weight greater than 10 and is supplied by people in London.

```
delete [part]
where  [color = red,
        wt > 10,
        [pno] in
                 select [pno]
                 from   [shipment]
                 where  [[sno] in
                                  select [sno]
                                  from   [supplier]
```

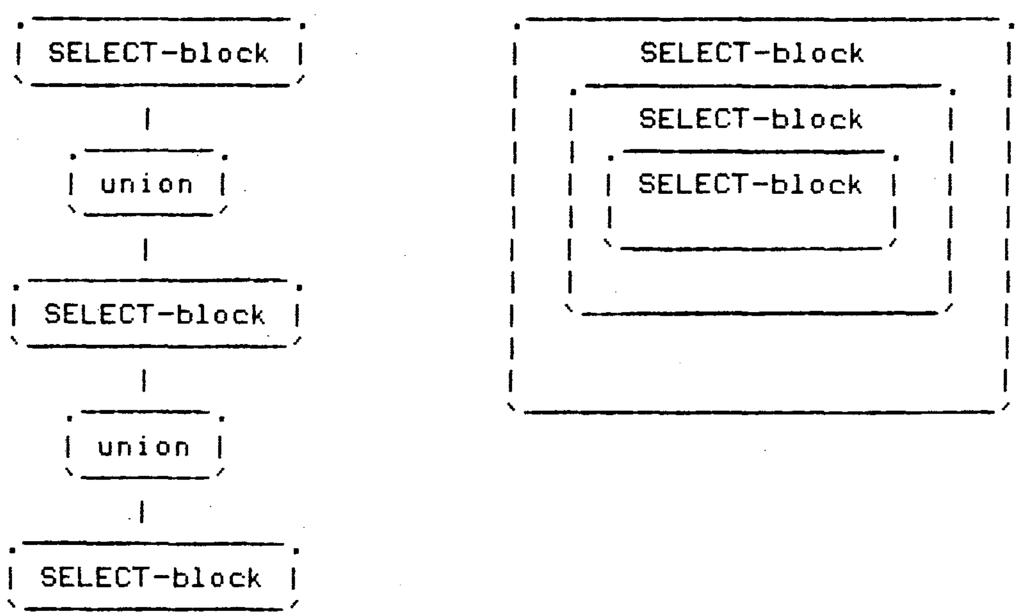
where [city = london]]].

From the discussion above, it can be seen that SQL is a self-contained, structured English keyword language that avoids the use of mathematical notations and concepts. The form SELECT ...FROM... is an expression which, as far as the user is concerned, produces a set of objects. The use of such expressions greatly simplifies the process of constructing a query. Many problems can be expressed in SQL more easily and concisely. However the nonprocedurality of SQL is open to question. The designers of SQL describe it as nonprocedural (descriptive), since a SELECT-block specifies only what data is wanted, not a procedure for obtaining the data. Some others describe it as procedural because there are some procedural elements, such as UNION, and GROUPED_BY, in that language.

3.4 THE SYNTAX OF SQL

In contrast to conventional languages, the syntax of SQL is not all that obvious. The definition of the syntax uses the Backus Naur Form meta-language. Appendix 3 gives the syntax of the query language.

The difference between SELECT-block and SELECT-blocks is given in Figure(3.1) below. It is easy to see that the syntax specification is recursive, permitting a SELECT-block to be nested within an outer SELECT-block. In the vast majority of SQL retrievals we construct one or more "SELECT ...FROM ..." blocks, and such a block specifies a set of tuples, that is a relation. To handle such relationships, we typically specify a tuple from one such block in another nested block, or that a block contains tuples of another block.



SELECT-blocks SELECT-block

Figure(3.1). SELECT-blocks and SELECT-block

3.5 QUERY PROCESSING

Figure(3.2). gives a schematic of the way a query is processed.

The steps involved in query processing are:

1. Input of task in SQL.
2. Parsing the query.
3. Intermediate form generation.
4. Final form generation
5. Execution of the final form.
6. Output.

3.5.1 Input Of Task In SQL

The query is input by the user on a terminal and sent for processing. At the moment this implementation is a bit rigid in the sense that, all queries are to be input in the lower case, and the arguments like relation list, attribute list and condition list have to be enclosed specifically in "[" and "]".

```
.-----------------------------------------------------.
|                    An input query                   |
|                                                     |
|        select [sname,city]                          |
|        from   [supplier]                            |
|        where  [[sno] in                             |
|                         select [sno]                |
|                         from   [shipment]           |
|                         where  [pno = p2,           |
|                                  qty > 200]].       |
|                                                     |
`-----------------------------------------------------'
                          |
                          V
            .----------------------------------.
            |          UNDERSTANDING           |
            `----------------------------------'
                          |
                          V
.-----------------------------------------------------.
|                  Intermediate Form                  |
|                                                     |
|  sub_q(supplier,[_12,_13,_14,_15],[],[],            |
|       [[_12,sno],[_13,sname],[_15,city]],!),       |
|  sub_q(shipment,[_12,p2,_16],[_16>200],[],          |
|       [[_12,sno],[_16,qty]],!).                     |
|                                                     |
`-----------------------------------------------------'
                          |
                          V
.-----------------------------------------------------.
|                     Final Form                      |
|                                                     |
| ans((_13,_15)) :-                                   |
|               shipment(_12,p2,_16),                 |
|               _16>200,                              |
|               supplier(_12,_13,_14,_15).            |
|                                                     |
`-----------------------------------------------------'
                          |
                          V
            .----------------------------------.
            |            Execution             |
            `----------------------------------'
                          |
                          V
                 -------------------
                 sname     city
                 -------------------
                 smith     london
                 jones     paris
```

Figure(3.2). A sample query processing and its answer.

3.5.2 Parsing The Query

This section is devoted to an implementation technique for the SQL parser. The technique employed is fairly general and can be employed for parsing other structured query languages. The emphasis is on those aspects which have been put to practical use in the programming language Prolog, showing how a structured query is understood by logic program solving. The advantage of writing a compiler in Prolog may be realised here.

3.5.3 Considering Keywords As Operators

Prolog can instantiate variables only to structures, atoms or numbers. As a result an arbitrary sentence cannot be read in using the "read(S)" predicate, because S will fail to instantiate to the sentence. One elegant way to overcome this problem is to pre-declare the key-words of the query language as operators. As a consequence of this declaration, read(S), may be able to read in the query and instantiate it to a structure if the input query is error-free.

One of the important strategies in writing the parser of SQL is that each structured English keyword in SQL is considered as an operator in Prolog by declaring it in advance. These operators not only provide syntactic convenience when reading or writing, but also allow the use of simple expressions for what would otherwise have to be implemented through more complicated code containing explicit control structures such as iteration. Each operator has three properties: a position, precedence and associativity. The position can be postfix, prefix or infix, that is, an operator with one argument can go after or before it; an operator with two arguments can go between them. The precedence is an integer and makes expressions unambiguous where the syntax of the terms is not made explicit through the use of parentheses. The associativity makes expressions unambiguous in which there are two operators with the same precedence. If we wish to declare that an operator with a given position, precedence and associativity is to be recognised when terms are read and written, use of the built-in predicate "op" is made. If name is the desired operator (the atom that we want to be an operator), prec the precedence (an integer within the appropriate range), and spec the position/associativity specifier, then the operator is declared by providing the following goal:

```
| ?- op(prec,spec,name).
```

Given below is the declaration of operators in the implementation:

```
:- op(40,xfx,in), op(40,xfx,not_in),
   op(40,xfx,'\='), op(40,xfx,@), op(185,yfx,having),
   op(190,yfx,where), op(192,yfx,grouped_by),
```

```
op(195,yfx,from), op(195,yfx,set), op(195,yfx,':'),
op(200,fx,select), op(200,fx,update), op(200,fx,insert_into),
op(200,fx,delete), op(215,xfy,union), op(200,fx,create),
op(200,fx,table), op(215,fx,'{'), op(200,xf,'}').
```

It is important to study how these operators have been defined. All querys will be read-in and appropriate structures for them will be constructed if the query is correct.

It is this declaration that permits the nesting of operators and such a nesting may be carried to any depth, so that queries in SQL could be constructed with arbitrary complexity. By means of the declaration, an SQL query is internally represented as a function or function form. We shall use Ei to denote the internal representation of an SQL query Qi. A few of the results of the translation are illustrated below:

```
Q1: SELECT [target list] FROM [relation] WHERE
    [conditions] GROUPED_BY [attribute].

E1: get_sub_qs(select(from(TARGET_LIST,
    grouped_by([RELATION],where([GROUPING_ATTRIBUTE],
    CONDITION_LIST)))),Xs,Hash,ANSWER) :-

Q2: SELECT [target list] FROM [relations]
    WHERE [conditions].

E2: get_sub_qs(select(from(TARGET_LIST,where(RELATION_LIST,
    CONDITION_LIST))),Xs,Hash,ANSWER) :-

Q3: SELECT [target list] FROM [relation] GROUPED_BY
    [attribute] HAVING [function
    COMPARISON OPERATOR number].

E3: get_sub_qs(select(from(TARGET LIST,grouped_by([RELATION],
    having([GROUP_ATTRIBUTE],[function COMPARISON
```

```
    OPERATOR number])))),Xs,Hash,ANSWER) :-

Q4: SELECT [target list] FROM [relation]
    GROUPED_BY [attribute].

E4: get_sub_qs(select(from(TARGET_LIST,grouped_by([RELATION],
    [GROUPING_ATTRIBUTE]))),Xs,Hash,ANSWER) :-

Q5: SELECT [target list] FROM [relation].

E5: get_sub_qs(select(from(TARGET_LIST,[RELATION])),
    Xs,Hash,ANSWER) :-

Q6: DELETE [relation] WHERE [conditions].

E6: get_sub_qs(delete(where([RELATION],CONDITION_LIST)),
    Xs,Hash,[]) :-

Q7: DELETE [relation].

E7: get_sub_qs(delete([RELATION]),Xs,Hash,[]) :-

Q8: UPDATE [relation] SET [attribute =
    arithmetic expression] WHERE [conditions].

E8: get_sub_qs(update(set([RELATION],where([ATTRIBUTE=
    EXPRESSION],CONDITON_LIST))),[[Var,Attr]],!,[]) :-

Q9: UPDATE [relation] SET [attribute=arithmetic expression]

E9: get_sub_qs(update(set([RELATION],[ATTRIBUTE=EXPRESSION])),
    Xs,!,[]) :-

Q10:INSERT_INTO [relation] : [constants].

E10:get_sub_qs(insert_into(:([RELATION],CONSTANT_LIST)),
    Xs,Hash,[]) :-

Q11:[select block] UNION [select block].

E11:get_sub_qs(union(SELECT_BLOCK1,SELECT_BLOCK2),
    Xs,Hash,ANSWER) :-
```

Since the SQL parser has a different entry for each possible SQL query, unacceptable queries can be discarded on grounds of a mismatch. This technique allows the implementation to resolve query ambiguities with comparatively little effort.

The subsequent chapter deals with the top level predicates involved in query processing and the execution of the answer set of tuples.

CHAPTER 4

TOP LEVEL QUERY PROCESSING

The top level predicates involved in nested query processing are discussed below. The rationale behind choosing this mode of processing should become clear as the predicates involved in the processing are discussed in detail.

In this implementation of the query processor, the important points to be noted are the creation of variables, unification of variables and treatment of conditions. One more point to be noted here is that the query processor converts the query into an intermediate form (called the CANONICAL LOGIC FORM [12]) which is then converted to the sub-clauses of the answer goal. These sub-clauses are all elements of a list in this implementation. When the processing is complete the elements of this list are called one at a time, and if all the sub-clauses are called successfully then the answer is printed.

By discussing the processing of a query of SELECT-FROM-WHERE type, I shall explain how nested queries are tackled (since this is the most general type of query possible). The reader should not be unduly worried about the fact that only a specific query has been discussed. In fact the general strategy involved in processing the different queries (like- SELECT-FROM-GROUPED_BY-WHERE or DELETE-WHERE or other queries involving WHERE) is not much different. The queries not involving WHERE are much easier to process and in fact are a special case of queries having WHERE, where the condition list is empty.

4.1 CREATION OF VARIABLES

The query which has been parsed, is first processed to extract the attributes asked for by the user from the structure created for the query. Then the attributes are processed and variables are created for them. Specifically if the attribute is simple then, its variable sublist will have two elements and if the attribute is one involving function like AVG, CNT, MAX etc., then the variable sublist will involve three elements. A typical variable list would look like - [[_128,pno],[_271,qty,sum]]. Once this list is created the variables from this list are extracted into just a variable list and kept for writing the answer set. If the above list is the Xs list then the plain variable list is [_128,_271]. Now depending on whether the query is a SELECT or DELETE or INSERT_INTO or UPDATE the appropriate get_sub_qs predicate is matched. The get_sub_qs predicate has the first argument for the query structure, the second one is the list of Xs, the third one for whether the query was one involving negation or NOT and the last argument is the answer list which contains the various tuples to be satisfied for getting the answers.

In assimilating the function (query structure), we have the problem of dealing with the situation when a new attribute is discovered. If the attribute has been encountered before, then the created variable corresponding to the attribute is a join variable that will appear in the intermediate form; otherwise we must ensure that the new variable we assign to it does not accidently coincide with one representing any other attribute. In both cases it is important to keep the correspondence between the attributes and the variables.

While dealing with an attribute, it is helpful not only to remember whether the corresponding variable is desired by the user but also to understand whether the variable is a join-variable or just an independent one. In the actual implementation there was no need to classify the different variables because new variables were created as and when needed and if the attribute was already present then the variable created was unified with the earlier created variable.

4.2 TREATMENT OF CONDITIONS

In general, a SELECT-block will be translated into a sub-query in an intermediate form. The most difficult part in processing the query is to understand the WHERE-conditions on which the answer to a query is to be obtained.

Consider the following example:

```
select [attr11,attr21]
from   [rel1,rel2]
where  [rel1@attr11 > rel2@attr21,
        rel1@attr13 = 5,
        rel2@attr22 in
                                select [attr31]
                                from   [rel3]
                                where  [attr32 = smith]].
```

It is easy to see that there are three kinds of conditions:

1. Join conditions: The comparison between two attributes belonging to different relations, such as

   rel1@attr11 > rel2@attr21

2. Chaining conditions: One of the chaining operators ("in", or "=", or "not_in") is used in the comparison in order to select information from one relation based on the contents of another relation, such as:

```
...rel2@attr22 in
                  select [attr31] from [rel3]
                  where  [sname = smith]].
```

3. Simple conditions: Comparison with a constant, (integer or atom), such as

rel1@attr13 = 5

The predicate "break_off" splits the condition list into two - one containing simple and chaining conditions and the other containing join conditions.

4.3 TOP LEVEL PROLOG IMPLEMENTATION CORRESPONDING TO RECURSIVE PROGRAMMING

Given below are the top level predicates involved in the query processor and their functions:

```
get_sub_qs(select(from(Attrs,where(Relations,Conditions))),
Xs,Hash,Ans) :-
     simplify_cols(Attrs,Cols),
     break_off(Cols4,Xs2,Conditions,S,V,F,FF),
     differ(Relations,S,V,F,FF,R),
     one_by_one(R,Ans).

break_off(S,V,[],S,V,[],[]).
break_off(S1,V1,[A|B],S,V,F,[A1|FF]) :-
  join_cond(S1,V1,A,S2,V2,A1),
  break_off(S2,V2,B,S,V,F,FF).
break_off(S1,V1,[A|B],S,V,[A|F],FF) :-
  break_off(S1,V1,B,S,V,F,FF).

differ([],_,_,_,_,[]).
differ([N1|N2],S,V,F,FF,[(N1,S1,V1,F1,FF1)|R]) :-
  pick_up(N1,S,V,F,FF,S1,V1,F1,FF1,V2,F2,FF2),
  differ(N2,S,V2,F2,FF2,R).

one_by_one([(N,S,V,F1,F2)|[]],Ans) :-
  get_tuples(N,S,V,F1,F2,A,Ans).
one_by_one([(N,S,V,F1,F2)|B],Ans1) :-
  get_tuples(N,S,V,F1,F2,A,Ans),
  one_by_one(B,T), append(Ans,T,Ans1).

get_tuples(N,S,V,C,F2,Sth,Ans) :- relation(K),
```

```
  K =.. [N|L], length(L,Num), get_skel(Num,A),
  break_down(S,V,C,Ss,Vs,Fs1,F21,Ans2),
  reform_f2(Ss,Vs,F2,Sss,Vss,F22), append(F21,F22,Fs2),
  sort(L,A,Sss,Vss), get_sth(L,A,Vss,Sth),
  unific(L,A,Fs1,Fs11), remove(Fs2,Fs22),
  asserta(sub_q(N,A,Fs11,Fs22,Vss,Sth)),
  get_querys(sub_q(N,A,Fs11,Fs22,Vss,Sth),Ans1),
  append(Ans2,Ans1,Ans).

break_down(S,V,[],S,V,[],[],[]) :- !.
break_down(S1,V1,[A|B],S2,V2,Fs1,Fs2,Ans) :- A=..[I,J,K],
  (((atom(K) ; number(K)), simplify_relatr([J],[J1]),
    cut_off1(S1,V1,I,J1,K,S,V,F1,F2,Ans1))
    ;
   cut_off(S1,V1,I,J,K,S,V,F1,F2,Ans1)),
  break_down(S,V,B,S2,V2,F12,F22,Ans2),
  (F1==!, Fs1=F12; Fs1=[F1|F12]),
  (F2==!, Fs2=F22; Fs2=[F2|F22]),
  append(Ans2,Ans1,Ans).

cut_off(S,V,in,J,K,S1,V1,!,!,Ans) :-
  simplify_cols(J,L), subst(S,V,L,Xs,S1,V1),
  arg(1,K,Temp), ext(Temp,[Atr|_]),
  create_var(Atr,Attr1), get_only2(Xs,Tem2),
  set_diff(Attr1,Tem2,Tem3),
  get_xs(Tem3,Xs,Xs1), get_sub_qs(K,Xs1,!,Ans),
  extract(Xs,Tempo), extract(Xs1,Tempo).
cut_off(S,V,=,J,K,S1,V1,!,!,Ans) :-
  cut_off(S,V,in,J,K,S1,V1,!,!,Ans).
cut_off(S,V,not_in,J,K,S1,V1,!,!,Ans) :-
  simplify_cols(J,L), subst(S,V,L,Xs,S1,V1),
  arg(1,K,Temp), ext(Temp,[Atr|_]),
  create_var(Atr,Attr1), get_only2(Xs,Tem2),
  set_diff(Attr1,Tem2,Tem3),
  get_xs(Tem3,Xs,Xs1), get_sub_qs(K,Xs1,not,Ans),
  extract(Xs,Tempo), extract(Xs1,Tempo).
cut_off(S1,V1,I,J,K,S,V,!,F2,Ans) :- !, simplify_cols(J,J1),
  subst(S1,V1,J1,[Xs],S,V), arg(1,K,R), ext(R,[H|T]),
  H = [H1|T1],
  ((atom(H1), Xs1 = [[Y2|J1]])
  ;
   (H1 =.. [Op,_],
    append(J1,[Op],Atr1), Xs1 = [[Y2|Atr1]])),
  get_sub_qs(K,Xs1,!,Ans),
  extract(Xs,[Y1]), F2=..[I,Y1,Y2].

cut_off1(S1,V1,I,J,K,S,V,F1,!,[]) :-
  substitute(S1,V1,I,J,K,S,V), F1 =.. [I,J,K].
```

break_off(S,V,C,S1,V1,F,FF) :- The predicate "break_off" is used to break off such a SELECT-block in which the FROM clause contains several relations. As a result, by creating join variables, the join conditions specified in the WHERE clause will be replaced by relevant formulas.

This predicate breaks the input list of attributes S, input list of variables V and conditions C into output list S1 consisting of S plus those attributes obtained from C which are not already present in S, (same definition for V which gets converted to V1). The conditions C get broken into two; F - the independent formulas and FF - the relevant formulas. F consists of all simple conditions and chaining conditions, whereas FF consists of all conditions involving comparison with the same attributes of other relations. For the moment all the chaining conditions are assumed to be simple ones.

break_down(S,V,C,S1,V1,Fs1,Fs2,Ans) :- The predicate "break_down" is only used to break down such a SELECT-block in which the WHERE clause may contain chaining conditions. As a result, by creating join variables, the chaining conditions will be broken down into pseudo-independent formulas and relevant formulas.

Predicate break_down deals with the list of conditions C, containing simple conditions and chaining conditions. Each condition is tested to see whether it is simple or chaining. If the condition is simple predicate cut_off1 is called else cut_off is called. As a result all new attributes that are found in C are added to S to give S1, new variables are added to V to give V1. New independent formulas and new relevant formulas that accrue from chaining conditions are put in Fs1 and Fs2 respectively. Ans is the list of answers obtained so far from processing the nested sub querys.

differ(N,S,V,F,FF,R) :- The function of the predicate "differ" is that from the collection of all attributes S, variables V, non-join conditions F, and relevant formulas FF which is the result of the execution break-off, each relation picks up its own attributes S1, variables V1, non-join formulas F1, and relevant formulas FF1 respectively, and puts them in the box R in the particular order:

[(N1,S1,V1,F1,FF1),(N2,S2,V2,F2,FF2),...]

The box R will be dealt by the predicate one_by_one.

one_by_one(R,Ans) :- This predicate takes the list R obtained above and processes the list of sub-querys (nested querys) for all the relations present in R, and gets the final list of answer templates for finding answers.

get_tuples(N,S,V,F1,F2,A,Ans1) :- As the sub-goal of the predicate one_by_one, the predicate get_tuples performs the storing of the query tuple in the intermediate form.

Typically the intermediate form is:

sub_q(Name,Attr_lis,Indef_lis,Relef_lis,Var_lis,Grp_atr_lis)

Here Name is the name of the relation, Attr_lis the list of attributes involved in relation Name, Indef_lis is the list of independent formulas, Relef_lis is the list of relevant formulas, Var_lis is the list of variables involved and their associations with the attributes and whether they have any aggregation operations on them. Grp_atr_lis is the list containing the attribute whose aggregation is needed. This predicate performs the intent of one_by_one for each element of R. The Ans1 obtained here is a sublist of Ans in the above predicate one_by_one. 'A' here is instantiated to a list containing any aggregation functions like grouped_by, SUM, COUNT, AVG, MAX, MIN.

cut_off(S,V,in,J,K,S1,V1,!,!,Ans) :- Deals with sub-querys having 'in' as the chaining operator. J being the 'target list' of the sub-query K. I have put target list in quotes to indicate that the target list in the sub query K may not eactly match; in the sense that, for instance J might be [pno,qty] whereas the target list in K might be [pno,sum(qty)]. Ans is the answer obtained after processing the sub-querys K. S,V get modified to S1,V1 containing all attributes and variables in J not already present in S and V respectively.

cut_off(S,V,=,J,K,S1,V1,!,!,Ans) :- Here '=' is used instead of 'in', and its meaning is the same as the cut_off given above.

cut_off(S,V,not_in,J,K,S1,V1,!,!,Ans) :- Is the same as above but for the chaining operator which is 'not_in'.

cut_off(S,V,I,J,K,S1,V1,!,F2,Ans) :- Deals with all those chaining conditions where the chaining operator happens to be one of >, >=, <, =<, =.

cut_off1(S,V,I,J,K,S1,V1,F1,!,[]) :- Deals with all simple conditions. F1 gets instantiated to the simple condition comparison.

If there exist nested queries embedded in the chaining condition then the predicate get_sub_qs will be called again and again until the break_off and break_down processing hits the bottom of the recursion.

It would be relevant here to show into what intermediate form the nested query is reduced after a major part of the processing is over.

In SQL:

```
select [ename]
from   [employee]
where  [age < 35,
        [salary] >
                          select [avg(salary)]
                          from   [employee]].
```

In Intermediate Form:

```
sub_q(employee,[_13,_14,_15,_16],[_14<35],[_15>_24],
      [[_13,name],[_14,age],[_15,salary]],!),
sub_q(employee,[_17,_18,_19,_20],[],[],[[_19,salary]],
      [[_24,salary,avg]]).
```

In the final executable form:

```
[avg(_19,employee(_17,_18,_19,_20),_24),
      employee(_13,_14,_15,_16),
              _14<35, _15>_24]
```

It can be seen that the nested query is decomposed into a simple collection of query tuples in its intermediate form. This intermediate form is then converted into the final executable form, and will be ultimately executed.

4.4 TRACE OF QUERY PROCESSING BY AN EXAMPLE

It would be prudent here to discuss the function of the top level predicates with the help of an example.

EXAMPLE: Find the names and ages for employees in department "d1" who earn more than the average salary of those who are in the same department with age greater than 30:

```
select [ename,age]
from   [employee]
where  [dno = d1,
        [salary] >
                  select [avg(salary)]
                  from   [employee]
                  where  [dno = d1, age > 30]].
```

The query is read through the built-in predicate "read(S)", where S will now be instantiated to:

```
S = select [ename,age] from [employee] where [dno=d1,[salary] >
    select [avg(salary)] from [employee] where [dno=d1,age>30]]
```

whose internal representation is:

```
S = select(from([ename,age],where([employee],[dno=d1,[salary]>
select(from([avg(salary)],where([employee],
[dno=d1,age>30]))))])))
```

Then a predicate "create_var([ename,age],L)", creates a list of variables for final output. Here L is instantiated to:

```
L = [[_412,ename],[_428,age]]
```

Also a separate list of only these variables is kept in L1, which is done with the help of "extract(L,L1)" where L1 is:

```
L1 = [_412,_428]
```

Another list L2 is created from the attribute list in S the query structure.

```
L2 = [ename,age]
```

After confirming what is needed the predicate "get_sub_qs(S,L,!,_21)", is called, where _21 will ultimately be containing the answer tuples.

From the fixed structure of S above, the list containing the relation names can be obtained. Therefore it is now easy to obtain a skeleton list containing variables, the length of the list being equal to the number of attributes of the relation in question.

Now the predicate

```
break_off(L2,L,[dno=d1,[salary] > select [avg(salary)]
from [employee] where [age>30, dno=d1]],_542,_543,_544,_545)
```

is processed, which gives the output as:

```
break_off(L2,L,[dno=d1,[salary] > select [avg(salary)]
from [employee] where [age>30, dno=d1]],[ename,age],
[[_412,ename],[_428,age]],[dno=d1,[salary]>select [avg(salary)]

from [employee] where [age>30,dno=d1]],[])
```

Here the variable _542 is the same as the list L2, since no new attributes were discovered in the condition list - the third argument of break_off. Also it might be remembered here that the nested SELECT is ignored for the moment. Therefore there is no corresponding change in the fifth argument (_543) of break_off, and it is the same as L. _544 is the same as the condition list, because the final argument (_545) is the empty list, i.e. the condition list is broken into the fifth and sixth arguments where the former contains simple conditions and nested "SELECT" and the latter contains join conditions.

The predicate

```
differ([employee],[ename,age],[[_412,ename],[_428,age]],
[dno=d1,[salary]>select [avg(salary)] from [employee] where
[age>30,dno=d1]],[],_546)
```

becomes

```
differ([employee],[ename,age],[[_412,ename],[_428,age]],
[dno=d1,[salary]>select [avg(salary)] from [employee] where
[age>30,dno=d1]],[],[(employee,[ename,age],[[_412,ename],
[_428,age]],[dno=d1,[salary]>select [avg(salary)] from
[employee] where [age>30,dno=d1]],[])]
```

Here the last argument gets instantiated to the list whose elements are such that each element contains the attributes, variables and conditions for each relation. In this case since there was only one relation the final list likewise contains only one element.

The predicate

```
one_by_one([(employee,[ename,age],[[_412,ename],[_428,age]],
[dno=d1,[salary]>select [avg(salary)] from [employee] where
[age>30,dno=d1]],[])],_549)
```

becomes

```
one_by_one([(employee,[ename,age],[[_412,ename],[_428,age]],
[dno=d1,[salary]>select [avg(salary)] from [employee] where
[age>30,dno=d1]],[])],[avg(_1949,[employee(_1935,_1942,_1949,d1),

_1942>30],_1224),employee(_412,_428,_1100,d1), _1100>_1224])
```

Here one_by_one, as the name suggests takes one element each of the list represented by _546, and processes them to give the final answer list _549. The predicate one_by_one makes use of predicates like "cut_off" and "cut_off1" which act on the nested SELECT and simple conditions (in the condition list) respectively.

Apart from this some other useful predicates remove simple conditions involving "equality". For instance if we have one of the answer tuples as:

```
employee(_1935,_1942,_1949,_1956)
```

and condition list as:

```
[dno=d1]
```

Then the answer tuple might as well be:

employee(_1935,_1942,_1949,d1)

and the condition list can now be made empty.

Once the answer set is obtained, which is:

```
[avg(_1949,[employee(_1935,_1942,_1949,d1),_1942>30],_1224),
employee(_412,_428,_1100,d1), _1100>_1224]
```

the predicate

```
calling1([avg(_1949,[employee(_1935,_1942,_1949,d1),
_1942>30],_1224),employee(_412,_428,_1100,d1),_1100>_1224])
```

calls each element of the list one at a time. In this case first

avg(_1949,[employee(_1935,_1942,_1949,d1),_1942>30],_1224)

is called which becomes:

avg(_1949,[employee(_1935,_1942,_1949,d1),_1942>30],8050)

It might be added here that "avg" finds the average of the first argument which satisfies certain conditions in the second argument, and then puts its answer in the third argument. Once this is done, the predicate calling1 calls its second element, which gets instantiated to:

employee(john,34,8600,d1)

Now calling1 calls for the third time as:

8600 > 8050

It exits with all the elements of calling1, satisfied, and then the predicate "write_attr", does pretty printing on the desired output stream. Then fail is encountered, whereby different elements of calling1 are resatisfied and thus ensuring exhaustive database search, until no more solutions are possible.

The final answer table is printed as:

ename	age
john	34
henry	29

Mention must be made here of the predicate "break_down". In the query discussed since there are no join conditions, this is not of use, but in queries involving join conditions, break_down does analogously to join conditions what break_off does to the simple and chaining conditions.

In concluding this chapter I must add that most of the query processing involves a lot of unification, and creation of variables. This observation makes the relevance of Prolog as a database language all the more significant.

CHAPTER 5

NEGATION, AGGREGATION AND OTHER USEFUL PREDICATES

This chapter discusses negation involved in various queries, its interpretation and then goes on to discuss other facilities provided by SQL - namely aggregation operations, and also discusses some useful predicates.

5.1 NEGATION AS NONPROVABILITY

Many queries involve (explicitly or implicitly) the concept of negation. For e.g.1 :

```
select [sname]
from   [supplier]
where  [[sno] not_in
                   select [sno]
                   from   [shipment]
                   where  [pno = p2]].
```

5.1.1 The open and closed world assumptions

Care must be taken when representing negative information, as under certain circumstances, its operational and declarative interpretations might not coincide. Basically, two assumptions are possible: the open and closed world assumptions.

1. The open world assumption corresponds to the usual first order approach to query evaluation: Given a database DB and a query Q, the only answers to Q are those which obtain from proofs of Q given DB as hypotheses. With this definition we could represent negative information explicitly, for example, by adding assertions of the form "not(p)".

2. Under the closed world assumption, certain answers are admitted as a result of failure to find a proof. More specifically, if no proof of a positive ground literal exists, then the negation of that literal is assumed true. This can be viewed as equivalent to implicitly augmenting the given database with all such negated literals.

The second order approach presupposes a complete knowledge of the domain being represented. For many domains of application, this assumption is appropriate. For example it is natural to assume that those employees who are not stated to work at department "d1", do not work there. In other words, the individuals named in the database are the only individuals there are. For such domains, an implicit representation is usually preferable, as negative information outnumbers by far positive information, and it would be redundant to represent it explicitly when it can simply be established by default.

5.1.2 Negation as failure

In dealing with how meaning can be assigned to answers to negative questions, the standard negation of first order logic is problematic to implement and seems inappropriate for many purposes. By contrast, the great advantage of the closed world assumption is the ease with which it can be implemented. For pragmatic reasons Prolog provides a partial implementation of non-provability. To show that P is false we do an exhaustive search for a proof of P. If every possible proof fails, not P is 'inferred'. Given below is the implementation of not P.

```
not(P) :- call(P), !, fail.
not(P).
```

Therefore the query, e.g.1 would be interpreted as: Find supplier names for suppliers who are not known to supply part p2.

Unfortunately, dealing with closed world assumption does not itself guarantee negation by default to behave as expected within logic programs. Let us consider the following database which contains the following facts.

```
male(david).
male(albert).
female(susan).
female(margaret).
```

Under the closed world assumption, "male(susan)" and "male(margaret)" must be considered to be false. But whereas the queries "not male(susan)" and "not male(margaret)" will be correctly evaluated as "yes" since both "male(susan)" and "male(margaret)" will fail, the query "not male(X)" will just fail instead of just retrieving susan and margaret as values for X since male(X) can be proved by matching X with either david or albert.

As this example shows, within the closed world databases, negated predicates are only safe to evaluate when they contain no free variables, i.e., when all the variables they contain have been replaced by ground terms.

One possible solution to this problem is to dynamically postpone the evaluation of some atoms in a query, according to predefined criteria. The execution of a formal "not P" can be blocked until P contains no free variables.

5.2 THE PREDICATE "GROUP"

High level query languages very often involve the partition facility which conceptually rearranges a relation into groups such that in any one group all tuples have the same value for the grouped attribute.

The predicate group can be implemented as:

```
group(N,G,N) :- call(G), only(N).
only(N) :- not(ffound(N)), asserta(ffound(N)), !.
```

where ffound is a functor to remember what has been grouped so far during a query execution. There is a difference between the two N's which are the arguments of the second order predicate "group". If and only if the sub-goal "only(N)", in which the N was instantiated by calling G succeeds, then the predicate group succeeds, and only at that time the second N can be matched.

5.3 THE PREDICATE "UPDATE"

As far as the storage operations are concerned, we introduce a predicate update with three arguments. We can consider update(Old,G,New) as saying that replace an instantiated tuple G in the database by a new tuple which is the same as G except that the occurrence of Old becomes New. Given below is the "updat" predicate.

```
updat(X,G,Y,Attr) :-
  (calling1(G) ; (not(is_list(G)), call(G))),
  updated(X,G,Y,Attr), fail.

updated(X,G,Y,Attr) :- {find(Y,Y1)}, {integ(Attr,Y1)},
  replace(Y1,X,G,New), nl, retract(G),
  asserta(removable(G)), asserta(savable(New)),
  asserta(New), write(G),
  write('  :  updated to  :  '), write(New), !.

find(X,X) :- atomic(X).
find(E,R) :- R is E.

replace(New,Old,Old,New).
replace(New,Old,Val,Val) :- atomic(Val).
```

```
replace(New,Old,Val,NewVal) :-    functor(Val,Fn,N),
  functor(NewVal,Fn,N),
  subst_args(N,New,Old,Val,NewVal).

subst_args(0,_,_,_,_) :- !.
subst_args(N,New,Old,Val,NewVal) :- arg(N,Val,OldArg),
  arg(N,NewVal,NewArg), replace(New,Old,OldArg,NewArg),
  N1 is N-1, subst_args(N1,New,Old,Val,NewVal).
```

5.4 AGGREGATE FUNCTIONS

One of the important function of DBMS's is the application of aggregation or statistical functions. The functions that are currently supported are CNT, SUM, AVG, MAX and MIN. For example to calculate the average salaries in the relation EMPLOYEE in the department "d1", we can ask:

```
select [avg(salary)] from [employee] where [dno = d1].
```

The execution could be in the way that, first the relation EMPLOYEE would be selected under the department "d1", then projected on the attribute "salary"; finally the projection would be sent to the AVG function.

A formal definition of an aggregation function is:

An aggregate function takes a set of tuples (a relation) as an argument and produces a single simple value (usually a number) as a result.

5.5 IMPLEMENTATION OF AGGREGATION FUNCTIONS

The implementation strategy is based on two principles:

1. Any aggregate function is based on a base relation.

2. The query tuple involving an aggregate function in intermediate form will introduce an aggregate-predicate which incorporates the expressive power of high-order predicate calculus in the intermediate form.

Therefore the implementation of the above query would be as:

avg(_45,employee(_43,_44,_45,d1),_50)

where the variable _50 is instantiated to the average salary of the employee in department d1.

In general, the extension allows goals of the form:

aggregate_predicate(V,P,X)

to be read - the aggregate function specified by "aggregate_predicate" on V's such that P is provable is X, where P is a goal or a conjunction of goals.

With this implementation strategy, let us consider a more sophisticated query: Find the employee names and ages for employees in department "d1" who earn more than the average salary of those who are at the same department with age over 30:

```
select [ename,age]
from   [employee]
where  [dno = d1,
        salary >
                  select [avg(salary)]
                  from   [employee]
                  where  [dno = d1, age > 30]].
```

whose final executable form would be:

```
[avg(_41,[employee(_39,_40,_41,d1),_40>30],_50),
 employee(_51,_52,_53,d1), _53>_50]
```

N.B. Here _51 and _53 are the variables which get ultimately instantiated to the answers - employee name and employee salary satisfying the criteria.

In this implementation, the second argument of all aggregation predicates can be instantiated to a list containing arbitrary number of conditions such that the answer which will be instantiated in the third argument satisfies all these conditions.

5.6 A LOOK AT SOME OTHER USEFUL PREDICATES

The Prolog version of the query obtained so far incorprorates a simple proof procedure. However, the extralogical primitives provided by the Prolog system such as cut, fail, fork play an important role in improving efficiency, since they can decide how the proof can be carried out.

Let us consider the execution of

a :- b, c, d, e, f, ...

Suppose we know that the sub-goal f fails, and suppose we also know that there is no point in trying to resatisfy d and e because they will anyhow fail. In such a case when backtracking occurs (when f fails), we would like the subgoals d and e to be skipped for resatisfying and the subgoal c to be resatisfied.

To circumvent this problem we define a pair of prefix and postfix operators, "{" and "}" , whose intent is given in the following Prolog definition.

{X} :- call(X), !.

The X here can be instantiated to any number of sub-goals, which we know are invariably going to fail on backtracking. By doing so these sub-goals will never be attempted to be resatisfied in the event of a backtrack. For the example given above we may include d and e as shown below:

a :- b, c, {d, e}, f, ...

Now when f fails, e and d will not be resatisfied but directly the attempt to resatisfy c will be made. It means that these sub-goals d and e will only be entered from the "left" so to say.

The advantages of having such an implementation are obvious.

Since there is no in-built definition of the comparison operator "=" , it has been defined below:

```
A \= B :- A = B, !, fail.
A \= B.
```

I would like to highlight another simple predicate called "pick_up", which is the backbone of this implementation. It has been used to unify variables and for instantiating variables to atoms, and often to pick up certain elements of the list given the list and their positions in it from the beginning of the list.

The implementation in Prolog of pick_up is given below:

```
pick_up([],_,[]).
pick_up([M|N],L,[X|Y]) :- mem(M,X,L), pick_up(N,L,Y).

mem(1,X,[X|_]).
mem(N,X,[_|Y]) :- mem(M,X,Y), N is M+1.
```

This predicate may best be explained by giving examples:

```
  e.g.1 pick_up([4,2],[a,b,c,d],L).
          gives L = [d,b]

  e.g.2 pick_up(L,[a,b,c,d],[c,a])
          gives L = [3,1]

  e.g.3 pick_up([1,4],[a,b,c,d],[A,B])
          gives A = a, B = d

  e.g.4 pick_up([2,1],[A,B,C,D],[F,G])
          After this F shares with B and G with A

  e.g.5 However pick_up(L,[A,B,C,D],[C,A])
        is meaningless, in the  sense that
        L is instantiated to [1,1]
```

Other predicates are extract_attr which dives deep into a list and extracts all atoms and numbers and collects them in a list.

The implementation of extract_attr is given below:

```
extract_attr([],[]).
extract_attr(X,[X]) :- atomic(X).
extract_attr(T,L) :- arg(1,T,X),
        extract_attr(X,L1), arg(2,T,Y),
        extract_attr(Y,L2), append(L1,L2,L).
```

Apart from the top level predicates discussed in Chapter 4 and some of the important ones discussed above there are other predicates involved in various functions of finding aggregation of attributes and for simplifying the attributes, variables etc. during various stages of processing. It would be irrelevant to discuss them in detail as the purpose they serve is obvious from the implementation itself.

CHAPTER 6

SUGGESTIONS FOR IMPROVEMENT AND IDEAS FOR FUTURE WORK

At the end of implementing any system, one invariably feels that there is room for improvement in almost every aspect of the implementation. I shall discuss a few important points where I feel the performance of this system will be considerably improved.

The idea that suggestions for imrovement and ideas for future work go hand in hand because both these aspects are interconnected. One might build over the already existing work or one might try to rectify the flaws in this and then go ahead by building over it.

I shall specify the shortcomings of the implementation and a possible way to remove them as I list them:

1. The existing implementation allows queries to be input in the lower case only, and the attribute list, relation list, and condition lists have to be specially enclosed in brackets ("[" and "]"), due to the peculiarities of the Prolog uesd. There is a possibility of including a pipe (a utility which helps in giving the output of one program as the input of another) written in any of the procedural languages, where the input can be more flexible in that the query may be specified in either

upper or lower case and that elements of the attribute, relation, condition lists need not be specifically enclosed in brackets. In other words the job of this program is to make query entry more elegant.

2. After the processing of the query is over, a list of query tuples is created which have to be searched through the database. It can be shown that in certain cases, reordering the query tuples in the final list might mean the difference between getting almost instantaneous answers or having to wait for hours. Hence an optimizer, ensures that all queries are answered in a reasonable amount of time [1].

3. Improving the performance of this system can be done by designing more powerful predicates which involve less computation. This may be realized by using partial data structures called difference lists [6] and removal of redundant code. What I have suggested will hopefully be clearer with the following example:

   We write append predicate as:

   ```
   append([],L,L).
   append([H|X],Y,[H|Z]) :- append(X,Y,Z).
   ```

   If the purpose of this predicate is just to append two lists then it might as well be done using:

   ```
   append(X,Y,Y,X).
   ```

   However now we need to write - append two lists L1 and L2 to give L3 as:

append([L1|A],L2,A,L3).

For e.g. append([1,2,3|A],[c,d,e],A,Ans).

gives Ans = [1,2,3,c,d,e].

in one shot.

Many such improvements are possible in the implementation, and this will substantially reduce query processing time. Such optimizations and improvement in the style of writing the code will undoubtedly make it more understandable and amenable to changes without much difficulty.

4. Comparison of the performance of this implementation in processing queries (i.e. time taken to process it) with that of SPREADS (Simple-to-use Portable RElAtional Database System). has been a bit difficult because this implementation is on the VAX 11/750 while SPREADS is on IMPACT. Hoewever, preliminary studies indicate that this implementation is much slower, maybe by three or four times than SPREADS because of a few unavoidable reasons and otherwise.

Other possibilities for improvements are discussed below. They are:

1. The implementation is working on a Prolog interpreter, whereas SPREADS is compiled and ready to execute. This is one major factor which prevents any really reasonable benchmarking to be done.

2. The second and no less important factor is that there is some redundancy in the code and other suggestions for improvement mentioned above (use of partial data structures etc.), if implemented, should make it preform faster.

3. One more problem which significantly effects performance is the hardware itself. Most hardware has been designed keeping in view the programming languages it would support (usually procedural languages like Pascal or C), and since Prolog is a relatively new language none of the designs include any features germane to it.

4. Also, it has been found that changing the output stream to write in a file instead of the screen significantly improves its processing time, by a factor of two or three, especially where lot of I/O is involved.

5. Many other improvements are possible like use of disjunction in the condition list of the query itself. For instance: Get all supplier names from suppliers who are either in Paris or have status greater than 20, is now written as:

```
    select [sname] from [supplier] where [city = paris]
union
    select [sname] from [supplier] where [status > 20].
```

but could be:

```
select [sname]
from   [supplier]
where  [city = paris ; status >20].
```

6. There is also the possibility of including, queries of the type: Get those supplier names who "only" supply red parts. Many such extensions can be included in the system to enhance its power.

7. Integrity constraints is one more area where a lot can be done. At present the integrity checks are carried out only on the new value; i.e. to see whether the new value corresponding to a particular attribute violates the constraints or not. That means there are no temporal integrity checks. For instance, when the age of an employee is updated one should check that the new age is greater than the earlier one. Other integrity constraints may be in the form of rules. For instance : If the employee is in department d1, then his salary should not exceed $20,000 and if the employee is in any other department then no such constraints hold. At present the only integrity constraints are in the form of simple rules, where the body of the rule is the "cut".

8. Another problem to be taken care of is that of incomplete databases, i.e., certain attribute values in some relations may not be filled up. For such cases, there should be a consistent interpretation regarding what needs to be done during query processing.

9. Another area of interest would be to take care of deleting relations where deleting certain tuples from a relation, may also effect other tuples in other relations. For instance after deleting suppliers who supply part p2, we might even delete tuples in relation shipment where part p2 no more exists because it is not supplied. The presence of this data is not useful in any way as it might be "inaccessible". For the moment deleting tuples, is done only for a given relation.

10. Many other appendages may be added to the program which would make it more user friendly and make debugging of erroneous queries simpler. The debugging aids to check wrongly input queries are at the moment very primitive.

11. To make the system really useful while updation and insertion and deletion of data is done, some work needs to be done in the area of file handling. This is of paramount importance because proper recovery procedures should be available in the event of a system crash.

12. Other built-in predicates, specific to the efficient manipulation of databases - like database search are available, but have not been made use of. There is a possibilty of improving database search using these built-in predicates which make use of keys for searching.

13. At present queries involving WHERE allow only one aggregation operator, if the WHERE contains a nested SELECT. This can therefore be modified to include more than one aggregation operator.

14. There is no implementation of undoing any updation, deletion or insertion operation when it has been found that the updation, deletion or insertion has been done wrongly.

The implementation is raw and a lot of fine points and improvements need to be done. The one difference between this system and SPREADS is that while SPREADS does not allow nesting of queries, one can ask nested nested queries, here - theoretically to any depth. This feature is helpful in answering the negative queries, like getting all suppliers names who do not supply part p2, and in many places where complicated queries may be used to get answers where it is not possible to do so on SPREADS.

Testing of this system was done extensively on the supplier-part-department database and some of the times taken to answer the queries have been given. Paucity of time precluded testing this implementation on databases of any reasonable size, but there should be no problems. When writing answers, Prolog suffers from one drawback, in that, the attributes have to be written one at a time and a somewhat clumsy method of giving tab spacing has to be given (for an elegant output - refer to the predicates "write_attr" and "writeln" in the source listing). In SPREADS this problem does not exist, because it can write all values at once.

One more test was carried out on a single relation database, involving about a hundred tuples and ten attributes. It was found to work correctly and reasonably fast.

One area of interest in making the system more versatile would be in making it more user friendly. For instance by including fragments of natural language into it. Work in this area has already been done [13]. However this itself will be a part of another project.

Work has also been done in the area of knowledge representation, and Prolog has been found to be quite a powerful tool. Use of Prolog for making database systems more intelligent involves the representation of knowledge in the form of semantic networks. Here instead of just representing plain facts as has been done in conventional databases, rules can be used to implement databases, and thereby the transition from databases to knowledge bases is possible.

Prolog at present can handle databases of limited size, and its performance in retrieving information from large databases is yet to reach the speeds of more conventional approaches. This can certainly be an interesting area of work.

I would like to conclude with the comment that all these possibilities are exciting and a lot work can be done too.

APPENDIX 1

THE SAMPLE DATABASE.

supplier

sno	sname	status	city
s1	smith	20	london
s2	jones	10	paris
s3	blake	30	paris
s4	clark	20	london
s5	adams	30	athens

shipment

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

part

pno	pname	color	wt	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

stock

dept	pno	qty
d1	p1	600
d1	p2	350
d2	p1	450
d3	p1	500
d3	p4	220

department

dept	city	manager
d1	london	smith
d2	perth	long
d3	london	lee

employee

ename	age	salary	dept
john	34	8600	d1
morgan	24	6300	d2
lewis	42	9000	d3
long	34	8500	d2
henry	29	12300	d1
thomas	31	7300	d3
martin	45	7500	d1

APPENDIX 2

SAMPLE QUERIES, THEIR ANSWERS AND THEIR RESPONSE TIMES

e.g.1 Get full details of all parts supplied.

```
select * from [part].
```

pno	pname	color	wt	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

Time taken: 4.9833s

e.g.2 Find all parts, their weight and the city where they are kept.

```
select [pname,wt,city] from [part].
```

pname	wt	city
nut	12	london
bolt	17	paris
screw	17	rome
screw	14	london
cam	12	paris
cog	19	london

Time taken: 3.5s

e.g.3 Find the average age, number of employees and average salary of all employees.

```
select [avg(age),cnt(ename),avg(salary)]
from   [employee].
```

avg(age)	cnt(ename)	avg(salary)
34.1429	7	8500

Time taken: 2.75s

e.g.4 For each part supplied, get the part number and the total quantity supplied of that part.

```
select          [pno,sum(qty)]
from            [shipment]
grouped_by      [pno].
```

pno	sum(qty)
p1	600
p2	1000
p3	400
p4	500
p5	500
p6	100

Time taken: 3.15001s

e.g.5 Find the average age, number of employees and average salary in each department.

```
select          [cnt(ename),avg(salary),avg(age)]
from            [employee]
grouped_by      [dno].
```

cnt(ename)	avg(salary)	avg(age)
3	9466.66	36
2	7400	29
2	8150	36.5

Time taken: 3.8334s

e.g.6 Get part numbers for all parts supplied by more than two suppliers.

```
select          [pno,sum(qty)]
from            [shipment]
grouped_by      [pno]
having          [cnt(sno)>2].
```

pno	sum(qty)
p2	1000

Time taken: 2.6s

e.g.7 Set the quantity to zero of all stocks.

```
update [stock] set [qty = 0].
```

```
stock(d1,p1,600) : updated to : stock(d1,p1,0)
stock(d2,p2,350) : updated to : stock(d2,p2,0)
stock(d2,p1,450) : updated to : stock(d2,p1,0)
stock(d3,p1,500) : updated to : stock(d3,p1,0)
stock(d3,p4,220) : updated to : stock(d3,p4,0)

Time taken: 2.70003s
```

e.g.8 Enhance salary of all employees to double their original salary.

```
update [employee] set [salary = salary * 2].

employee(john,34,8600,d1) : updated to :
                      employee(john,34,17200,d1)
employee(morgan,24,6300,d2) : updated to :
                      employee(morgan,24,12600,d2)
employee(lewis,42,9000,d3) : updated to :
                      employee(lewis,42,18000,d3)
employee(long,34,8500,d2) : updated to :
                      employee(long,34,17000,d2)
employee(henry,29,12300,d1) : updated to :
                      employee(henry,29,24600,d1)
employee(thomas,31,7300,d3) : updated to :
                      employee(thomas,31,14600,d3)
employee(martin,45,7500,d1) : updated to :
                      employee(martin,45,15000,d1)

Time taken: 5.06667s
```

e.g.9 Change the color of all parts to red.

```
update [part] set [color = red].

part(p1,nut,red,12,london) : updated to :
                      part(p1,nut,red,12,london)
part(p2,bolt,green,17,paris) : updated to :
                      part(p2,bolt,red,17,paris)
part(p3,screw,blue,17,rome) : updated to :
                      part(p3,screw,red,17,rome)
part(p4,screw,red,14,london) : updated to :
                      part(p4,screw,red,14,london)
part(p5,cam,blue,12,paris) : updated to :
                      part(p5,cam,red,12,paris)
part(p6,cog,red,19,london) : updated to :
                      part(p6,cog,red,19,london)

Time taken: 4.31667s
```

e.g.10 Add part p7 (name WHEEL, color BLACK, weight 24, city HYDERABAD) to table "part".

```
insert_into [part] : [p7,wheel,black,24,hyderabad].

Fact: part(p7,wheel,black,24,hyderabad)
      inserted into the database

Time taken: .866673s
```

e.g.11 Find the name of employees, department and salaries who earn the most in their respective departments.

```
select        [ename,age,max(salary),dno]
from          [employee]
grouped_by    [dno].
```

ename	age	max(salary)	dno
henry	29	12300	d1
long	34	8500	d2
lewis	42	9000	d3

Time taken: 3.66667s

e.g.12 Get maximum salary, age and department number from employee grouped by department number.

```
select        [max(salary),dno]
from          [employee]
grouped_by    [dno].
```

max(salary)	dno
12300	d1
8500	d2
9000	d3

Time taken: 2.26667s

e.g.13 Delete relation stock.

```
delete [stock].

stock(d1,p1,600) deleted.
stock(d2,p2,350) deleted.
stock(d2,p1,450) deleted.
stock(d3,p1,500) deleted.
stock(d3,p4,220) deleted.
```

Time taken: 1.50001s

e.g.14 Get supplier name and status from suppliers having status greater than 20 and who are in Paris.

```
select [sname,status]
from   [supplier]
where  [status > 20, city = paris].
```

sname	status
blake	30

Time taken: 1.85s

e.g.15 Get supplier number and supplier name from supplier who are in Paris or who have status greater than 20.

```
        select [sno,sname]
        from   [supplier]
        where  [city = paris]
union
        select [sno,sname]
        from   [supplier]
        where  [status > 20].
```

sno	sname
s2	jones
s3	blake
s5	adams

Time taken: 3.23334s

e.g.16 Get part number whose weight is greater than 18, or located in Paris, or supplied by Jones.

```
        select [pno]
        from   [part]
        where  [wt > 18]
union
        select [pno]
        from   [part]
        where  [city = paris]
union
        select [pno]
        from   [shipment]
        where  [[sno] in
                     select [sno]
```

```
                                      from   [supplier]
                                      where  [sname = jones]].
```

```
-----------
 pno
-----------
 p2
 p5
 p6
 p1

Time taken: 3.63334s
```

e.g.17 Get supplier numbers for suppliers who supply both part p1 and p2.

```
select [sno]
from   [shipment]
where  [pno=p1,
        [sno] in
              select [sno]
              from   [shipment]
              where  [pno = p2]].
```

```
------------
 sno
------------
 s1
 s2

Time taken: 1.98334s
```

e.g.18 Get supplier names and status for suppliers who live in London and supply at least one red part.

```
select [sname,status]
from   [supplier]
where  [city = london,
        [sno] in
              select [sno]
              from   [shipment]
              where  [[pno] in
                              select [pno]
                              from   [part]
                              where  [color = red]]].
```

```
-----------------------
 sname      status
-----------------------
 smith      20
 clark      20
```

Time taken: 3.81668s

e.g.19 Get supplier numbers for suppliers not currently supplying any parts.

```
select [sno]
from   [supplier]
where  [[sno] not_in
                      select [sno]
                      from   [shipment]].
```

```
-----------
 sno
-----------
 s5
```

Time taken: 1.41667s

e.g.20 List all the suppliers' names and locations for supplier whose status is greater than Smith's.

```
select [sname,city]
from   [supplier]
where  [[status] >
                    select [status]
                    from   [supplier]
                    where  [sname = smith]].
```

```
-------------------
 sname    city
-------------------
 blake    paris
 adams    athens
```

Time taken: 2.65001s

e.g.21 Get supplier names and status for suppliers who do not supply part p2.

```
select [sname,status]
from   [supplier]
where  [[sno] not_in
                      select [sno]
                      from   [shipment]
                      where  [pno = p2]].
```

```
-------------------
 sname    status
-------------------
 adams    30
```

```
Time taken: 2.50002s
```

e.g.22 Collect suppliers' names, part names and quantities supplied for suppliers in London.

```
select [sname,pname,qty]
from   [supplier,shipment,part]
where  [supplier@city = london,
        supplier@sno  = shipment@sno,
        shipment@pno  = part@pno].

---------------------------------
 sname      pname      qty
---------------------------------
 smith      nut        300
 smith      bolt       200
 smith      screw      400
 smith      screw      200
 smith      cam        100
 smith      cog        100
 clark      bolt       200
 clark      screw      300
 clark      cam        400

Time taken: 6.03336s
```

e.g.23 How many suppliers are there in London?

```
select [cnt(sno)]
from   [supplier]
where  [city = london].

-----------
 cnt(sno)
-----------
 2

Time taken: 1.40002s
```

e.g.24 How many people earn more than $8000 in department d1?

```
select [cnt(ename)]
from   [employee]
where  [salary > 8000, dno = d1].

------------
 cnt(ename)
------------
 2
```

```
Time taken: 1.68336s
```

e.g.25 Find the names of employees who are under 35 years of age and are paid more than the average salary for all employees.

```
select [ename]
from   [employee]
where  [age < 35,
        [salary] >
                   select [avg(salary)]
                   from   [employee]].

-------------
 ename
-------------
 john
 henry

Time taken: 2.06669s
```

e.g.26 Find the employees names and their managers names for employees who are under 30 years of age, work in London, and are paid more than the average salary for all employees with age over 40.

```
select [ename,manager]
from   [employee,department]
where  [employee@dno = department@dno,
        city = london,
        [ename] in
                  select [ename]
                  from   [employee]
                  where  [age < 30,
                          [salary] >
                                     select [avg(salary)]
                                     from   [employee]
                                     where  [age > 40]]].

----------------------
 ename    manager
----------------------
 henry    smith

Time taken: 4.11667s
```

e.g.27 How many people earn less than $8700 in each department.

```
select     [dno,cnt(ename)]
```

```
from        [employee]
grouped_by  [dno]
where       [salary < 8700].
```

dno	cnt(ename)
d1	2
d2	2
d3	1

Time taken: 2.78336s

e.g.28 Double the salaries of all employees in the department "d1".

```
update [employee]
set    [salary = 2*salary]
where  [dno = d1].
```

```
employee(john,34,8600,d1) : updated to :
                      employee(john,34,17200,d1)
employee(henry,29,12300,d1) : updated to :
                      employee(henry,29,24600,d1)
employee(martin,45,7500,d1) :  updated to :
                      employee(martin,45,15000,d1)
```

Time taken: 2.80004s

e.g.29 Delete parts supplied by Smith from table "part" whose color is red and weight is less than 15.

```
delete [part]
where  [color = red,
        wt < 15,
        [pno] in
               select [pno]
               from   [shipment]
               where  [[sno] in
                              select [sno]
                              from   [supplier]
                              where  [sname = smith]]].
```

```
part(p1,nut,red,12,london) deleted.
part(p4,screw,red,14,london) deleted.
```

Time taken: 3.53339s

e.g.30 Set the quantity to zero for all suppliers in London.

```
update [shipment]
set    [qty = 0]
where  [[sno] in
                 select [sno]
                 from   [supplier]
                 where  [city = london]].
```

```
shipment(s1,p1,300) deleted.
shipment(s1,p2,200) deleted.
shipment(s1,p3,400) deleted.
shipment(s1,p4,200) deleted.
shipment(s1,p5,100) deleted.
shipment(s1,p6,100) deleted.
shipment(s4,p2,200) deleted.
shipment(s4,p4,300) deleted.
shipment(s4,p5,400) deleted.

        Time taken: 5.91672s
```

e.g.31 Calculate the average salary at the department "d2".

```
select [avg(salary)]
from   [employee]
where  [dno = d2].
```

```
----------------
 avg(salary)
----------------
 7400

Time taken: 1.38336s
```

e.g.32 Find the employee names and ages for employees at "d1" department who earn more than the average salary of those who are in the same department with age over 30.

```
select [ename,age]
from   [employee]
where  [dno = d1,
        [salary] >
                   select [avg(salary)]
                   from   [employee]
                   where  [age > 30, dno = d1]].
```

```
--------------------
 ename      age
--------------------
 john       34
 henry      29
```

Time taken: 3.11667s

e.g.33 Find the maximum salary for young employees aged under 35 at each department.

```
select      [dno,max(salary)]
from        [employee]
grouped_by  [dno]
where       [age < 35].
```

dno	max(salary)
d1	12300
d2	8500
d3	7300

Time taken: 2.90002s

e.g.34 Find the department and its location, where there is at least one part the quantity of which happens to be the total number of the same part supplied by all all suppliers.

```
select [dno,city]
from   [department]
where  [[dno] in
                select [dno]
                from   [stock]
                where  [[pno,qty] in
                                      select [pno,sum(qty)]
                                      from   [shipment]
                                      grouped_by [pno]]].
```

dno	city
d1	london

Time taken: 3.85003s

e.g.35 Find average salary and number of employees in department d1.

```
select [avg(salary),cnt(ename)]
from   [employee]
where  [dno=d1].
```

avg(salary)	cnt(ename)

```
 9466.66     3

Time taken: 2.26672s
```

e.g.36 Get the table for part.

```
table [part].

     part
---------------------------------------------
 pno       pname      color      wt      city
---------------------------------------------

Time taken: 1.75005s
```

e.g.37 Create a new table by name emp_data having attributes name, age, wt, salary, dno

```
create [emp_data] : [name,age,wt,salary,dno].
```

APPENDIX 3

SYNTAX OF THE QUERY LANGUAGE

```
<a task written in EL> ::=
            <retrieval operation> | <update operation>
            <insert operation>    | <delete operation>

<retrieval operation> ::= <select-blocks>

<update operation> ::=
    UPDATE <relation> SET <attribute> = <arithmetic expression> |
    UPDATE <relation> SET <attribute> = <arithmetic expression>
                                             WHERE <conditions>

<insert operation> ::= INSERT_INTO <relation> : <constants>

<delete operation> ::= DELETE <relation> |
                  DELETE <relation> WHERE <conditions>

<select-blocks> ::= <select-block> |
               <select-block> UNION <select-blocks>

<select-block> ::=
    SELECT <target list> FROM <relations> |
    SELECT <target list> FROM <relations> WHERE <conditions> |
    SELECT <target list> FROM <relation> GROUPED_BY <attribute> |
    SELECT <target list> FROM <relation> GROUPED_BY <attribute>
             HAVING <function> <comparison operator> <real number>|

    SELECT <target list> FROM <relation> GROUPED_BY <attribute>
                                             WHERE <condition>

<target list> ::= <target> | <target>, <target list>

<target> ::= <attribute> | <relation>@<attribute> | <function>

<conditions> ::= <condition> | <condition>, <conditions>

<condition> ::= <simple condition> | <chaining condition> |
           <join condition>   | <set requirement>

<simple condition> ::=
    <attribute> <comparison operator> <constant> |
    <relation>@<attribute> <comparison operator> <constant>

<chaining condition> ::=
    <attribute> <chaining operator> <select-block>

<join condition> ::=
    <relation>@<attribute> <comparison operator>
                                 <relation>@<attribute>
```

```
<set requirement> ::=
    (<attributes>) <chaining operator> <select-block>

<arithmetic expression> ::= <constant> |
            <something> <arithmetic operator> <something>

<chaining operator> ::= IN | = | NOT_IN | > | < | >= | =< | \=

<comparison operator> ::=  < | > | =< | >= | = | \=

<arithmetic operator> ::=  + | - | * | /

<function> ::= <built-in function>(<attribute>)

<built-in function> ::=  CNT | MAX | MIN | SUM | AVG

<something> ::= <real number> | <attribute>

<relations> ::= <relation> | <relation>, <relations>

<relation> ::= <relation name>

<attributes> ::= <attribute> | <attribute>, <attributes>

<attribute> ::= <attribute name>

<constants> ::= <constant> | <constant>, <constants>

<constant> ::= <atom> | <integer>
```

APPENDIX 4

INTERNAL REPRESENTATION OF THE SAMPLE DATABASE

```
supplier(s1,smith,20,london).
supplier(s2,jones,10,paris).
supplier(s3,blake,30,paris).
supplier(s4,clark,20,london).
supplier(s5,adams,30,athens).

shipment(s1,p1,300).
shipment(s1,p2,200).
shipment(s1,p3,400).
shipment(s1,p4,200).
shipment(s1,p5,100).
shipment(s1,p6,100).
shipment(s2,p1,300).
shipment(s2,p2,400).
shipment(s3,p2,200).
shipment(s4,p2,200).
shipment(s4,p4,300).
shipment(s4,p5,400).

part(p1,nut,red,12,london).
part(p2,bolt,green,17,paris).
part(p3,screw,blue,17,rome).
part(p4,screw,red,14,london).
part(p5,cam,blue,12,paris).
part(p6,cog,red,19,london).

stock(d1,p1,600).
stock(d2,p2,350).
stock(d2,p1,450).
stock(d3,p1,500).
stock(d3,p4,220).

department(d1,london,smith).
department(d2,perth,long).
department(d3,london,lee).

employee(john,34,8600,d1).
employee(morgan,24,6300,d2).
employee(lewis,42,9000,d3).
employee(long,34,8500,d2).
employee(henry,29,12300,d1).
employee(thomas,31,7300,d3).
employee(martin,45,7500,d1).
```

INTERNAL REPRESENTATION OF RELATIONAL SCHEMA AND INTEGRITY CONSTRAINTS

```
relation(supplier(sno,sname,status,city)).

relation(shipment(sno,pno,qty)).

relation(part(pno,pname,color,wt,city)).

relation(stock(dno,pno,qty)).

relation(department(dno,city,manager)).

relation(employee(ename,age,salary,dno)).


integrity(status,A,[A>=10,A=<1000,integer(A)]) :- !.

integrity(qty,A,[A>=0,A=<100000,integer(A)]) :- !.

integrity(wt,A,[A>0,A=<1000]) :- !.

integrity(age,A,[A>18,A=<62]) :- !.

integrity(salary,A,[A>=1000,A=<50000]) :- !.
```

APPENDIX 5

```
/*********************************************/
/***** SOURCE CODE OF THE DATABASE PROGRAM *****/
/*********************************************/

/* CONSULTATION OF THE DATABASE, RELATIONAL SCHEMA */
/*                AND INTEGRITY FILES               */

:- [database,relation,integrity].

/*              OPERATOR DECLARATION PART                 */
/* TREATS ALL KEY-WORDS OF THE QUERY LANGUAGE AS OPERATORS */
/*  WHICH SIMPLIFIES SYNTAX ANALYSIS AND PARSING OF QUERY  */

:- op(40,xfx,in), op(40,xfx,not_in), op(40,xfx,'\='),
   op(40,xfx,@), op(185,yfx,having),
   op(190,yfx,where), op(192,yfx,grouped_by),
   op(195,yfx,from), op(195,yfx,set), op(195,yfx,':'),
   op(200,fx,select), op(200,fx,update),
   op(200,fx,insert_into), op(200,fx,delete),
   op(215,xfy,union), op(200,fx,create),
   op(200,fx,table), op(215,fx,'{'), op(200,xf,'}').

/*          THE QUERY READING AND ANSWERING PART            */
/* READS IN THE QUERY, CREATES VARIABLES FOR THE ATTRIBUTES */
/*   AND FINALLY FINDS THE ANSWERS TO THE TUPLES CREATED    */

do :- nl, write('Output stream Screen or File (s./f.) ? '),
      read(Mode), {nl, nl, write(' QUERY :'),
      nl, write('          '), read(S),
      (((S = ex; S = exit), exit) ; true),
      arg(1,S,X), ext(X,[H|T]), length(H,Num),
      ( (H =[*])
      ; (S = update(_))
      ; (S = delete(_))
      ; (S = insert_into(_))
      ; (S = create(_), L1 = [])
      ; (S = table(_), L1 = [])
      ;
      (for_write(H,Tem1), nl, writeln(Num), nl, write_attr(Tem1),

      nl, writeln(Num), create_var(H,Xs), extract(Xs,L1), nl))},

      {((Mode = s) ; (Mode = f, tell(file), nl, writeln(Num), nl,
       write_attr(Tem1), nl, writeln(Num), nl))},
      {get_sub_qs(S,Xs,!,Ans),
      ((H = [*], Ans = [B|T3], B =.. [_|L1]) ; H \= [*])},
      calling1(Ans), {write_attr(L1), nl}, fail.
do :- retract(ffound(N)), fail.
do :- told, do.
```

```
/*************************************************************/
/*************** QUERY PROCESSING PART *******************/
/*************************************************************/

/*************************************************************/
/*** (1) SELECT <target list> FROM <relation> GROUPED_BY ***/
/************** <attribute> WHERE <conditions> *************/

get_sub_qs(select(from(Attrs,grouped_by([Name],where([Grp_atr],
Conditions)))),Xs,Hash,Ans) :-
   !, {relation(K), K =.. [Name|L5], length(L5,Num2),
   get_skel(Num2,B), Tempo1 =.. [Name|B],
   simplify_cols(Attrs,Cols), pick_up(Num3,L5,[Grp_atr]),
   pick_up(Num3,B,[Var]), get_only3(Xs,Len3),
   length(Len3,Len3_num), get_only2(Xs,Len2),
   ((Len3_num > 1, Tem is  Len3_num - 1,
     get_set_skels(Num2,Tem,Skel_lis),
     unify(Num3,Skel_lis,[Var]), Len3 = [H3|T3],
     simplify(Attrs,Attr_lis,Aggr_lis),
     convert(Aggr_lis,[NN|New_lis]), Aggr_lis = [Head|Tail],
     form(New_lis,Skel_lis,L5,Name,Set), extract(Xs,L1),
     pick_up(Nu,L5,Attr_lis), pick_up(Num_list,Attrs,Attr_lis),
     pick_up(Num_list,L1,Tempo),  unify(Nu,Skel_lis,Tempo),
     pick_up(Nemo,Attrs,Aggr_lis), pick_up(Nemo,L1,[_|Temp2]),
     unifier(Tail,Set,Temp2), H3 = [_,Colo,Grp],
     append(Attr_lis,[Colo],Cols4), append(Len2,[H3],Xs2))
   ;
    (Xs2 = Xs, Cols4 = Cols)),
   ((member(Grp_atr,Cols4), extract(Xs2,L6),
    pick_up(Num1,Cols4,[Grp_atr]), pick_up(Num1,L6,[Var]),
    Xs1 = Xs2, Cols1 = Cols4)
   ;
    (append([Grp_atr],Cols4,Cols1),
     append([[Var,Grp_atr]],Xs,Xs1))),
  Ans2 =.. [group,Var,Tempo1,Var],
  break_off(Cols1,Xs1,Conditions,S,V,F,FF),
  differ([Name],S,V,F,FF,R), one_by_one(R,Ans1),
  ((Hash = !, Ans3 = Ans1)
  ;
   (Ans1 = [H|T], H1 =.. [not,H], Ans3 = [H1|T])),
  append([Ans2],Ans3,Ans4),
  ((Len3_num > 1, append(Set,Ans4,Ans))
   ;
   (Ans = Ans4))}.

/**************************************************/
/*** (2) SELECT <target list> FROM <relations> ***/
/************** WHERE <conditions> ***************/

get_sub_qs(select(from(Attrs,where(Relations,Conditions))),
Xs,Hash,Ans) :-
```

```
   length(Relations,Numb), simplify_cols(Attrs,Cols),
   ((Numb = 1, Relations = [Name|List],
     Conditions = [Hea|Tai], Hea =.. [Oper,Hea1,Tail],
           relation(K), K =.. [Name|L5], length(L5,Num2),
     get_skel(Num2,B), Tempo1 =.. [Name|B], get_only3(Xs,Len3),
     length(Len3,Len3_num), get_only2(Xs,Len2),
     ((Len3_num > 1, Tem is  Len3_num - 1,
     get_set_skels(Num2,Tem,Skel_lis), pick_up(Num3,L5,[Hea1]),
     unify(Num3,Skel_lis,[Tail]), Len3 = [H3|T3],
     simplify(Attrs,Attr_lis,Aggr_lis),
     convert(Aggr_lis,[NN|New_lis]), Aggr_lis = [Head|Tail],
     form(New_lis,Skel_lis,L5,Name,Set), extract(Xs,L1),
     pick_up(Nu,L5,Attr_lis), pick_up(Num_list,Attrs,Attr_lis),
     pick_up(Num_list,L1,Tempo),  unify(Nu,Skel_lis,Tempo),
     pick_up(Nemo,Attrs,Aggr_lis), pick_up(Nemo,L1,[_|Temp2]),
     unifier(Tail,Set,Temp2), H3 = [_,Colo,_],
     append(Attr_lis,[Colo],Cols4),
     create_var(Attr_lis,Tem_lis), append(Tem_lis,[H3],Xs2))
   ;
    (Xs2 = Xs, Cols4 = Cols)))
   ;
   (Len3_num = 0, Cols4 = Cols, Xs2 = Xs)),
   {break_off(Cols4,Xs2,Conditions,S,V,F,FF),
   differ(Relations,S,V,F,FF,R), one_by_one(R,Ans1),
   ((Hash = !, Ans1 = Ans4)
   ;
    (Ans1 = [H|T], H1 =.. [not,H], Ans4 = [H1|T])),
   ((Len3_num > 1, append(Set,Ans4,Ans))
    ;
    (Ans = Ans4))}.

/***********************************************************/
/*** (3) SELECT <target list> FROM <relation> GROUPED_BY ***/
/************** <attribute> HAVING <function> *************/

get_sub_qs(select(from(Attrs,grouped_by([Name],having([Grp_atr],
[Fun])))),Xs,Hash,Ans) :-
        {Fun =.. [Comp_op,Function,Something], extract(Xs,L1),
  Function =.. [Aggrop,Fun_Attr], conf([Name]),
  relation(K), K =.. [Name|L], pick_up(Fun_Num,L,[Fun_Attr]),
  length(L,Num), get_skel(Num,A), pick_up(ZX,L,[Grp_atr]),
  simplify(Attrs,Attr_lis,Aggr_lis), length(Aggr_lis,Te)},
  (({member(Grp_atr,L), member(Fun_attr,L), Te \= 0,
  pick_up(ZX,A,[Variable]), pick_up(Num_list,Attrs,Attr_lis),
  convert(Aggr_lis,New_lis), length(Aggr_lis,Aggr_lis_len),
  get_set_skels(Num,Aggr_lis_len,Skel_lis),
  unify(ZX,Skel_lis,[Variable]), pick_up(Nu,L,Attr_lis),
  form(New_lis,Skel_lis,L,Name,Set),
  pick_up(Num_list,L1,Tempo), unify(Nu,Skel_lis,Tempo),
  pick_up(Nemo,Attrs,Aggr_lis), pick_up(Nemo,L1,Temp2),
  chec_attr(Attr_lis,L), unifier(Aggr_lis,Set,Temp2),
```

```
  A1 =..[Name|A], get_skel(Num,C), pick_up(ZX,C,[Variable]),
  B1 =.. [group,Variable,A1,Variable], get_skel(Num,B),
  pick_up(ZX,B,[Variable]), pick_up(Fun_Num,C,[Func_Attr]),
  A2 =.. [Name|B], A3 =.. [Name|C],
          A4 =.. [Aggrop,Func_Attr,A3,Fun_ans],
  C1 =.. [Comp_op,Fun_ans,Something],
  append([B1|Set],[A4,C1],Ans1)})
  ;
  ({Te=0, pick_up(Number,L,Attrs), pick_up(Number,A,L1),
  chec_attr(Attrs,L), get_skel(Num,B), get_skel(Num,C),
  pick_up(ZX,A,[Variable]), pick_up(ZX,B,[Variable]),
  pick_up(Number,B,L1), pick_up(Number,C,L1),
  A1 =.. [Name|A], A2 =.. [Name|B], pick_up(ZX,C,[Variable]),
  B1 =.. [group,Variable,A1,Variable], A3 =.. [Name|C],
  A4 =.. [Aggrop,Func_Attr,A3,Fun_ans],
  pick_up(Fun_Num,C,[Func_Attr]),
  C1 =.. [Comp_op,Fun_ans,Something],  Ans1 = [B1,A2,A4,C1]})),
  ((Hash = !, Ans = Ans1)
  ;
   (Ans1 = [H|T], H1 =.. [not,H], Ans = [H1|T])).

/**********************************************/
/*** (4) SELECT <target list> FROM <relation> ***/
/************ GROUPED_BY <attribute> ************/

get_sub_qs(select(from(Attrs,grouped_by([Name],[Grp_atr]))),
Xs,Hash,Ans) :-
  {conf([Name]), relation(K), K =.. [Name|L], extract(Xs,L1),
  length(L,Num), get_skel(Num,A), pick_up(ZX,L,[Grp_atr]),
  simplify(Attrs,Attr_lis,Aggr_lis), length(Aggr_lis,Te)},
  (({ Te \= 0, member(Grp_atr,L), pick_up(ZX,A,[Variable]),
  pick_up(Num_list,Attrs,Attr_lis), pick_up(Nu,L,Attr_lis),
  convert(Aggr_lis,New_lis), length(Aggr_lis,Aggr_lis_len),
  get_set_skels(Num,Aggr_lis_len,Skel_lis),
  pick_up(Num_list,L1,Tempo), unify(Nu,Skel_lis,Tempo),
  unify(ZX,Skel_lis,[Variable]),
  form(New_lis,Skel_lis,L,Name,Set),
  pick_up(Nemo,Attrs,Aggr_lis), pick_up(Nemo,L1,Temp2),
  unifier(Aggr_lis,Set,Temp2), chec_attr(Attr_lis,L),
  A1 =..[Name|A], B1 =.. [group,Variable,A1,Variable],
  get_skel(Num,B), pick_up(ZX,B,[Variable]),
          Ans1 = [B1|Set]})
  ;
  ({Te = 0, pick_up(Number,L,Attrs), pick_up(Number,A,L1),
  chec_attr(Attrs,L), pick_up(ZX,L,[Grp_atr]), get_skel(Num,B),
  pick_up(ZX,A,[Variable]), pick_up(ZX,B,[Variable]),
  A1 =.. [Name|A], A2 =.. [Name|B], pick_up(Number,B,L1),
  B1 =.. [group,Variable,A1,Variable], Ans1 = [B1,A2]})),
  ((Hash = !, Ans = Ans1)
  ;
   (Ans1 = [H|T], H1 =.. [not,H], Ans = [H1|T])).
```

```
/**********************************************/
/*** (5) SELECT <attributes> FROM <relation> ***/
/**********************************************/

get_sub_qs(select(from(Attrs,[Name])),Xs,Hash,Ans5) :- !,
  {conf([Name]), relation(K), K =.. [Name|L],
  length(L,Num), extract(Xs,L1)},
  (({Attrs = '*', get_skel(Num,An),
  nl, writeln(Num), nl, write_attr(L), nl, writeln(Num), nl,
  Ans1 =.. [Name|An], Ans = [Ans1]})
  ;
  ({simplify(Attrs,Attr_lis,Aggr_lis),
  length(Aggr_lis,Te), Te \= 0,
  pick_up(Num_list,Attrs,Attr_lis), pick_up(Nu,L,Attr_lis),
  convert(Aggr_lis,New_lis), length(Aggr_lis,Aggr_lis_len),
  get_set_skels(Num,Aggr_lis_len,Skel_lis),
  pick_up(Num_list,L1,Tempo), pick_up(Nemo,Attrs,Aggr_lis),
  pick_up(Nemo,L1,Temp2), unify(Nu,Skel_lis,Tempo),
  form(New_lis,Skel_lis,L,Name,Ans),
  unifier(Aggr_lis,Ans,Temp2), chec_attr(Attr_lis,L)})
  ;
  ({pick_up(Number,L,Attrs),
  chec_attr(Attrs,L), get_skel(Num,B), pick_up(Number,B,L1),
  A2 =.. [Name|B], Ans = [A2]})),
  ((Hash = !, Ans5 = Ans)
  ;
   (Ans = [H|T], H1 =.. [not,H], Ans5 = [H1|T])).

/**********************************************/
/*** (6) DELETE <relation> WHERE <conditions> ***/
/**********************************************/

get_sub_qs(delete(where([Name],Conditions)),Xs,Hash,[]) :- !,
  {relation(K), K =.. [Name|Lis], create_var(Lis,Lis1),
  extract(Lis1,Lis2), break_off(Lis,Lis1,Conditions,S,V,F,FF),
  differ([Name],S,V,F,FF,R), one_by_one(R,Ans),
  reverse(Ans,[H|T]), reverse(T,Tem1)}, calling1(Ans),
  find_all([Name|Lis2]).

/**********************************************/
/*********** (7) DELETE <relation> *************/
/**********************************************/

get_sub_qs(delete([Name]),Xs,Hash,[]) :-
  {conf([Name]), skeleton(Name,B)}, find_all([Name|B]).
get_sub_qs(delete(_),Xs,Hash,[]).

/*********************************************************/
/******* (8) UPDATE <relation> SET <attribute> = *********/
/****** <arithmetic expression> WHERE <conditions> *******/
```

```
get_sub_qs(update(set([Name],where([Attr=Exp],Conditions))),
 [[Var,Attr]],!,[]) :- !,
          {break_off([Attr],[[Var,Attr]],Conditions,S,V,F,FF),
  differ([Name],S,V,F,FF,R), one_by_one(R,Ans),
  substitute(Var,Exp,New_exp), reverse(Ans,[H|T]),
  reverse(T,Tem1)}, calling1(Tem1),
  updat(Var,H,New_exp,Attr).

/*********************************************/
/*** (9) UPDATE <relation> SET <attribute> = ***/
/********** <arithmetic expression> ************/

get_sub_qs(update(set([Name],[Attr=Exp])),Xs,!,[]) :- !,
  {conf([Name]), relation(K), K =..[Name|L], length(L,Num),
  chec_attr([Attr],L), !, get_skel(Num,A),
  pick_up(NUM,L,[Attr]), pick_up(NUM,A,[Var]),
  A1 =.. [Name|A], substitute(Var,Exp,New_exp)},
  updat(Var,A1,New_exp,Attr).

/*************************************************/
/*** (10)  INSERT_INTO <relation>  : <constants> ***/
/*************************************************/

get_sub_qs(insert_into(:([Name],L)),Xs,Hash,[]) :-
  {{confirm(Name,L), A1 =.. [Name|L], relation(K),
  not(A1), K =.. [Name|Lis]}, {int(Lis,L)},
  asserta(savable(A1)), assertz(A1),
  write('Fact:  '), write(A1), nl,
  write('    inserted into the database ')}, !.
get_sub_qs(insert_into(:([Name],L)),Xs,Hash,[]) :-
  !, nl, write('Such a fact already exists - not entered').

/*********************************************/
/*** (11) <select-block> UNION <select-block> ***/
/*********************************************/

get_sub_qs(union(A,B),Xs,Hash,Ans) :- !, {extract(Xs,L1),
  get_sub_qs(A,Xs,Hash,Ans1), get_sub_qs(B,Xs,Hash,Ans)},
  ((calling1(Ans1), {write_attr(L1), nl}, fail)
  ;
   true).

/****************************************************/
/*** (12) CREATE <relation-name> : <attribute-list> ***/
/****************************************************/

get_sub_qs(create(:([Name],Attrs)),[],_,[]) :-
  not(duplic(Name)), B =.. [Name|Attrs],
  asserta(relation(B)), asserta(rel(relation(B))),
  write(' Do you want '), write(Attrs), write(' to have any
```

```
   integrity constraints ? (y./n.) '), nl, read(Ans),
   create_integ(Ans,Attrs), al, nl,
   write('Have you finished creating relations ? (y./n.)'),
   nl, read(Ans1), do_correct(Ans1).
 get_sub_qs(create(:([Name],Attrs)),[],_,[]) :-
   nl, write(' Such a relation name already exists
   - enter another name ').

/*************************************************/
/********** (13) TABLE <relation-name> ***********/
/*************************************************/

 get_sub_qs(table([Name]),[],_,[]) :-
   relation(K), K =.. [Name|L], length(L,Num), nl,
   write(Name), nl, writeln(Num), nl,
   write_attr(L), nl, writeln(Num), nl.

/***********************************************************/
/*   MIDDLE LEVEL PREDICATES FOR QUERYS CONTAINING 'WHERE'   */
/***********************************************************/

 break_off(S,V,[],S,V,[],[]).
 break_off(S1,V1,[A|B],S,V,F,[A1|FF]) :-
   join_cond(S1,V1,A,S2,V2,A1), break_off(S2,V2,B,S,V,F,FF).
 break_off(S1,V1,[A|B],S,V,[A|F],FF) :-
   break_off(S1,V1,B,S,V,F,FF).

 differ([],_,_,_,_,[]).
 differ([N1|N2],S,V,F,FF,[(N1,S1,V1,F1,FF1)|R]) :-
   pick_up(N1,S,V,F,FF,S1,V1,F1,FF1,V2,F2,FF2),
   differ(N2,S,V2,F2,FF2,R).

 one_by_one([(N,S,V,F1,F2)|[]],Ans) :-
   get_tuples(N,S,V,F1,F2,A,Ans).
 one_by_one([(N,S,V,F1,F2)|B],Ans1) :-
   get_tuples(N,S,V,F1,F2,A,Ans),
   one_by_one(B,T), append(Ans,T,Ans1).

 get_tuples(N,S,V,C,F2,Sth,Ans) :- relation(K),
   K =.. [N|L], length(L,Num), get_skel(Num,A),
   break_down(S,V,C,Ss,Vs,Fs1,F21,Ans2),
   reform_f2(Ss,Vs,F2,Sss,Vss,F22), append(F21,F22,Fs2),
   sort(L,A,Sss,Vss), get_sth(L,A,Vss,Sth),
   unific(L,A,Fs1,Fs11), remove(Fs2,Fs22),
   asserta(sub_q(N,A,Fs11,Fs22,Vss,Sth)),
   get_querys(sub_q(N,A,Fs11,Fs22,Vss,Sth),Ans1),
   append(Ans2,Ans1,Ans).

break_down(S,V,[],S,V,[],[],[]) :- !.
break_down(S1,V1,[A|B],S2,V2,Fs1,Fs2,Ans) :- A=..[I,J,K],
  (((atom(K) ; number(K)), simplify_relatr([J],[J1]),
```

```
    cut_off1(S1,V1,I,J1,K,S,V,F1,F2,Ans1))
    ;
   cut_off(S1,V1,I,J,K,S,V,F1,F2,Ans1)),
  break_down(S,V,B,S2,V2,F12,F22,Ans2),
  (F1==!, Fs1=F12; Fs1=[F1|F12]),
  (F2==!, Fs2=F22; Fs2=[F2|F22]),
  append(Ans2,Ans1,Ans).

cut_off(S,V,in,J,K,S1,V1,!,!,Ans) :-
  simplify_cols(J,L), subst(S,V,L,Xs,S1,V1), arg(1,K,Temp),
  ext(Temp,[Atr|_]), create_var(Atr,Attr1),
  get_only2(Xs,Tem2), set_diff(Attr1,Tem2,Tem3),
  get_xs(Tem3,Xs,Xs1), get_sub_qs(K,Xs1,!,Ans),
  extract(Xs,Tempo), extract(Xs1,Tempo).
cut_off(S,V,=,J,K,S1,V1,!,!,Ans) :-
  cut_off(S,V,in,J,K,S1,V1,!,!,Ans).
cut_off(S,V,not_in,J,K,S1,V1,!,!,Ans) :-
  simplify_cols(J,L), subst(S,V,L,Xs,S1,V1), arg(1,K,Temp),
  ext(Temp,[Atr|_]), create_var(Atr,Attr1),
  get_only2(Xs,Tem2), set_diff(Attr1,Tem2,Tem3),
  get_xs(Tem3,Xs,Xs1), get_sub_qs(K,Xs1,not,Ans),
  extract(Xs,Tempo), extract(Xs1,Tempo).
cut_off(S1,V1,I,J,K,S,V,!,F2,Ans) :- !, simplify_cols(J,J1),
  subst(S1,V1,J1,[Xs],S,V), arg(1,K,R), ext(R,[H|T]),
  H = [H1|T1],
  ((atom(H1), Xs1 = [[Y2|J1]])
  ;
   (H1 =.. [Op,_], append(J1,[Op],Atr1), Xs1 = [[Y2|Atr1]])),
  get_sub_qs(K,Xs1,!,Ans),
  extract(Xs,[Y1]), F2=..[I,Y1,Y2].
cut_off1(S1,V1,I,J,K,S,V,F1,!,[]) :-
  substitute(S1,V1,I,J,K,S,V), F1 =.. [I,J,K].

get_xs([],A,A).
get_xs([H|T],Xs,A) :- H = [Var,Attr,Op], extract(Xs,Tem1),
  extract_attr(Xs,Tem2), pick_up(Num,Tem2,[Attr]),
  pick_up(Num,Tem1,[Var]),
  subst([Var,Attr],Xs,[Var,Attr,Op],Ans), get_xs(T,Ans,A).

get_querys(sub_q(N,A,F,FF,Var,Sth),Temp1) :-
  {Tem =.. [N|A], append(F,FF,Conds),
  ((length(Sth,3), Sth = [Op,Atr,Ans], append([Tem],Conds,Tum),
   append([Atr|[Tum]],[Ans],Tum1),  Temp2=.. [Op|Tum1],
   append([],[Temp2],Temp1))
   ; (Temp3 = Tem, append([Temp3],Conds,Temp1)))}.

modify([],[]).
modify([H1|T1],[H2|T2]) :- H1 =.. [Comp_op,Relatr1,Relatr2],
  Relatr1 =.. [@,Rel1,Atr1], Relatr2 =.. [@,Rel2,Atr2],
  H2 =.. [Comp_op,Atr1,Atr2], modify(T1,T2).
```

```
tab(N,H) :- relation(K), K =.. [N|L],
  length(L,Num), get_skel(Num,H).

simplify_relatr([],[]).
simplify_relatr([H|T],[H|T1]) :- atom(H), simplify_relatr(T,T1).
simplify_relatr([H|T],[H1|T1]) :- not atom(H), H =..[@,_,H1],
  simplify_relatr(T,T1).

subst(S,V,[],[],S,V).
subst(S,V,[H|T],Xs,S1,V1) :- member(H,S), extract(V,Vtem),
  extract_attr(V,Vatr), pick_up(Num,Vatr,[H]),
  pick_up(Num,Vtem,[Ans]), subst(S,V,T,Xs1,S1,V1),
  append([[Ans,H]],Xs1,Xs).
subst(S,V,[H|T],[[B,H]|Xs1],S2,V2) :-
  append([H],S,S1), append([[B,H]],V,V1),
  subst(S1,V1,T,Xs1,S2,V2).

substitute(S1,V1,I,J,K,S1,V1) :- member(J,S1).
substitute(S1,V1,I,J,K,S,V) :- not member(J,S1),
  append([J],S1,S), append([[M,J]],V1,V).

reform_f2(Ss,Vs,[],Ss,Vs,[]).
reform_f2(Ss,Vs,[H|T],Sss,Vss,[H1|T1]) :-
  {H =.. [Op,Relatr1,Relatr2], Relatr1 =.. [@,Rel1,Atr1]},
  Relatr2 =.. [@,Rel2,Atr2], H1 =.. [Op,Atr1,Atr2],
  ((member(Atr1,Ss), reform_f2(Ss,Vs,T,Sss,Vss,T1))
  ;(append([Atr1],Ss,Sss), append([[B,Atr1]],Vs,Vss),
    reform_f2(Sss,Vss,T,Ssp,Vsp,T1))).

sort(_,_,[],_).
sort(L,A,Sss,Vss) :- get_only2(Vss,Temp1), extract(Temp1,Temp2),
  extract_attr(Temp1,Temp3), pick_up(Num,L,Temp3),
  pick_up(Num,A,Temp2).

get_only2([],[]).
get_only2([H|T],[H|T1]) :- length(H,2), get_only2(T,T1).
get_only2([H|T],T1) :- not length(H,2), get_only2(T,T1).

get_sth(_,_,[],[]).
get_sth(L,A,[H|T],Sth) :- length(H,3), H = [Var,Attr,Op],
  pick_up(NU,L,[Attr]), pick_up(NU,A,[Ans]), Sth = [Op,Ans,Var].
get_sth(L,A,[H|T],Sth) :- not length(H,3), get_sth(L,A,T,Sth).

join_cond(S1,V1,A,S2,V2,A) :- {A =.. [Comp_Op,Rel1,Rel2],
  Rel1 =.. [Op,Rel_nam1,Attr1], Rel2 =.. [Op,Rel_nam2,Attr2]},
  Rel_nam1 \= Rel_nam2, get_s2v2(Attr1,S1,V1,S2,V2).

get_s2v2(Attr1,S1,V1,S1,V1) :- member(Attr1,S1).
get_s2v2(Attr1,S1,V1,S2,V2) :- not member(Attr1,S1),
  append([Attr1],S1,S2), append([[B,Attr1]],V1,V2).
```

```
pick_up(N1,S,V,F,FF,S1,V1,F1,FF1,V2,F2,FF2) :-
  relation(K), K =.. [N1|L],
  get_indef(N1,F,F1,L), get_relef(N1,FF,FF1),
  get_attr(L,S,S1), get_var(S1,L,V,V1), deal_wit(FF1,V,V2),
  set_diff(F,F1,F2), set_diff(FF,FF1,FF2).

deal_wit([],V,V).
deal_wit([H|T],V,V2) :- H =.. [=,_,_], deal_wit(T,V,V2).
deal_wit([H|T],V,V2) :- not H =.. [=,_,_],
  H =.. [Op,Relatr1,Relatr2], Relatr1 =.. [@,Rel1,Atr1],
  efface([A,Atr1],V,V_temp), deal_wit(T,V,V_temp),
  append([B,Atr1],V_temp,V2).

remove([],[]).
remove([H|T],T1) :- H =.. [=,_,_], remove(T,T1).
remove([H|T],[H|T1]) :- remove(T,T1).

unific(_,_,[],[]).
unific(L,A,[H|T],T1) :-
  H =.. [=,Atr1,Atr2], pick_up(Num,L,[Atr1]),
  pick_up(Num,A,[Atr2]), unific(L,A,T,T1).
unific(L,A,[H|T],[H1|T1]) :-
  not H =.. [=,_,_], H =..[Op,Atr1,Atr2],
  pick_up(Num,L,[Atr1]), pick_up(Num,A,[TY]),
  H1 =.. [Op,TY,Atr2], unific(L,A,T,T1).

set_diff([],_,[]).
set_diff([H|T],F,F2) :- member(H,F), set_diff(T,F,F2).
set_diff([H|T],F,[H|F2]) :- not member(H,F), set_diff(T,F,F2).

get_var([],_,_,[]).
get_var([H|T],L,V,V1) :- member(H,L),
  (member([A,H],V) ; member([A,H,B],V)), get_var(T,L,V,V2),
  ((nonvar(B), append([[A,H,B]],V2,V1))
  ;
   (append([[A,H]],V2,V1))).

get_attr([],_,[]).
get_attr([H|T],L,[H|T1]) :- member(H,L), get_attr(T,L,T1).
get_attr([H|T],L,T1) :- not member(H,L), get_attr(T,L,T1).

create_var([],[]).
create_var([H|T],[H1|T1]) :- atom(H),
  append([A],[H],H1), create_var(T,T1).
create_var([H|T],[H1|T1]) :- not atom(H), H =..[B,C],
  append([A],[C,B],H1), create_var(T,T1).

get_indef(_,[],[],_).
get_indef(N1,[H|T],[H|B],L) :- H =.. [Op,R1,R2],
  ((member(R1,L))
  ;
```

```
    (R1 =.. [@,N1,R12])), get_indef(N1,T,B,L).
 get_indef(N1,[H|T],[H|B],L) :- H =.. [Op,R1,R2],
   R2 = select(_), get_indef(N1,T,B,L).
 get_indef(N1,[H|T],Lis,L) :- get_indef(N1,T,Lis,L).

 get_relef(N1,[],[]).
 get_relef(N1,[H|T],[H|B]) :- {H =.. [Op,R1,R2]},
   R1 =.. [@,N1,N2], get_relef(N1,T,B).
 get_relef(N1,[H|T],Lis) :- get_relef(N1,T,Lis).


/*************************************************************/
/*********        LOWER LEVEL PREDICATES       **************/
/*************************************************************/

{X} :- call(X), !.

A\=B :- A=B, !, fail.
A\=B.

get_only3([],[]).
get_only3([H|T],[H|T1]) :- length(H,3), get_only3(T,T1).
get_only3([H|T],L) :- get_only3(T,L).

create_integ(n,_) :- al, assertz(integrity(_,_,[])).
create_integ(y,[]) :- assertz(integrity(_,_,[])),
  nl, write(' Integrity checks over ').
create_integ(y,[H|T]) :- nl, write('Integrity constraints for: '),

  write(H), nl, nl, write(' give list of conditions :'), nl,
  read(S), replace(Var,H,S,Lis),
  assertz(:-(integrity(H,Var,Lis),!)),
  ((retract(:-(integrity(_,_,[]),!))) ; true),
  assertz(integra(:-(integrity(H,Var,Lis),!))), create_integ(y,T).


al :- retract(:-(integrity(_,_,[]),!)), fail.
al :- retract(integrity(_,_,[])), fail.
al :- assertz(integrity(_,_,[])).

convert([],[]).
convert([H|T],Aggregate_list) :- H =.. L, convert(T,New_aggr_lis),

  append([L],New_aggr_lis,Aggregate_list).

get_set_skels(Num,0,[]).
get_set_skels(Num,Aggr_lis_len,Ans) :- A is Aggr_lis_len - 1,
  get_skel(Num,A1), get_set_skels(Num,A,A2), append([A1],A2,Ans).

form([],[],_,_,[]).
form([H1|T1],[H2|T2],L,Name,Ans) :- H1 = [H|T], pick_up(Num,L,T),
  pick_up(Num,H2,[A1]), A2 =.. [Name|H2], A3 =.. [H,A1,A2,A4],
  form(T1,T2,L,Name,A5), append([A3],A5,Ans).
```

```
unify(_,[],_).
unify(ZX,[H|T],Var) :- pick_up(ZX,H,Var), unify(ZX,T,Var).

unifier([],_,[]).
unifier([H|T],[H1|T1],[H2|T2]) :- H1 =.. [_,_,_,H2],
  unifier(T,T1,T2).

calling1([]).
calling1([H|T]) :- length([H|T],Num), Num \= 1,
  H =.. [not,_], reverse([H|T],L), calling1(L).
calling1([H|T]) :- H =.. [Op,B1,B2,AN], (Op = max ; Op = min),
  H, B1 = AN, (calling1(B2) ; (not(is_list(B2)), B2)),
  !, calling1(T).
calling1([H|T]) :- H, calling1(T).

for_write([],[]).
for_write([H|T],[H|T1])  :- atom(H), for_write(T,T1).
for_write([H|T],[H1|T1]) :- not atom(H), H =.. [A,B],
  name(A,A1), name(B,B1), append(A1,[40|B1],Temp1),
  append(Temp1,[41],Temp), name(H1,Temp), for_write(T,T1).

skeleton(Name,A) :- relation(K), K =.. [Name|L], length(L,Num),
  get_skel(Num,A).

deal(L,Name,B) :- nl, !, A1 =..[Name|B],A1, write('       '),
  pick_up(L,B,L1), write_attr(L1), nl, fail.

deal_with(Num,A1,Exp) :- A1, A1 =..[H|T], replace(Num,T,Exp).

find_all(A) :- nl, nl, J =..A, !, J,
  retract(J), asserta(removable(J)),
  write(J), write('       deleted.'), nl, fail.

evaluate(Old,Exp,Exp) :- atom(Exp).

evaluate(Old,Exp,New) :- Exp =..[A,B,C], atom(B),
  Temp =..[A,Old,C], New is Temp.
evaluate(Old,Exp,New) :- Exp =..[A,B,C], atom(C),
  Temp =.. [A,B,Old], New is Temp.

subst(_,[],_,[]).
subst(X,[X|L],A,[A|L]).
subst(X,[Y|L],A,[Y|M]) :- subst(X,L,A,M).

pick_up([],_,[]).
pick_up([M|N],L,[X|Y]) :- mem(M,X,L), pick_up(N,L,Y).

mem(1,X,[X|_]).
mem(N,X,[_|Y]) :- mem(M,X,Y), N is M+1.

get_skel(0,[]).
get_skel(1,[H]) :- !.
```

```
get_skel(N,[H|T]) :- N1 is N-1, get_skel(N1,T).

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

write_attr([]).
write_attr([H|T]) :- write(H), name(H,L), length(L,N),
  N1 is 11 - N, tab(N1), write_attr(T).

exit :- my_save(saved,savable), my_save(deleted,removable), halt.
ex :- exit.

substitute(Var,Exp,Exp) :- atom(Exp).
substitute(Var,Exp,Exp) :- number(Exp).
substitute(Var,Exp,New_exp) :- Exp =.. [Op,T1,T2], atom(T1),
  New_exp =.. [Op,Var,T2].
substitute(Var,Exp,New_exp) :- Exp =.. [Op,T1,T2], atom(T2),
  New_exp =.. [Op,T1,Var].

conf([]) :- !.
conf([H|T]) :- relation(K), K =..[H|L], conf(T).
conf(_) :- !, nl, nl, write('Check relation name(s) '), fail.

check(L,Attrs) :- relation(K),
  K =.. [L|Lis], chec_attr(Attrs,Lis).
check(_,_) :- !, write('Check spelling of relation '), fail.

duplic(Name) :- relation(K), K =.. [Name|_].

do_correct(y) :- my_save(integrity,integra),
  my_save(relation,rel), nl, my_save(relation,relation),
  write(' Integrity(s) and relation(s) saved').
do_correct(n) :- do.

chec_attr([],Lis).
chec_attr([H|T],Lis) :- member(H,Lis), chec_attr(T,Lis).
chec_attr([H|T],Lis) :- !,
  write(' Attribute list is incorrect '), fail.

confirm(Name,Args) :- relation(K), K=..[Name|L], length(L,Nu1),
  length(Args,Nu2), Nu1 = Nu2.


/**********************************************************/
/******* SAVE ALL UPDATED CLAUSES AND NEW FACTS  **********/
/**********************************************************/

my_save(File,Y) :-
  tell(File), save_predicate(X,Y), fail.
my_save(_,_) :- told.
```

```
save_predicate(X,savable) :- savable(X), write(X),
  write('.'), tab(5), fail.
save_predicate(X,removable) :- removable(X), write(X),
  write('.'), tab(5), fail.
save_predicate(X,integra) :- integra(X), write(X),
  write('.'), tab(5), fail.
save_predicate(X,rel) :- rel(X), write(X),
  write('.'), tab(5), fail.
save_predicate(X,relation) :- relation(X),
  write(relation(X)), write('.'), tab(5), fail.
save_predicate(_,_).


/*******************************************************/
/*************  ALL AGGREGATION FUNCTIONS **************/
/*******************************************************/

cnt(X,G,_) :- asserta(found(mark)),
  (calling1(G)
  ; (not(is_list(G)), call(G))), asserta(found(X)),
  fail.
cnt(X,G,C) :- cnt_found(0,C), !.

sum(X,G,_) :- write_down(X,G), fail.
sum(X,G,S) :- sum_found(0,S), !.

avg(X,G,_) :- write_down(X,G), fail.
avg(X,G,A) :- avg_found(0,0,A), !.

max(X,G,_) :- write_down(X,G), fail.
max(X,G,M) :- max_found(0,M), !.

min(X,G,_) :- write_down(X,G), fail.
min(X,G,M) :- min_found(1.0e38,M), !.

cnt_found(I,J) :- getnext(X), K is I+1, cnt_found(K,J).
cnt_found(K,K).

sum_found(I,J) :- getnext(X), K is I+X, sum_found(K,J).
sum_found(K,K).

avg_found(I,J,K) :- getnext(X), M is I+1,
               N is J+X, avg_found(M,N,K).
avg_found(M,N,A) :- A is N/M, !.

max_found(I,J) :- getnext(X),
     (X>I, max_found(X,J); max_found(I,J)).
max_found(K,K).

min_found(I,J) :- getnext(X),
     (X<I, min_found(X,J); min_found(I,J)).
```

```
min_found(K,K).

write_down(X,G) :- asserta(found(mark)),
  (calling1(G)
  ;
   (not(is_list(G)), call(G))), asserta(found(X)).

all(Ans,Q,_) :- asserta(found(mark)),
  (calling1(Q) ; (not(is_list(G)), call(G))),
  one_of_them(Ans), fail.
all(Ans,Q,Set) :- collect_found([],Set), !.

one_of_them(Ans) :- found(Ans), !.
one_of_them(Ans) :- asserta(found(Ans)).

collect_found(S,Set) :- getnext(X), collect_found([X|S],Set).
collect_found(S,S).

getnext(X) :- retract(found(X)), !, X \== mark.

updat(X,G,Y,Attr) :- (calling1(G) ; (not(is_list(G)), call(G))),
  updated(X,G,Y,Attr), fail.

updated(X,G,Y,Attr) :- {find(Y,Y1)}, {integ(Attr,Y1)},
  replace(Y1,X,G,New), nl, retract(G),
  asserta(removable(G)), asserta(savable(New)),
  asserta(New), write(G),
  write('  :  updated to  :  '), write(New), !.

int([],[]).
int([H|T],[H1|T1]) :- integrity(H,H1,Lis),
  checking(Lis), int(T,T1).

integ(Attr,Y1) :- integrity(Attr,Y1,Lis), checking(Lis).

integrity(_,_,[]).

checking([]).
checking([H|T]) :- H, !, checking(T).
checking([H|T]) :- !, {nl, write(H),
  write(' :  violates integrity constraints'), nl,
  write('   No updation/insertion done for this tuple ')}, fail.

find(X,X) :- atomic(X).
find(E,R) :- R is E.

replace(New,Old,Old,New).
replace(New,Old,Val,Val) :- atomic(Val).
replace(New,Old,Val,NewVal) :-    functor(Val,Fn,N),
  functor(NewVal,Fn,N), subst_args(N,New,Old,Val,NewVal).
```

```
subst_args(0,_,_,_,_) :- !.
subst_args(N,New,Old,Val,NewVal) :- arg(N,Val,OldArg),
  arg(N,NewVal,NewArg), replace(New,Old,OldArg,NewArg),
  N1 is N-1, subst_args(N1,New,Old,Val,NewVal).

group(N,G,N) :- (calling1(G) ; (not(is_list(G)), call(G))),
  only(N).

only(N) :- not ffound(N), asserta(ffound(N)), !.

is_list([]).
is_list([_|_]).

writeln(0).
writeln(1) :- write('-----------').
writeln(N) :- N1 is N-1, writeln(1), writeln(N1).

simplify([],[],[]).
simplify([H|T],[H|Attr_lis],Aggr_lis) :- atom(H),
  simplify(T,Attr_lis,Aggr_lis).
simplify([H|T],Attr_lis,[H|Aggr_lis]) :- not atomic(H),
  not number(H), simplify(T,Attr_lis,Aggr_lis).

simplify_cols([],[]).
simplify_cols([H|T],[H|Attr_lis]) :- atom(H),
  simplify_cols(T,Attr_lis).
simplify_cols([H|T],[B|Attr_lis]) :- not atomic(H), not number(H),

  H =.. [_,B], simplify_cols(T,Attr_lis).


/**************************************************/
/*******       SOME USEFUL PREDICATES      ********/
/**************************************************/

append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).

extract(X,[]) :- atomic(X).
extract(X,[X]) :- var(X).
extract(T,L) :- arg(1,T,X), extract(X,L1), arg(2,T,Y),
  extract(Y,L2), append(L1,L2,L).

extract_attr(X,[]) :- var(X).
extract_attr([],[]).
extract_attr(X,[X]) :- atomic(X).
extract_attr(T,L) :- arg(1,T,X), extract_attr(X,L1),
  arg(2,T,Y), extract_attr(Y,L2), append(L1,L2,L).

ext(*,[[*]]).
ext(X,[]) :- atomic(X).
ext(X,[X]) :- is_list(X).
```

```
ext(T,L) :- {arg(1,T,X), ext(X,L1)},
  arg(2,T,Y), ext(Y,L2), append(L1,L2,L).
ext(T,L) :- arg(1,T,X), ext(X,L).

efface(A,[A|L],L) :- !.
efface(A,[B|L],[B|M]) :- efface(A,L,M).

reverse([],[]).
reverse([H|T],L) :- reverse(T,R), append(R,[H],L).
```

APPENDIX 6

BIBLIOGRAPHY AND REFERENCES

[1]. R.A.Kowalski, "Prolog as a Logic Programming Language", AICA Congress, 23-25 Sept 1981, Vol 2, pp - 1029-34.

[2]. J.W.Lloyd, "Foundations of Logic Programming", Symbolic Computation Series in A.I., Springer-Verlag, 1984.

[3]. W.F.Clocksin & C.S.Mellish, "Programming in Prolog", Springer International Students Edition, Springer-Verlag, Berlin, Heidelberg, 1986.

[4]. Zohar Manna, "Mathematical Theory of Computation, McGraw-Hill International Student Edition, McGraw-Hill Kogakusha Ltd., 1974.

[5]. R.A.Kowalski, "Logic for Data Description", in Logic and Databases, edited by H.Gallaire and J.Minker, Plenum Press, New York, 1978, pp- 73-103.

[6]. D.H.D.Warren, "Logic Programming and Compiler Writing", Software Practice & Experience, Vol 10, pp- 97-125, 1980.

[7]. Leon Sterling & Ehud Shapiro, "The Art Of Prolog: Advanced Programming Techniques", The MIT Press, Cambridge, Massachusetts, 1986.

[8]. R.Fagin, "Horn Clauses and Database Dependencies", Proc. ACM Sigact symposium on Theory of Computation, pp- 123-134, 1980.

[9]. Y.Sagiv, R.Fagin et al, "An Equivalence Between Relational Database Dependencies & a Fragment of Propositional Logic, JACM, Vol 28, pp- 435-453, 1981.

[10]. C.J.Date, "An Introduction to Database Systems", Addison-Wesley/Narosa Indian Student Edition, 1987.

[11]. J.D.Ullman, "Principles of Database Systems", Pitman Publishing Limited, 1983.

[12]. Deyi Li, "A Prolog database System", Research Studies Press Ltd., Letchworth, Hertfordshire, England, 1984.

[13]. R.N.Cuff, "HERCULES: Database Query Using Natural Language Fragments", Proc. of the Third British National Conference on Databases, 1984