Implementation of Prolog Database System

SHU-MIN KUO,* YUKIO KANEDA** and SADAO MAEKAWA**

A predicate logic programming language system that can be used both as a problem-solver and as a relational database system is implemented on NEC PC-9801F MS/DOS system in *C*. With this system, the relational database system functions are built in the Prolog language system. Therefore relational database query statements can be simply embedded in the Prolog program in the first order predicate logic formula. Interpreter approach, backtracking and unification are operated on IDB (Intensional DataBase) goals. While set operations are applied on EDB (Extensional DataBase) goals and system output predicates. Thus variables may be bound by a single term or a set of terms. At each node of a proof tree, the binding environments are placed on the variable stack for single terms and on the table stack for sets. The table stack contains some temporary binding tables in which each column represents a set of terms for a bound variable. The binding tables are generated at each EDB goal call and registered in the database dictionary. Once backtracking occurs the table stack is popped and the binding tables are dropped from the database dictionary. The variable stack resides in main memory while the table stack resides in secondary memory. In executing a goal statement the search is based on a simple left-to-right and depth-first strategy.

1. Introduction

It has been recognized that the relation between predicate logic and relational databases is very close [1, 2]. Current implementations of predicate logic language such as Prolog assume that all general rules and facts reside in primary memory. However, the memory capacity is limit. It is impossible to place a large volume of facts in main memory. On the other hand, current implementations of relational database system provide optimal operations for very large databases but lack inference ability. Therefore, connecting predicate logic language with relational database becomes an important theme. Two approaches to this interface have been discussed by Chakravarthy [3]—the compiled approach and the interpretive approach. In the compiled approach the deductive portion is separate from the database portion. One can apply general rules until all goal statements consist of relations which are base tables in the relational database then transmit to a RDBMS (Relational DataBase Management System) to obtain the answers. Implementations with a compiled approach have also been represented by *Li* [4] and by Yokota [5]. In the interpretive approach the set-oriented approach is introduced. The set operations are applied on each node of the search tree. Variables are bound by sets. The binding environments are placed on tables.

Both approaches above assume that general rules are not recursive and the objective focuses on database retrieval. Thus some efforts [5] must be made on recursive cases and on the combination of database query statements and a logic program.

In this paper a system which combines the Prolog language system and the relational database system is represented. This system is implemented on NEC PC-9801F MS/DOS system in *C*. The relational database functions are built in the Prolog language system. Therefore, relational database query statements can be elegantly embedded in the Prolog program in the first order predicate logic formula. By interpretive approach, backtracking and unification are operated on user defined IDB (intensional database) goals while set operations are applied on EDB (extensional database) goals and system output predicates.

2. The Architecture of the Prolog Database System

The architecture of the Prolog database system consists of four principal components—the Prolog interpreter, the Prolog-DBMS interface, the RDBMS and the database dictionary. (Fig. 1)

The Intensional Database contains general rules and exceptions in horn clauses. Once a goal statement is executed it will be loaded into main memory throughout the execution time.

The Extensional Database (abbreviated as database) contains only facts in tables. It always resides in the secondary memory. A table is loaded into main memory only when it is needed for current operation. Once the operation is finished the memory space for this table is immediately free for other operations.

*Division of System Science, Graduate School of Science and Technology, Kobe University, Rokkodai, Nada, Kobe 657 Japan.
**Department of Systems Engineering, Faculty of Engineering, Kobe University, Rokkodai, Nada, Kobe 657 Japan.

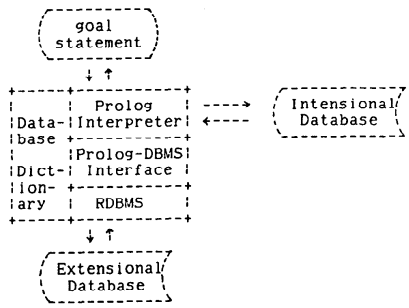


Fig. 1 The architecture of Prolog database system.

The Prolog interpreter performs the deduction of a goal statement and separates IDB goals from EDB goals. When an EDB goal or an output predicate is encountered it passes the control to the Prolog-DBMS interface.

The Prolog-DBMS interface generates set operation commands to the RDBMS and enables the RDBMS.

The RDBMS performs the set operation and produces the result in the table stack which is one of the binding environments of variables then returns control to the Prolog-DBMS interface. The Prolog-DBMS interface checks whether the result is empty then returns success or failure to the Prolog interpreter.

The Database Dictionary provides extensional database information and acts as a bridge among the three components above.

3. The Usage of the Prolog Database System

In addition to except, the well-known pure Prolog language, the relational database language is also represented in predicate logic. In this system we unify the data description language and data manipulation language. Some additional built-in predicates are necessary for organizing a database.

The predicate "db__init" is defined to create a new database. It requires a database name as its single argument.

db__init(database__name).

The predicate "invoke" enables the system to load the indicated database dictionary into main memory. The indicated database name is required as its single argument.

invoke(database__name).

The predicate "define" defines a table in the currently invoked database. Two arguments are required—a table name and the number of columns of the table. Differring from the conventional way it need not define attribute name, data type and data length for each column. Instead of the attribute name, the column number is used to recognize the data field by the set operation command. Furthermore the system classifies data types automatically so that a structure may be matched in pattern.

define(table__name, no.__of__columns).

The predicate "input" enables the system to accept data from a terminal in batch and store them to the indicated table. The indicated table name is required as its single argument.

input(table__name).

The predicate "drop" causes the indicated table to be erased from the current database. Its single argument is a table name which has been defined by the "define" predicate.

drop(table__name).

As to the database retrieval language, it is well-known that set operations can be represented in PROLOG easily [4]. The operator "union" is equivalent to the disjunction of goals. The "intersection," "Cartesian product" and "join" operators can be expressed by the conjunction of goals. The "projection" operator is equivalent to the single goal. The "selection" operator can be represented by an EDB goal with some logic expressions as its conditions. The "difference" operator is equivalent to the "not" predicate. For example, suppose p is a table then $p(X, Y)$ represents that table p has two columns. Both columns have their domains, say X and Y respectively. Thus the difference of table $p(X, Y)$ minus table $q(X, Y)$ can be represented as:

$$\text{ans}(X, Y) :- p(X, Y), \text{not}(q(X, Y)).$$

and table $r(X, Y, A, B)$ divided by table $s(A, B)$ may be represented as:

$$\text{ans}(X, Y) :- r(X, Y, \_, \_), \text{not}(r(X, Y, \_, \_), s(A, B), \text{not}(r(X, Y, A, B))).$$

To update a database, the predicate "insert" and the predicate "delete" are provided. The formats are

insert(assertion).
delete(assertion).

The single argument is an assertion which consists of an EDB predicate with its arguments. Both predicates can handle a set of tuples which are represented in the binding environment. Thus we can save the result in a new table, named $r(X, Y)$, after the join of table $p(X, Z)$ and table $q(Z, Y)$. The expression is shown below.

?— define(r, 2), $p(X, Z)$, $q(Z, Y)$, insert($r(X, Y)$).

Predicate logic can also be used to express integrity and security constraints. Both are not discussed at this stage. In the following example we will reveal how easy it is to use the system.

Let's consider a family system. The lineage is drawn

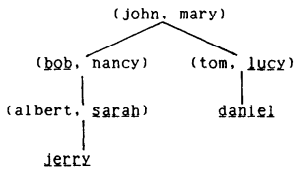


Fig. 2 The lineage.

as Fig. 2. A pair closed in a parentheses represents a couple. Names with underline represent children. The database "family" includes two tables—table "father" and table "mother." By the Prolog database system, we may create the database "family" and define some general rules as below.

```
?— db_init(family).          /* create database
YES                             "family"
?— invoke(family).           /* load the database
                                dictionary "family"
YES                             into main memory
?— define(father, 2).        /* define table "father"
                                with two columns
YES                             in the database
                                "family"
?— input(father).            /* cause data input for
                                table "father"
ACCEPT DATA READY, NO. OF COLUMNS=2
0: (john, bob).              /* john is bob's father
1: (john, lucy).                and so on
2: (bob, sarah).
3: (tom, daniel).
4: (albert, jerry).
5:
YES                          /* data have been stored
                                in table "father"
?— define(mother, 2).        /* define table "mother"
                                with two columns
YES                             in the database
                                "family"
?— input(mother).            /* cause data input for
                                table "mother"
ACCEPT DATA READY, NO. OF COLUMNS=2
0: (mary, bob).              /* mary is bob's mother
1: (mary, lucy).                and so on
2: (nancy, sarah).
3: (lucy, daniel).
4: (sarah, jerry).
5:
YES                          /* data have been stored
                                in table "mother"
?— [user].                   /* transfer mode to
                                user for defining
                                general rules
| couple(X, Y):-             /* if X is Z's father
    father(X, Z),               and Y is Z's mother
    mother(Y, Z).               then X and Y are
                                a couple
| parent(X, Y):-             /* if X is Y's father
    father(X, Y);               or X is Y's mother
    mother(X, Y).               then X is Y's parent
| offspring(X, Y, 1):-        /* if Y is X's parent
    parent(Y, X).               then X is Y's first
                                generation offspring.
| offspring(X, Y, N):-        /* if Y is Z's parent,
    parent(Y, Z),               X is Z's Mth generation
    offspring(X, Z, M),         offspring and N is
    N is M+1.                   M+1 then X is Y's
                                Nth generation
                                offspring
|
USER CONSULTED
?— couple(X, Y).             /* this query causes the
{(X, Y)} = {(john, mary),       join operation of
            (john, mary),       table "father" and
            (bob, nancy),       table "mother"
            (tom, lucy),
            (albert, sarah)}
YES
more?(y/n)y
NO
?— offspring                 /* fine john's Nth
    (X, john, N).               generation offsprings
{(X, N)} = {(bob, 1),
            (lucy, 1)}
YES
more?(y/n)y
{(X, N)} = {(sarah, 2)}
YES
more?(y/n)y
{(X, N)} = {(daniel, 2)}
YES
more?(y/n)y
{(X, N)} = {(jerry, 3)}
YES
more?(y/n)y
NO
```

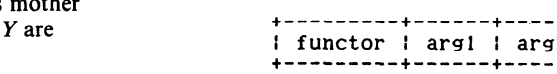
4. The Implementation

4.1 The Data Structure

In the Prolog database system, a term may be an integer, an atom, a variable, or a structure. The data format for a term is shown as below.

The tag represents the data type and the value represents an integer value or a pointer.

A compound term consists of three terms—a functor and two arguments as shown below. The functor is an atom and the argument may be any term.



example: The structure

```
maxi(X,Y,Y) :- X<Y,!.
```

is represented as

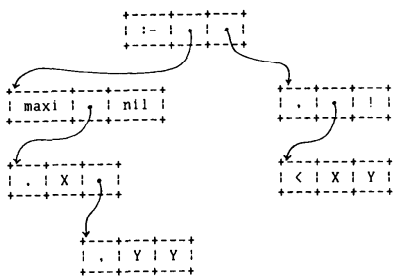


Fig. 3 Compound terms.

A structure can be represented by compound terms. An example is illustrated in Fig. 3. This data structure causes the garbage collection and the implementation ease.

4.2 The Binding Environments of Variables

In this system backtracking and unification are operated on IDB goal calls while set operations are applied on EDB goal calls. Therefore, a variable may be bound by a single term or a set of terms. The single term is placed on the variable stack which resides in main memory and the set of terms without variables is placed on the table stack which resides in secondary memory. The table stack is a stack of binding tables in which each column represents a set of terms for a bound variable. The binding tables are generated at each EDB goal call and registered in the database dictionary. Once the backtracking occurs the variable stack is popped. Similarly, the table stack is popped too and the binding tables are dropped from the database dictionary. These two stacks represents a set of substitutions.

In a program, each variable has its location in the variable stack. Once a variable is bound by a single term it can't be modified again until backtracking occurs. However, if a variable has been bound by a set after an EDB goal call it may be rebound by a single term after a successful IDB goal call again. In this case, if the rebinding term is not a variable the Prolog-DBMS interface will generate a condition for the next set operation. This condition will cause the bound variable to disappear from the binding table after the execution of the next set operation. An example is illustrated in Fig. 4. Once the query statement is evaluated the locations are reserved for variables X, Y and Z in the variable stack. The first goal is an EDB goal call. Hence the Prolog-DBMS interface generates a set operation command to RDBMS. After the execution of the set operation a binding table $t1$ is produced by RDBMS in the table stack. The variables X and Y are bound by a set at this time. The second goal is an IDB goal call. After the unification operation the variable X is rebound by an integer. This will cause the Prolog-DBMS interface to generate a condition $X=5$ for the third goal which is an EDB goal call. The variable X disappear from the current binding table $t2$ after the third goal call.

Example:

Suppose that table p and table q have been defined in the database "exampl", whose contents are shown as below.

p[1]	p[2]	p[3]
a	2	d
b	5	e
c	3	f
a	5	g

q[1]	q[2]
g	b
f	c
g	e

A general rule

```
r(5,W).
```

is defined as intensional database. After the evaluation of goal

```
?-invoke(exampl).
```

the database "exampl" is available for the following query

```
?-p(a,X,Y),r(X,Z),q(Y,Z).
```

The answers will be ((X,Y,Z))=((5,g,b),(5,g,e)) which are stored in the binding environment.

The proof tree followed by the set operation commands and the binding environments at each node is shown below.

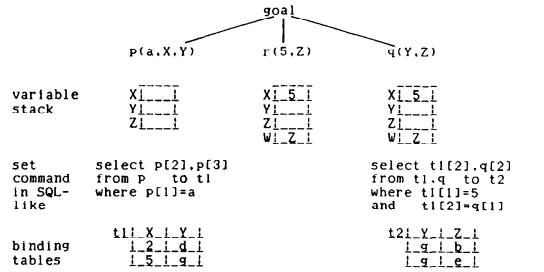


Fig. 4 An Example for Binding Environments and Set operation commands at each node of proof tree.

In fact, the table stack need not hold all binding tables which are produced at all EDB goal calls. From Fig. 4, after finishing the evaluation of the third goal the results are stored in the variable stack and the binding table $t2$. If there is no backtracking, the binding table $t1$ is not used any more and may be dropped from the table stack immediately. However, if there are alternatives for goal $r(X, Z)$, the binding table $t1$ can't be erased. Once the backtracking occurs, the binding table $t2$ is popped and the current binding table changes from $t2$ to $t1$. Of course, the variable stack is popped too.

4.3 The Set Operation Commands

Executing a goal statement the search is based on simple left-to-right and depth-first strategy. When an EDB goal or an output predicate is encountered the Prolog interpreter transfers control to the Prolog-DBMS interface. The Prolog-DBMS interface then generates set operation commands to the RDBMS.

In set operations, the operator "union" is equivalent to the operator "or" (;) in Prolog which causes the

set operator	source table 1	source table 2	destination table	condition pointer	action pointer

(a)

Fig. 5(a) The Set Operation Command Format.

log./arith. operator	operand 1 table id	operand 1 col. #	operand 2 table id	operand 2 col. #

(b)

Fig. 5(b) The Condition/Action Command Format.

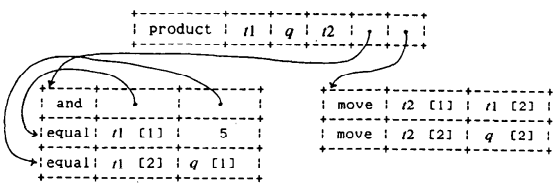
system to backtrack to the alternative. Except "union" operation, each non-procedure set operation can be converted into set operation commands as shown in Fig. 5. This command format may be explained as

"Operate table 1 and table 2. For each tuple if the condition is (not) satisfied then do the action and store the result in the destination table"

In the condition/action command, the column number is used to recognize the data field in a table rather than the attribute name. For example, a join operation as shown in Fig. 4

select $t1[2]$, $q[2]$
from $t1$, q to $t2$
where $t1[1]=5$
and $t1[2]=q[1]$

can be represented as



In this system, the set operation commands produce results with duplicate tuples. The set operators include:

Selection

This operation selects only one table to operate. The conventional selection and projection operations can be converted into this command.

Product

This operation concatenates two tables to operate. The conventional intersection, cartesian product and join operations belong to this command.

Difference

This operation produces a result table which is the difference of table 1 minus table 2.

The other additional operators (such as deletion, insertion etc.) can also be implemented easily in this format.

5. The Conclusion

In the conventional database management system, the data definition language is separate from the data manipulation language. Furthermore a host language (such as FORTRAN, COBOL, *PL/1*) is required in order to solve more complicated problems. Users have to take considerable time to learn these languages. In contrast, predicate logic serves as a single uniform language. The distinction between programs and data disappears. In addition, predicate logic has been recognized to be an effective language for representing knowledge. It also has "self-maintain" capability. Thus Prolog is the best choice to be a problem solver and relational database management language.

In our approach, we combine the Prolog programming language and the relational database system in a single Prolog database system. The binding environments consist of a variable stack and a table stack. Set operations are invisibly built in the system. Therefore, it facilitates the program/data description. The processing of a large volume of databases is not a problem any more. We believe that this system will provide a higher availability in future knowledge base researches.

References

1. KOWALSKI, R. (1978), Logic for Data Description. In: Logic and Data Bases (eds. Gallaire, H. and Minker, J.), Plenum Press, New York, pp. 77–103.

2. PARSAYE, K. (1983), Database Management, Knowledge Base Management and Expert System Development in PROLOG. Databases for Business & Office Applications. *Proc. of Annual Meeting, ACM*, pp. 159–178.

3. CHAKRAVARTHY, U. S., MINKER, J. and TRAN, D. (1982), Interfacing Predicate Logic Languages and Relational Databases. In Proceedings of the first Int. *Logic Programming Conf., Faculte des Sciences de Luminy Marseilles, France, Sept.* **14-17**th (1982), pp. 91–98.

4. LI, D. (1984), A PROLOG Database System, Research Studies Press Ltd.

5. YOKOTA, H., KUNIFUJI, S., KAKUTA, T., MIYAZAKI, N., SHIBAYAMA, S., MURAKAMI, K. (1984), An Enhanced Inference Mechanism for Generating Relational Algebra Queries. *Proc. of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, April **2–4** (1984), pp. 229–238.

(Received October 9, 1985)