A database support system for PROLOG

Jan Chomicki
Wlodzimierz Grudzinski

Institute of Informatics
Warsaw University
PKiN
POB 1210
00-901 Warsaw, Poland

1.Introduction

In the current literature PROLOG is often connected with databases, eg (Warren 81), (Lloyd 82). That intuition is, in our opinion, correct. Nevertheless, two main points have been missed: how to organize large PROLOG databases and why they should be superior to the conventional, eg relational, ones. In this paper, we concentrate on the former question. We start by discussing the advantages of PROLOG over the relational model of data . Then we outline simple PROLOG solutions to several database problems. However, some of the most difficult issues of database management do not depend on the language used for defining and manipulating databases.

We describe a database support system for PROLOG implemented at the Institute of Informatics, Warsaw University. According to the taxonomies of (Lloyd 82), our efforts may be described as an interpreted approach, providing (currently)limited database management system capabilities.

The system is primarily meant for the storage, retrieval and modification of a form of PROLOG clauses. Unit clauses designated by PROLOG are managed by the support system which stores them on disk. Other clauses are treated in the standard way by the PROLOG interpreter. When a clause from disk is required, the interpreter issues a query to the support system. Then the system performs preliminary unification and returns all the clauses possibly matching the query(one at a time). The final unification and the binding of variables are performed by the interpreter.

To cope with growing files the database organization and access method is dynamic and based on extendible hashing (Fagin 79), augmented to allow partial-match retrieval (Lloyd 80). A general method has been developed for handling incomplete information (Chomicki 83). This method is used for storing, retrieving and modifying unit PROLOG clauses with variables.

2.PROLOG compared to relational languages

2.1.Relational model of data

Base relations are represented in PROLOG as procedures consisting only of unit clauses.

Domains are not directly mapped to the constructs of the language. They may be defined as unary (not necessarily base) relations, but then unit clauses of a n-ary relation should be augmented by the calls to the procedures defining its domains, so the clauses are no more unit. Or, to achieve more restrictive typing, domains may be treated as functors. But then, for the matching to succeed, queries should submit appropriate domain (functor) names.

Attributes are implicit in the order of relation columns. Their naming for further reference may be locally

(within a clause) obtained by introducing appropriately named PROLOG variables.

Keys(unique tuple identifiers) are in no way supported by PROLOG and, if required, should be defined as integrity constraints(cf section 2.5).

Throughout this paper, we use the original PROLOG notation (Roussel 75). Predicate names are in uppercase, variable names - in lowercase. The goal(procedure head) of the clause is preceded by the "+" character, the premisses - by the "-" character.

2.2.Relational algebra

As the following examples show, PROLOG easily supports relational algebra (Ullman 80) operators: selection, projection and join.

Example 1.

Let R be a binary relation with two numeric attributes: A and B.

Selection:

$\sigma_{A=5}(R)$	+R1(5,x) -R(5,x).
$\sigma_{A=B}(R)$	+R2(x,x) -R(x,x).
$\sigma_{5<A}(R)$	+R3(x,y) -R(x,y) -LESS(5,x).

Projection:

$\pi_A(R)$	+R4(x) -R(x,y).

Equijoin:

$R \bowtie S$	+R5(x,y,z) -R(x,y) -S(y,z).

Conjunction of selection conditions may be expressed by their enumeration in clause premisses and disjunction - by clause variants. Hence in PROLOG the selection condition is in disjunctive normal form.

Now consider set-theoretic operators: union and difference. (or equivalently (Ullman 80): union and division). Union is straightforwardly modelled by clause variants. To define set difference we should resort to some form of negation.

Example 2.

Suppose we write down the difference as

+Q(x,y) -R(x,y) -NOT(S(x,y)).

If no clause defining R contains variables, the argument of NOT will be ground and NOT will be evaluated correctly in the standard way.

2.3.Nulls

The difficulty in defining relational algebra operators in the presence of null(undefined) values has been recognized for some time already: (Codd 79), (Vassiliou 80). There are two ways of representing nulls in PROLOG: as variables or as ground terms.

If null is represented as an unbound variable, the semantics of stored null values ('missing', 'any', 'not applicable')may be enforced by restricting the result of the query (by the predicates differentiating several sorts of nulls). Furthermore, the standard predicates, eg EQUAL or LESS, should be extended to capture properties of different nulls.

If nulls are represented as ground terms, a null matches only either a variable or itself in the query. This disallows the interpretation of a null as an 'any' value matching all the values.

In our opinion, different nulls should be differently represented. More general nulls, eg 'missing' or 'any', expressing possible relevance of the information in a clause to many queries, should be handled as variables. More specific ones, eg 'not applicable', matching no value in a query except itself and a variable, may be treated like a

ground term. Various null values and their semantics are described in: (ANSI 75), (Vassiliou 80), (Zaniolo 82). Note that representing null value as a variable may lead to well known problems with negation in PROLOG: (Clark 79), (Naish 83).

Example 3.

Returning to Example 2, let us define a clause +R(5,g) with the variable g standing for the 'any' null value.
Now writing

+Q(x,y) -R(x,y) -NOT(S(x,y)).

fails to produce the desired result: "all the elements of R which are not in S".
To see that, it is sufficient to make both procedures R(and S) consist only of one unit clause: +R(5,g). (and +S(5,2).).
The query: "is Q non-empty? ", expressed as

-Q(u,w)

fails instead of producing a positive answer.

The above effect may be partly remedied by extending NOT, like EQUAL or LESS, to capture the semantics of nulls.

2.4. Views

Precisely in the same way as queries (relational algebra expressions), database views may be defined in PROLOG. However it is unclear how to do view updates. Currently in PROLOG, modifying the view has no effect on the underlying base relations. Each updating user must access base relations. So PROLOG views do not fulfill their fundamental role of protection mechanisms, hiding information from users. An explicit translation of view updates to the updates of the underlying base relations is required. That translation is determined by the semantics of relations and attributes involved. Sevaral strategies have been proposed in: (Dayal 82), (Bancilhon 81), (Paolini 82), (Siklossy 82). A general mechanism should allow the definition of procedures updating base relations and triggered by view updates (Shipman 81). Such a mechanism is outlined in section 2.5.

It is rather obvious that PROLOG clauses, containing eg recursive calls, are more general than relational views. Nevertheless it is an open problem whether the strategies for updating relational views may be generalized to arbitrary PROLOG views.

2.5. Integrity constraints

Furthermore, it is not well known how to impose integrity constraints in PROLOG. The constraints assert about the consistency of the database, so they should be defined at some meta-PROLOG level. The only solution of that problem we know of was proposed in (Bowen 81). It requires major changes in the PROLOG interpreter. As it supports arbitrary assertions expressed in first order logic, the computational complexity of its implementation would be enormous. We think of developing a less powerful but simpler and more efficient method, outlined below.

Pattern-directed procedure invocation in PROLOG may be used to solve the problems of integrity constraints enforcement, view updates and general triggers. We define two database modification operators: INSERT(clause) and DELETE(clause). They are hidden from the user who sees only "safe" (consistency preserving) operators: INCLUDE(clause) and EXCLUDE(clause) defined as

+INCLUDE(c) -INSERTSAFE(c) -INSERT(c).
+EXCLUDE(c) -DELETESAFE(c) -DELETE(c).

Both INSERTSAFE and DELETESAFE are defined as conjunctions of individual integrity constraints:

+INSERTSAFE(c) -IS1(c) ... -ISk(c).
+DELETESAFE(c) -DS1(c) ... DSp(c).

The problem is how to get the individual integrity constraints. They may be generated manually from informal specifications.

Example 4.
In the relation R first attribute functionally determines the second attribute.
+IS1(R(x,y)) -R(x,y1) -NOT(EQUAL(y,y1)) -/ -FAIL.
+IS1(R(x,y)).

Example 5.
The domain of the first attribute of S contains the domain of the second attribute of R.
+IS2(R(x,y)) -S(y).
+IS2(S(y)).
+DS2(S(y)) -R(x,y) -/ -FAIL.
+DS2(S(y)).
+DS2(R(x,y)).

As may be seen from the above examples, one of the pair of constraints on insertion (deletion) is often tautologically true and may be omitted in the definition of INSERTSAFE (DELETESAFE). However we do not address here further issues of optimizing constraint checking, referring the reader to (Nicolas 82) and (Blaustein 81).

Triggers (Eswaran 76) provide an interesting alternative (or complement) to assertions. In tead of checking the constraints, we may prefer to correct their violations. Now

+INCLUDE1(c) -INSERTTRIGGER(c) -INCLUDE(c).
+EXCLUDE1(c) -DELETETRIGGER(c) -EXCLUDE(c).

and

+INSERTTRIGGER(c) -IT1(c) ... -ITm(c).
+DELETETRIGGER(c) -DT1(c) ... -DTn(c).

The execution of a trigger may obviate integrity checking as seen below.

Example 6.
All the assertions from Example 5 may be replaced by three triggers:

+IT1(R(x,y)) -INSERT(S(y)).
+DT1(S(y)) -R(x,y) -DELETE(R(x,y)) -DT1(S(y)) -/.
+DT1(S(y)).

Triggers may perform view updates identically as corrective actions above.
Our approach to the consistency of PROLOG databases is rather preliminary. In fact, we omitted the most

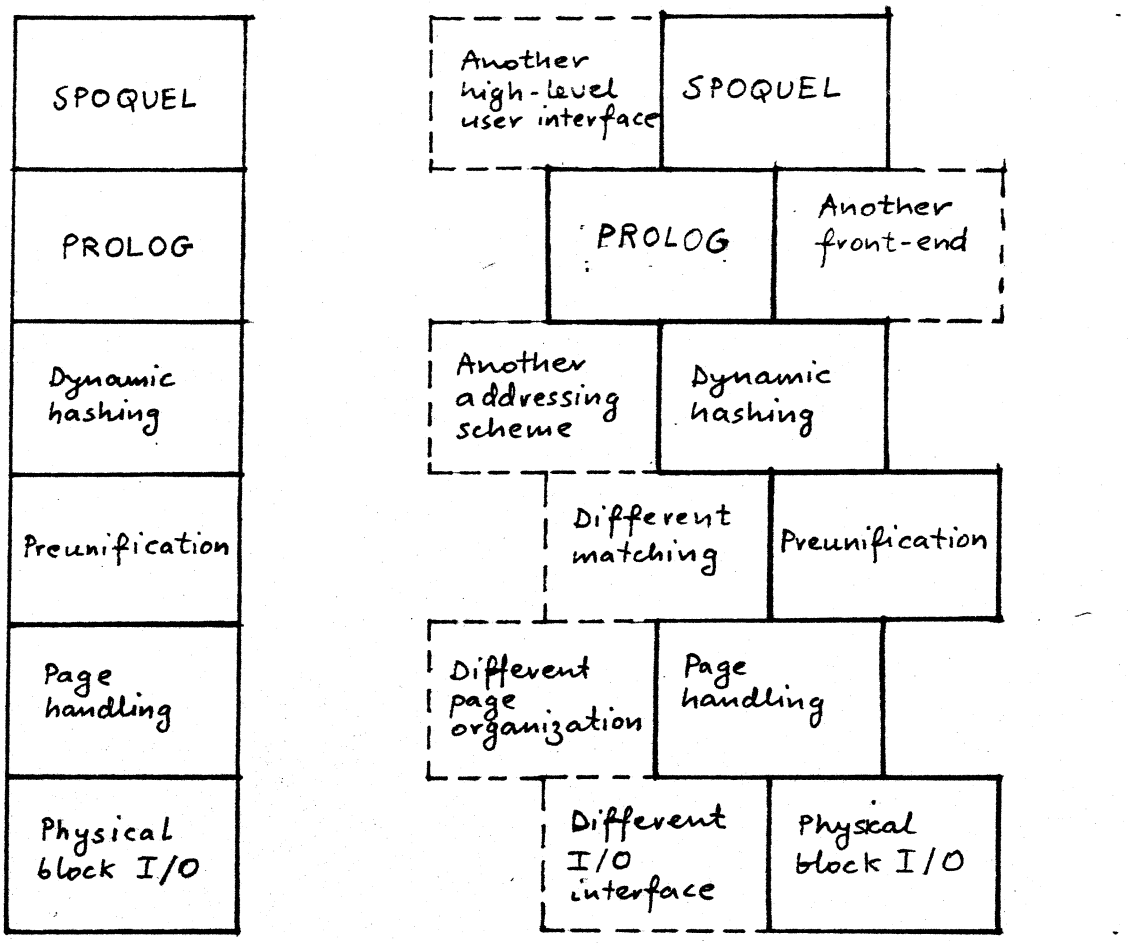
difficult problems of:

- supporting transactions (multiple actions) as consistency units (Eswaran 75)
- backing out the transactions violating database consistency
- efficient checking of arbitrary integrity assertions during or at the end of a transaction.

We do not think that the solution of the above problems is any easier with PROLOG than with conventional programming languages.

3.System architecture

System architecture may be described as an hierarchy of six levels. This paper discusses only the lowest four.



1.SPOQUEL is a SEQUEL-like query language described in (Kluzniak 83b). The interpreter of SPOQUEL is written in PROLOG.

2.The PROLOG interpreter (Kluzniak 83a) underwent only a minor change by incorporating new database primitives described in the next section.

3.Upon receiving a request from PROLOG, the addressing layer forwards the request to specific database pages. This layer performs also simple catalog management to keep track of definitions of relations.

4.Preunification filters out clauses retrieved by page handling layer and not matching the clause passed from PROLOG as the argument of the request. The variables are not instantiated, but only the matching clauses are returned. The unification is completed by the interpreter.

5.The page handling layer performs software paging and retrieves (inserts,deletes) tuples and auxiliary information from core buffers. The tuples are stored as variable-length and arbitrarily nested records. The page size is 2048 bytes.

6.Physical block I/O which supports paging is based on the file system of the underlying operating system.

Now consider the (relatively) easy-to-do alternatives marked by dashed lines.

The effects of their introduction would be limited to to the layer of their application.

1.Another high-level user interface, eg Query-By-Example, may be implemented in PROLOG analogously to SPOQUEL.

2.Another front-end would make possible to get into our system beside PROLOG. We actually had to develop

such a front-end for the purpose of testing.
3.If dynamic hashing does not turn out to perform satisfactorily in some applications, we may choose to replace it by another addressing scheme, eg multidimensional binary search trees. Such a possibility was outlined in (Chomicki 83).
4.Currently the preunification adheres to standard PROLOG semantics. However, it would be easy to adapt the preunification to somewhat different requirements, eg pattern-matching in text.
5.The variable-length records are flexible but may turn out to be inefficient, requiring an additional level of indirection. For conventional formatted databases, fixed length records should rather be considered.
6.It is also conceivable to bypass the file system and perform the I/O directly without protection and bookkeeping overhead.

On the whole, the interfaces between the layers are narrow, the layers are loosely coupled and besides there seems to be a high potential of asynchronous processing.

The PROLOG interpreter and the support system are written in CDL-2 (Koster 75), a highly modular language. CDL-2 gives the possibility of gluing high-level control and parameter-passing structures together with assembler macros. The support system consists of 3000 lines of source code. About 15% of code are written in MACRO-11 assembler. The system is being developed on the SM-4 minicomputer (functionally equivalent to PDP-11/40) under the RSX-11M operating system.

4. Database Support System.

In our database support system we have implemented extendible hashing scheme based on (Fagin 79) with an extension to partial-match retrieval along the lines proposed by (Lloyd 82). That method was chosen because of very good search performance which doesn't deterioriate for dynamic (growing and shrinking) files. This method is simple to implement, even with an extension to handle partial-match queries typical for PROLOG oriented databases.

This scheme has been adapted to handle incomplete information (Chomicki 83).

In the sequel we shall refer to PROLOG unit clauses as tuples.

4.1. Extendible hashing.

Extendible hashing is a method for handling dynamic files. There is a hash function h from the key space (domains of attributes) of the tuples to the set of bit strings of length k

h : K -> Bk

The details of the hash function will be discussed in section 4.2.

The file is structured into directory and database level.

The directory contains pointers to database pages and is characterized by a number, called the depth of the directory. The directory entries are indexed by all bit strings of the length d (d<=k) and d is the current depth of the directory. We called these bit strings (after (Lloyd 82)) the indexing strings.

When a directory entry is indexed by the string b1.....bd it means that the database page, pointed at by this entry, stores all the tuples for which h(K) starts with b1.....bd.

There are 2**d entries in the directory.

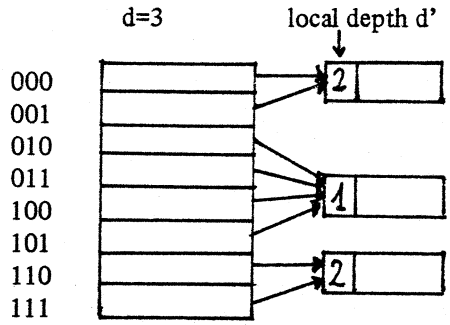


Fig 4.1

Database pages contain tuples. The order of tuples is immaterial. They are stored in structured compact form as variable-length arbitrarily-nested records. Each database page has a header which contains the local depth d' for this page. The local depth of a page is d' (d'<=d) iff there are 2**(d-d') entries in the directory, pointing at that page.

In Figure 4.1 the local depth of database page pointed at by the first directory entry is 2 and depth of the directory is 3. That means that this page contains all tuples for which h(K) agrees with 000 on the first 2 places. Thus the 001 entry also points at this database page.

Suppose that we want to insert a new tuple with a key K0. We calculate h(K0) and select its first d bits. Next we do a simple computation to find a coresponding entry in the directory. Following the pointer we find a database page on which a tuple should be placed. When this page is already full and d>d' then t splits into two database pages.

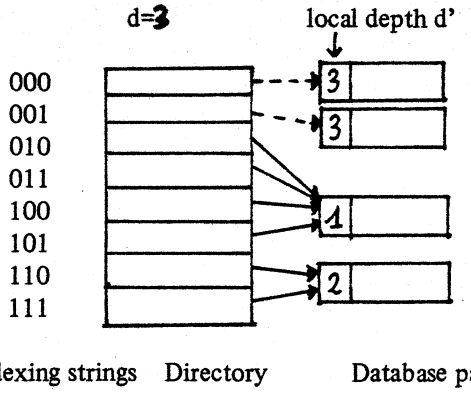


Fig 4.2

A new, initialy empty, database page is allocated and the tuples from the full page are hashed again. If the d'+1 th bit from h(K) of a tuple is 1 it is put on the new page. Otherwise it remains on the old one. The new tuple is treated identically. The local depths of both pages are set to d'+1.

If the database page is full and local depth is equal to the depth of directory (d=d') then the directory doubles in size as shown on the Figure 4.3 and the database page splits. The depth of the directory is increased by 1.

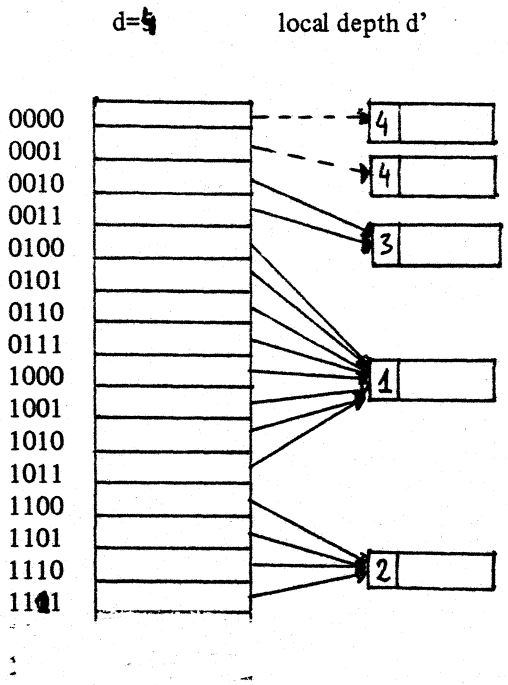


Fig 4.3

There is no need to access database pages during the doubling of the directory. When the directory exceeds one page, new directory page must be allocated during its doubling. However it does not happen very often.

In the case when the file is shrinking significantly after a large number of deletions and dictionary occupied many pages, it can be reduced twice in size.

4.2. Partial-match retrieval using extendible hashing.

The hash function h : A →Bk where A - set of all domains of relation's attributes is constructed. We are using one hash function (a kind of square hashing) for all attributes (but different hash functions can be constructed for each domain). The hash function uses as arguments the top level functor of clause attributes.

Let the value of the attribute ai in a query be vi for i=1,...,n. The final string which indexes a directory entry is constructed by selecting bits from each h(vi) according to a predefined choice vector (i1,i2,....,id) where d-depth of the directory. The m'th position in the indexing string will be filled by the first so far unused bit in the string h(vim).

Example 7.

When (4,1,4,2) is a choice vector then the indexing string contains
- first bit from the string h(v4),
- first bit from the string h(v1),
- second bit from the string h(v4),
- first bit from the string h(v2).

The methods of designing choice vectors are not considered here. The reader is referred to (Lloyd 80) and (Lloyd 82) for a discussion of this issue.

A partial-match query is a query in which an arbitrary subset of the attributes of the tuple is specified.

A query and a tuple match if all theirs specified attributes values are equal.

The result set of a query is the set of all pages where a matching tuples may reside.

When an incomplete query is considered and the choice vector indicates a bit from the unspecified attribute on the m'th place, then result set doubles, because both 0 and 1 should appear on the m'th place of all appriopriate indexing strings.

Example 8

For a relation T, choice vector (1,2,1) and the value of the hash function
for the second attribute h(GREEN)=010 the query
-T(x,GREEN,15) has the result set indexed by strings :
000, 001, 100, 101.

For a fully specified (complete) query only one page address is computed, so no more than two page accesses are necessary to find an answer.

4.3. Storage and retrieval of incomplete tuples.

The value of 'any' is introduced which is equivalent to PROLOG variable in the sense that 'any' matches every value in each domain and itself. When incomplete tuples appear, the set of tuples possibly matching a query grows exponentialy with the number of any-valued (unspecified) attributes.

Example 9

The query -S(TOKYO,1964) has four possible matchings :
+S(any,any), +S(any,1964), +S(TOKYO,any), +S(TOKYO,1964).

When an unspecified attribute provides no bits for the choice vector, the tuple is treated as complete.

We implemented a parametrized family of methods of the storage and retrieval of incomplete tuples.

Let m be the length of choice vector for the relation F and t an incomplete tuple to be inserted into F.
There are two extremal methods :
Method m (full replication)

- compute the result set of t and put a copy of t on each database page of this set. This method is time-optimal because the number of page accesses it requires is equal to the cardinality of the result set of a query.

However it gives a high storage redundancy - in the worst case 2**m.

Method 0 (no replication)

- put t on an arbitrary chosen page from the result set of t. This method is space-optimal, but search time can be unacceptable, requiring an access to each page, for we have no cues whether and where the incomplete tuples matching the query reside.

There is a family of intermediate methods numbered from 1 to m-1.

Method i

- tuples from each database page indexed by string b1...bi-1 bi bi+1....bm obtained by Method i-1 are specified in the following way :

if the i-th bit of the choice vector is given and equal ci then move the tuple to the database page indexed by the string b1.. b1.....bi-1 ci bi+1....bm,

otherwise put the tuple on both database pages indexed by strings b1....bi-1 0 bi+1.....bm and b1.....bi-1 1 bi+1.....bm.

In other words, to insert a tuple t, the set of indexing strings is constructed. First i-bits are selected in the way described in the previous section. Then to each of them m-i bits are appended and the copies of t are put on each indexed database page.

The parameter i, called any-depth, can be chosen by the user or be given by a system option.

It should be taken into account that the properties of Method i are changing during file evolution. As long as the depth of the directory is less qr equal than any-depth, full replication is performed.

When a choice vector is chosen then attributes which can assume the value of 'any' should rather not serve as a source of bits for the choice vector, because the result set for unspecified queries and consequently the number of accesses to database pages are growing.

4.4. Modyfing a file with incomplete tuples.

A partial ordering $\preceq$ is defined (interpreted as " t1 is no less precise then t2 ") among the tuples as follows :
t1 $\preceq$ t2 iff t1=(a1,.....an), t2=(b1,.....,b2) and ai=bi or bi=any for all i=1,...,n.

To insert a tuple we simple put it on the database page determined by an any-depth parameter in the way de cribed in previous sections .

The deletion from a file has a tuple t0 as an argument and is defined in two basic ways.
A. Delete tuple (delete no less precize) : delete all tuples t such that t $\preceq$ t0.
B. Delete this tuple (exact delete) : delete only those tuples t such that t=t0.

Example 10.

The request of deleting all tuples of relation R with first attribute equal to 1
may result in deleting

A. +R(1,YELLOW),....,+R(1,GREEN), +R(1,any)
B. +R(1,any).

The deletion of all tuples matching this tuple is not acceptable for it causes unintended loss of information. In the above example

+R(1,YELLOW),....., +R(1,GREEN), +R(1,any),
+R(any,YELLOW),....., +R(any,GREEN), +R(any,any).

Update is implemented as consecutive deletion and insertion in two variants.
1. Delete this tuple(t1),Insert(t2) - to update exactly one tuple.
2. Delete tuple(t1),Insert(t2) - to replace a set of tuples by one tuple.

4.4. Additional aspects of implementation.

Descriptions of all relations (PROLOG procedures) which are stored in the database are in a catalog.
For each relation the catalog contains :
- the unique identifier,
- the cardinality,
- the choice vector,
- the any-depth,
- the current depth of the directory,
- the address of the directory descriptor.

Each relation has its own directory. During the session the catalog resides in core and at the end of the session it is copied back to disk as the header of the database.

During query evaluation or modification preunification, which directly corresponds to unification in PROLOG, is performed on all levels. Only the binding of variables is left to the PROLOG interpreter.

There is a stack of queries which is used to handle of backtracking. Matching tuples are returned to the PROLOG interpreter one by one. The computation continues and, after backtracking, may return to one of the previous queries and request another matching tuple.

The stack for each query contains :

- indexing string which determines the address of last accessed database page;
- location of the last returned tuple on the database page;
- the pattern of indexing string (the string with 1 on the places fixed by specified attributes of query or tuple) which is used to find the next elements of a query result set;
- the local depth of database page during last matching.

A priority mechanism connected with buffer management is used for optimizing number of page accesses. The database page, on which more than one tuple matching a query reside, is kept in a buffer so long as it is possible.

However for a query which required many backtracking, in particular for a nested query , we cannot avoid many disk accesses for the same database page.

Another problem arises when, between two matchings for the same query (before backtracking), the database page on which we found last matching tuple splits. For example it can occur during the checking of integrity constraints for insertion, when a few additional tuples are stored. In this case the order of tuples is changed as an effect of a database page spliting, so the the next tuple is undefined.

Our solution is provisional. We retrieve all matching tuples from both old and new pages. It causes that some of the matching tuples are returned twice to the PROLOG interpreter like in the case of replication of incomplete tuples when the interpreter also receives duplicates and handles them. It seems that duplicate elimination would require quite a lot of additional data structures so we postponed it to the future development of the system.

4.6. Interface with PROLOG.

The interface between the PROLOG interpreter and the support system is very simple. The interpreter sends a request which is fulfilled and, if necessary, the matching clauses are returned one by one. The requests are treated by PROLOG like I/O commands.

The requests are :

Create relation(name, cardinality, any-depth) - adds a description of relation to the catalog and creates a new directory for it. The system creates a rest of a description. (The choice vector may be also a request parameter).

Drop relation(name) - deletes all tuples (if there are any), frees all database and directory pages and removes a relation from catalog.

Insert tuple(relation name, tuple address) - inserts a tuple to database.

Delete tuple(relation name, tuple address) and Delete this tuple(relation name, tuple address) - deletes tuples or tuple in the way described in section 4.4.

Delete all tuples(relation name) - deletes all tuples of an indicated relation.

Give first tuple(relation name, query pattern address, request number) - returns first tuple matching specified query pattern and pointer to the request stack entry which contains description of this request.

Give next tuple(relation name, query pattern address, request number) - returns next tuple matching specified query pattern (generally after backtracking). The pointer to request stack entry doesn't change.

The PROLOG interpreter and support system are working as synchronous processes.

5.Conclusions

Our proposals in section 2.3 and 2.4 demonstrated the conceptual conciseness of PROLOG in dealing with several database problems. The implementation of simple integrity assertions and triggers does not require any extensions of the language. The deductive capabilities of PROLOG are unquestionable. This all makes PROLOG an attractive programming language for an implementor of sophisticated user interfaces for databases. However,

using PROLOG in an actual database application requires further development of database management system mechanisms.

No form of recovery, concurrency control and protection is provided in our system, so it is certainly not a "full-fledged" database management system. It is nevertheless, to our knowledge, the first attempt to handle non-toy PROLOG databases. Other approaches: (Lloyd 82), (Kunifuji 82), (Warren 81) neglected the problems of: efficient secondary storage organization and access, the support for null values and views, and even the simplest integrity checking. We have proposed solutions for the above problems and incorporated the solutions into an actual system. The experience with this system will certainly give further arguments for (or against) the use of PROLOG in the database area.

6. Acknowledgements

We are greatly indebted to Feliks Kluzniak and Stanislaw Szpakowicz for the constant inspiration and help in the development of our system.

7. References.

(ANSI 75) ANSI/X3/SPARC Interim Report, FDT Bulletin of ACM-SIGMOD, 7:2, 1975.

(Blaustein 81) Blaustein,B.T., Enforcing database assertions : techniques and applications. Ph.D. thesis Harvard University, 1981.

(Bowen 81) Bowen,K.A. and Kowalski,R.A., Amalgamating language and metalanguage in logic programming. School of Computer and Information Science, Syracuse University, 4/81, June 1981.

(Chomicki 83) Chomicki,J., Implementing null values. (Submitted for publication).

(Clark 79) Clark,K.L., Negation as failure in : Logic and data bases, Eds. Gallaire,H. and Minker,J. Plenum Press 1979.

(Codd 79) Codd,E.F. Extending the relational data base model to capture more meaning. ACM Transactions on Database Systems, 4,4, 1979.

(Dayal 82) Dayal,U. and Bernstein,P.A., On the correct translation of update operations on relational views. ACM Transactions on Database Systems, 7,3, 1982.

(Eswaran 75) Eswaran,K.P. and Chamberlin,D.D., Functional specifications of a subsystem for database integrity. Proceedings , 1st Int'l Conf.on Very Large Data Bases, 1975.

(Eswaran 76) Eswaran,K.P., Specifications, implementations and interactions of a trigger subsystem in an integreted database system. IBM RJ 1820, August 1976.

(Fagin 79) Fagin,R. at all, Extendible hashing - a fast access method for dynamic files. ACM Transactions on Database Systems, 4,3, 1979.

(Fernandez 81) Fernandez,E.B. and Summers,R.C. and Wood,C., Database security and integrity. Addison-Wesley 1981.

(Kluzniak 83a) Kluzniak,F., PROLOG for SM-4. Technical documentation, Institute of Informatics, Warsaw University 1983.

(Kluzniak 83b) Kluzniak,F. and Szpakowicz,S., SPOQUEL - a Simple PROLOG-Oriented Query Language. Institute of Informatics, Warsaw University (in preparation).

(Koster 74) Koster,C.H.A., Using the CDL compiler-compiler. In : Compiler construction, an advanced course. Eds. Bauer,F.J. and Eickel,J. Lectures notes in computer science 21, Springer-Verlag 1974.

(Kunifuji 82) Kunifuji,S. and Yokota,A., PROLOG and relational databases for fifth generation computer systems. In : Preprints,Workshop on Logical Bases for Databases, Toulouse, France, Dec.1982.

(Lloyd 80) Lloyd,J.W., Optimal partial-match retrieval. BIT 20,1980.

(Lloyd 82) Lloyd,J.W., An introduction to deductive database systems. TR 81/3, Dept.of Computer Science, Univ.of Melbourne, revised 1982.

(Naish 83) Naish,L., Introduction to MU-PROLOG. TR 82/2, Dept.of Computer Science, Univ.of Melbourne, revised 1983.

(Nicolas 82) Nicolas,J.M., Logic for improving integrity checking in relational databases. Acta Informatica 18,1982.

(Paolini 82) Paolini,P. and Zicari,R., Properties of views and their implementation. In : Preprints, Workshop on Logical Bases for Databases, Toulouse, France, Dec.1982.

(Roussel 75) Roussel,Ph., PROLOG, manuel de reference et d'utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, 1975.

(Siklossy 82) Siklossy, L., Updating views : a constructive approach. In : Preprints, Workshop on Logical Bases for Databases, Toulouse, France, Dec.1982.

(Ullman 80) Ullman,J.D., Principles of Database Systems. Computer Science Press, 1980.

(Vassiliou 80) Vassiliou,Y., A formal treatment of imperfect information in database management. TR CSRG-123, Univ.of Toronto, Nov.1980.

(Warren 81) Warren, D.H.D., Efficient processing of interactive relational database queries expressed in logic. Proc.7th.Int'l Conf.on Very Large Data Bases, Cannes, France, Sept.1981.

(Zaniolo 82) Zaniolo,C., Database relations with null values. Proc.ACM Symp.on Principles of Database Systems, 1982.