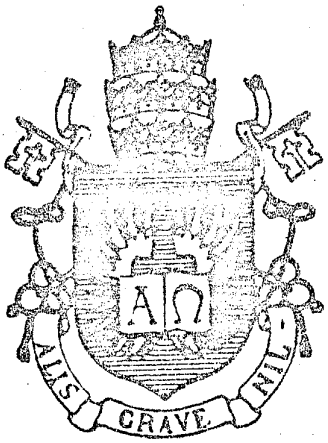

PUC

Series: Monografias em Ciencia da Computação

No. 21/89

TREATING WORKSPACE AND SQL FACTS UNIFORMILY IN PROLOG

Antonio L. Furtado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453

RIO DE JANEIRO - BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciencia da Computação, No 21/89

Editor: Paulo A. S. Veloso October, 1989

TREATING WORKSPACE AND SQL FACTS UNIFORMILY IN PROLOG

Antonio L. Furtado

* Research partly sponsored by FINEP

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

ABSTRACT

The transparent use of a Data Base Management System inside Prolog programs is motivated. A facility for this purpose is presented. The paper discusses the details of its implementation in ARITY Prolog.

KEYWORDS

Prolog, databases, relational model, SQL, null values.

RESUMO

O uso transparente de Sistemas de Gerencia de Bancos de Dados dentro de programas em Prolog é motivado. Uma facilidade com este objetivo é apresentada. O artigo discute os detalhes de sua implementação em ARITY Prolog.

PALAVRAS-CHAVE

Prolog, bancos de dados, modelo relacional, SQL, valores nulos.

1. Towards the transparent use of databases in Prolog

Throughout this paper, we shall be concerned with the representation of facts in Prolog. A Prolog fact is a ground unit clause, i.e. a clause without variables and without antecedents. Prolog facts and database records are then compatible alternatives to represent the same kind of information.

Prolog programmers may want to resort to a Data Base Management System (DBMS) to handle large collections of facts, without having to learn the DBMS syntax and semantics. They may even like to be unaware of whether certain classes of facts are ordinary Prolog clauses or DBMS records, when using a Prolog system interactively. For large "batch" Prolog programs, this transparency with respect to the use of a DBMS means that it should be possible to change the decision on which of the two ways will be used to keep each class of facts, without rewriting the programs significantly.

Some Prolog systems support interfaces with a DBMS. Relational DBMSs are paticularly suitable, since Prolog is congenial with the relational model [Ko]. Both ARITY Prolog [Ma] and IBM Prolog [Gi] provide an interface to an SQL package [Da]. Both interfaces consist of a Prolog predicate one of whose arguments is an SQL command, thereby requiring that users express themselves in SQL.

The present paper presents a facility, to be used with ARITY Prolog, which fully meets the objective proposed in the first paragraph with respect to queries. These are the topic of section 2. Section 3 discusses insertions and deletions. Section 4 suggests how to try the system, with the code supplied in the appendices.

2. Queries

If "emp(X,Y,Z)" is a Prolog predicate, where the parameters correspond respectively to name, salary and address of employees, and there exist in the workspace clauses such as:

```
emp($John$,100,$RJ$).
emp($Mary$,200,$SP$).
emp($Laura$,120,$RJ$).
emp($Peter$,300,$RJ$).
```

a goal like

```
:- emp(N,S,$RJ$), S > 100 .
```

would return the names and salaries of employees who live in RJ and earn more than 100. With our facility, if "emp" is instead an

SQL table, exactly the same goal can be entered, yielding the same result.

The reader will note that the result (in both cases) consists of the pair of values Laura and 120, for N and S respectively. If backtracking is requested, for example through the ";" prompt, the values Peter and 300 will be returned, again in both cases.

What our facility does is, in case of an SQL table, to transform the call to "emp" into the SQL command:

```
select name, salary from emp where city = 'RJ'
```

and execute it through the standard interface, making sure that variables N and S be instantiated with the name and salary values retrieved. Next, the Prolog expression "S > 100" is executed with the expected results.

It would be probably more efficient to look at the entire goal and generate something like:

```
select name, salary from emp where city = 'RJ'and salary > 100
```

but the usual way to accomplish this would be to process all goals through a meta-interpreter, which may be inconvenient. So, we decided not to worry about these kinds of optimization, and to use the following straightforward strategy with respect to the arguments supplied for each parameter P of "emp":

- if P is a variable,
  - the name of the corresponding SQL column is included in the list of columns for the "select" part of the command;
  - variable P itself is included in the list of variables to receive values from the tuples retrieved.
- if P is not a variable, the value of P is included in the list of terms "column = value" for the "where" part of the command.

The empty list, "[]" (which in some systems is identical with the "nil" atom), stands for the undefined value. It is recognized as such by the ARITY Prolog interface with SQL and our facility handles it as expected. In the second case above, terms in the "where" part become "column is null", whenever the value involved is the undefined value.

It would be desirable to further specialize the first case, to recognize when the anonymous variable "_" is used, so as not to include the respective column in the "select" list. Unfortunately this would again require a meta-interpreter, with the additional burden of providing a lexical scanner of its own to process each goal as the goal is read as a character string.

All that we require for SQL tables to be used as described, is to enter at the beginning of an interactive session (or of a program) a call to our predicate "sql_fact(<table>)", for each such table. For instance:

```
:- sql_fact(emp).
```

The consequence is that four clauses are added to the workspace, of which we now indicate the one that has the name of the table and that is used in our example goal:

```
emp(A,B,C) :-
    prep_sel(emp(A,B,C),[$name$,$sal$,$city$],D),
    D.
```

The names of the columns are taken by "sql_fact" from the SQL system catalogues. Predicate "prep_sel" produces in D, in our example query:

```
sql($select name, salary from emp where city = 'RJ';$,[A,B]).
```

which would seem to be a call to the interface predicate to execute the SQL command, given between quotes (actually "$" signs, in ARITY), to return the values retrieved in [A,B]. But "sql" is not yet the interface predicate, being defined by:

```
sql(X,Y) :-
  Z =.. [key|Y],
  exec_sql(X,key),
  findall(Y,(recorded(key,Z,R), erase(R)), S),
  on(Y,S).
```

where, finally, "exec_sql" is the interface predicate. The trouble with "exec_sql" is that it retrieves at once all tuples satisfying the command and stores them as ARITY records. This is probably more efficient than fetching each tuple separately upon bactracking, but unfortunately goes against our intention to hide the fact that SQL tables are being used. What our predicate, "sql", does is to immediately collect in a set S all these tuples, simultaneously erasing them from the workspace; then the set-membership predicate "on" takes one tuple at a time, on normal backtracking, from S. Note that a less careful program might leave one or more records unerased, wasting space and causing some unexpected results for other queries. We must admit, however, that our solution is far from ideally efficient.

3. Insertions and deletions

If "emp" were a Prolog predicate, we would be able to add new clauses to its definition by entering, say

```
:- assert(emp($Lucy$,130,$SP$)).
```

which cannot be directly extended to the case of "emp" being an SQL table, since we cannot modify a built-in predicate like "assert". But a still quite reasonable solution, involving a minimal change to programs to accomodate our transparency objective, is to define a new predicate, "xassert", with the ability to distinguish ordinary Prolog clauses (and treat them by the original "assert") from references to SQL tables that must be treated in a different way. The definition of "xassert" is:

```
xassert(P) :-
  not is_sql_fact(P),
  assert(P).
xassert(P) :-
  is_sql_fact(P),
  sql_assert(P).

is_sql_fact(X) :-
  functor(X,F,N),
  functor(Y,F,N),
  sql_attrs(Y).
```

A "fact" is therefore an SQL fact when this is indicated by an "sql_attrs" clause. This clause is one of the four produced by the initial call to "sql_fact"; for our example, it is:

```
sql_attrs(emp($name$,$sal$,$city$)).
```

The third of the four clauses produced is the "sql_assert" clause below:

```
sql_assert(emp(A,B,C)) :-
    prep_ass([$name$,$sal$,$city$],[A,B,C],D,E),
    F =.. [emp|D],
    G =.. [values|E],
    conc([$insert into $,F,G,$;$],H),
    exec_sql(H).
```

Thus, if "emp" is a table and the following goal is executed:

```
:- xassert(emp($Lucy$,130,$SP)).
```

The above "sql_assert" clause will generate in H and execute through the interface the command:

```
insert into emp(name,sal,city) values('Lucy',130,'SP')
```

Deletions are treated in an analogous way. The obvious "xretract" predicate is:

```
xretract(P) :-
  not is_sql_fact(P),
  retract(P).
xretract(P) :-
  is_sql_fact(P),
  sql_retract(P).
```

with "sql_retract", the fourth clause produced by "sql_fact", being for our example:

```
sql_retract(emp(A,B,C)) :-
    listvar(emp(A,B,C),D),
    ( D = [],
      !
    ; emp(A,B,C)
    ),
    prep_retr([$name$,$sal$,$city$],[A,B,C],E),
    conc([$delete from emp where $|E],F),
    exec_sql(F).
```

so that for the goal:

```
:- xretract(emp(N,_,$SP$)).
```

the commands below are generated and executed through the interface:

```
select name, sal from emp where city = 'SP'
delete from emp where name = 'Mary' and sal = 200 and city = 'SP'
```

the preliminary "select" command being intended to instantiate all variables. Upon backtracking, each tuple of employees in city SP will be deleted, and the names of the employees returned in variable N. Again, this is exactly as the ordinary "retract" is supposed to behave.

Undefined values are also correctly handled by both "xassert" and "xretract". Recall, in particular, that the SQL "insert" command places nulls in positions for which no values are indicated. So, the command

```
insert into emp(name,city) values('Judy','RJ')
```

is generated to execute

```
:- xassert(emp($Judy$,[],$RJ$)).
```

4. Trying the facility

Although not particularly geared to efficiency, our facility is still a convenient way to make database handling transparent to Prolog users. To summarize, it is sufficient to execute, at the outset, goals of the form

```
:- sql_fact(<name of table>).
```

for each table to be used, and to replace all occurrences of "assert" and "retract" by "xassert" and "xretract", without bothering to check whether a Prolog clause or an SQL tuple is involved.

Of course preliminary steps are needed: the facility must be loaded by some

```
:- consult(<file1>).
```

goal and the database must be created and initialized through

```
:- sql_consult('<file2>.sql').
```

To allow the interested reader to try the facility, we included both the complete program for the facility, to become the contents of the first file (say, 'rel.ari'), and a file of type 'sql', to create and populate an example database (say, 'ex.sql'). With these files, the preliminary steps are then:

```
:- consult(rel), sql_consult('ex.sql').
:- sql_fact(emp), sql_fact(dept).
```

Simple examples with the undefined value that can be tried are:

```
:- xassert(emp($Judy$,[],$RJ$)).
:- emp($Judy$,X,_), write(X), nl.
:- xretract(emp(X,[],Y)).
```

Another possible goal to try is:

```
:- dept(D,Mg), emp(Mg,_,C), write(D - C), nl, fail.
```

to find the headquarter of each department, assumed to be the city where the respective manager is located. This definition of "headquarter" can be also captured by:

```
hq(D,C) :-
  dept(D,Mg),
  emp(Mg,_,C).
```

which can be queried, for instance, by

```
:- hq(D,$SP$).
```

to find departments with headquarter in city SP.

Database practitioners will note that finding the headquarters of departments involve a database "join" of the two tables. Moreover, the "hq" predicate is reminiscent of a database "view". However our point is that Prolog-minded users need not be concerned with database concepts. They will quite naturally take advantage of the ability offered by Prolog to introduce very general recursive definitions.

The work reported is part of project NICE, towards cooperative environments for information systems usage. A companion paper [Fu] describes the plan-generation algorithm that is also part of the current prototype. With the facility, it has been easy to adapt the plan-generator to handle both Prolog and SQL facts in a totally uniform way.

References

[Da] C. J. Date - "Database - a primer" - Addison Wesley (1983).

[Fu] A. L. Furtado - "Some extensions to Warren's plan-generation algorithm" - PUC/RJ technical report 20/89.

[Gi] M. Gillet - "VM/Programming in Logic" - IBM Manual SB11-6374-0 (1985).

[Ko] R. Kowalski - "Logic for data description" - in "Logic and Data Bases" - H. Gallaire and J. Minker (eds.) - Plenum (1978).

[Ma] C. Marcus - "Prolog programming" - Addison-Wesley (1986).

Appendix I
Program file implementing the facility

```
% sql interface

sql_fact(F) :-
  columns(F,C),
  P1 =.. [F|C],
  assert(sql_attrs(P1)),
  card(C,N),
  functor(P,F,N),
  assert((P :-
            prep_sel(P,C,Sq),
            Sq)),
  P =.. [_|A],
  assert((sql_assert(P) :-
            prep_ass(C,A,C1,B),
            U =.. [F|C1],
            V =.. [values|B],
            conc([$insert into $,U,V,$;$],I),
            exec_sql(I))),
  concat([$delete from $,F,$ where $],R1),
  assert((sql_retract(P) :-
            listvar(P,L),
            (L = [], !; P),
            prep retr(C,A,T),
            conc([R1|T],R),
            exec_sql(R))).

columns(T,Cs) :-
  table(T,N,_,_),
  cls(T,N,N,Cs).

cls(T,0,N,[]).
cls(T,I,N,[C|R]) :-
  K is N - I + 1,
  column_table(C,T,K,_,_),
  J is I - 1,
  cls(T,J,N,R).

prep_sel(G,Cols,Sq) :-
  functor(G,F,N),
  G =.. [F|L],
  prep_sel1(L,Cols,L1,L2,L3),
  (L1 == [], !, S1 = $ * $ ;
   L2 == [], !, S1 = $ * $ ;
   string_term(St1,L1),
   string_length(St1,Le1),
   Le11 is Le1 - 2,
   substring(St1,1,Le11,S1)),
```

```
    (L2 == [], !, S2 = $$ ;
     conc([$ where $|L2],St2),
     string_length(St2,Le2),
     Le21 is Le2 - 5,
     substring(St2,0,Le21,S2)),
    (L3 == [], !, S3 = L ;
     S3 = L3),
    concat([$select $,S1,$ from $,F,S2,$;$],Sa),
    Sq = sql(Sa,S3).

prep_sel1([],[],[],[],[]).
prep_sel1([P|R],[D|S],[D|T1],T2,[P|T3]) :-
    var(P), !,
    prep_sel1(R,S,T1,T2,T3).
prep_sel1([P|R],[D|S],T1,[DQ, $ and $|T2],T3) :-
    (string(P), !, concat([$'$,P,$'$],Q), DQ = (D = Q);
     P == [], !, DQ = (D is null);
     DQ = (D = P)),
    prep_sel1(R,S,T1,T2,T3).

prep_ass([],[],[],[]).
prep_ass([C|R1],[V|R2],[C|R3],[V1|R4]) :-
    not V == [], !,
    prep_val(V,V1),
    prep_ass(R1,R2,R3,R4).
prep_ass([C|R1],[V|R2],R3,R4) :-
    prep_ass(R1,R2,R3,R4).

prep_retr([],[],[]).
prep_retr([C|R1],[V|R2],[C = V1, Conj|R3]) :-
    not V == [], !,
    (R1 == [], !, Conj = $;$;
     Conj = $ and $),
    prep_val(V,V1),
    prep_retr(R1,R2,R3).
prep_retr([C|R1],[V|R2],[C is null, Conj|R3]) :-
    (R1 == [], !, Conj = $;$;
     Conj = $ and $),
    prep_retr(R1,R2,R3).

prep_val(A,B) :-
    (string(A), !, concat([$'$,A,$'$],B);
     B = A).

sql(X,Y) :-
    Z =.. [key|Y],
    exec_sql(X,key),
    findall(Y,(recorded(key,Z,R), erase(R)),S),
    on(Y,S).

xassert(P) :-
```

```
  not is_sql_fact(P),
  assert(P).
xassert(P) :-
  is_sql_fact(P),
  sql_assert(P).

xretract(P) :-
  not is_sql_fact(P),
  retract(P).
xretract(P) :-
  is_sql_fact(P),
  sql_retract(P).

is_sql_fact(X) :-
  functor(X,F,N),
  functor(Y,F,N),
  sql_attrs(Y).


% general utilities

conc(Ls,S) :-
  conc1(Ls,Ls1),
  concat(Ls1,S).

conc1([],[]).
conc1([X|R],[Y|S]) :-
  string_term(Y,X),
  conc1(R,S).

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

append([],X,X).
append([X|R],Y,[X|S]) :-
  append(R,Y,S).

card([],0).
card([A|R],N) :-
  card(R,M),
  N is M + 1 .

reverse(Xs,Ys) :-
  rev1(Xs,[],Ys).

rev1([X|Xs],A,Ys) :-
  rev1(Xs,[X|A],Ys).
rev1([],Ys,Ys).

listvar(X,Y) :-
```

```
  v_flat(X,[],T1),
  reverse(T1,T2),
  no_dup(T2,T3),
  reverse(T3,Y).

no_dup([X|Xs],Ys) :-
  v_on(X,Xs), !,
  no_dup(Xs,Ys).
no_dup([X|Xs],[X|Ys]) :-
  not v_on(X,Xs),
  no_dup(Xs,Ys).
no_dup([],[]).

v_flat(A,B,[A|B]) :-
  var(A), !.
v_flat(A,B,B) :-
  atomic(A), !.
v_flat([A|R],B,C) :- !,
  v_flat(R,B,T),
  v_flat(A,T,C).
v_flat(A,B,C) :-
  A =.. [F|R], !,
  v_flat(R,B,C).

v_on(X,[Y|Z]) :-
  var(Y),!,
  (X == Y ;
   v_on(X,Z)).
v_on(X,[X|_]).
v_on(X,[Y|Z]) :-
  v_on(X,Z).
```

Appendix II
Database creation file

```
create table emp  (
        name char(10) not null,
        sal integer,
        city char(2)
);

create table dept (
        dname char(2) not null,
        mg char(10)
);

insert into emp values ('John',100,'RJ');
insert into emp values ('Mary',200,'SP');
insert into emp values ('Laura',120,'RJ');
insert into emp values ('Peter',300,'RJ');

insert into dept values ('D1','Mary');
insert into dept values ('D2','Peter');
```