An External Database for Prolog

José Paulo Leal
Centro de Informática da Universidade do Porto
R. das Taipas 135 / 4000 Porto
Portugal

Abstract

This work describes a disk-resident database for Prolog which uses mechanisms similar to the ones used by the clausal database for recording and retrieving terms. It is intended to be used by applications requiring a flexibility greater than that provided by an interface to traditional database system. There is almost no restriction on the terms stored in the database and the retrieval mechanisms produces terms in the order they were recorded. To enhance the performance of the system, the database organization provides an access mechanism using hash-codes on "key" arguments of the recorded term. The database also provides basic support for multi-user access.

Area : Architectures and Languages / Logic Programming

1 Introduction

Although many Prolog implementations provide mechanisms for interfacing with classic database systems, in some situations those systems are not suited to store the kind of data needed by the applications. This is the case of applications where the data takes the form of complex Prolog terms representing, for instance, expert system rules, natural language grammar rules or dictionary entries.

Such systems are implemented, at least in a first stage, using the clausal database or, if one exists, an internal database. When a great amount of data is needed the choice is usually a relational database system. A certain relationship between the relational model and Prolog is then assumed:

relation	$\longrightarrow$	predicates
tuple	$\longrightarrow$	clause
attribute	$\longrightarrow$	argument

This relationship enables the implementation of relational databases in Prolog. On the other hand, due to some of the characteristics of the relational model the arrows cannot be reversed:

		Relational Model	Prolog
data structure			
	domains	atomic values (and NULLS)	all kinds of terms (including list, functor-terms and variables with multiple occurrences)
	n-ary relations	$n > 0$	$n \geq 0$
		set of unordered tuples	set of ordered tuples
data integrity constrains			
	keys	primary keys cannot be NULL	no restrictions on particular arguments
data manipulation			
		queries return a table	a clause at a time is returned trough backtrack

The use of clauses (or the internal database) to store the data. has, however, the following drawbacks: all the data must be resident in memory, there is no persistence, and no mechanisms for multi-user access are provided.

This work describes an external database, which aims at overcoming the inconvenients mentioned above, for the YAP Prolog system [Damas et al.]. YAP is based on a portable Prolog compiler compatible with C-Prolog. Like the YAP compiler, the external database is written in C and runs at present under the UNIX operating system. It was developed in a SUN3/60 workstation.

The external database began originally as a permanent (i.e. disk resident) version of the internal database. In the sequel, features were included to make multi-user access possible.

No restrictions are imposed on the terms stored in the database. They may contain variables with multiple occurrences, and may have any size or depth. Thus any kind of information representable by a Prolog term may be recorded.

Simultaneous use of the same database file by several Prolog programs is supported through predicates for modifying a term while guaranteeing exclusive access to it so that database consistency can be kept.

It should be stressed that the aim of this work was to implement a tool for Prolog applications that need to store a great amount of data using all the flexibility of the clausal database. It is, by no means, intended to produce a full-fledged relational database system using Prolog as a query language or as an host programming language. Therefore, only the basic aspects of database access and control are addressed. Some functions usually found in conventional database management systems, such as query optimization, integrity constrains, security, and views were totally ignored. They can be implemented, in Prolog, using well known techniques [Parsaye].

2 Conceptual Architecture of the Database

As mentioned in the introduction, the database was originally modeled on the clausal database of Prolog systems, and so it can be seen, in a simplistic way, as an ordered collection of Prolog terms which is accessed and modified through the following kinds of primitives:

`recorded_db(X)`	for querying the database
`erase_db(X)`	for removing a term from the database
`record_db(X)`	for inserting a term a the the end of the database

In clausal database unification is the matching mechanism beetwen clauses and goals. Following the same paradigm the external database is searched for terms unifying with the argument of the `recorded_db(X)` predicate. We also include two other forms of this predicate. They use, as a matching criteria, the term stored in the database being an instance of the argument, and equality of terms modulo variable names. Note also that this predicate is backtrackable and, thus, can be used to access, by backtracking, all the terms matching its argument.

The `erase_db(X)` predicate removes the first term matching its argument from the database. As with the `recorded_db/1` predicate, there are two other forms of this predicate with different matching criteria.

Although the outlined above was a starting point for the design of the conceptual architecture of the external database, it is obvious that, in order to achieve efficiency and multi-user support, same changes were required.

Firstly, most of the applications organize data around classes of terms similar to the notion of *relation* in the conventional database systems. This led us to change our view of the database as an ordered set of terms with the same functor. Since the only Prolog terms with no functor are the atomic ones, they are collected in a "functor-less" relation. This forced us to restrict the arguments of the basic access predicates described above to be non-variables. In order to improve the efficiency of the access for a given relation, the programmer can use the predicate

`relation_usage(Name,Arity,Indices)`

to specify indexing information for the relation name `Name` with arity `Arity`. This does not impose any restriction on the terms used as arguments of the basic access predicates as it only conveys some pragmatic information for speeding up database look up. This predicate can also be used to retrieve information about the indices currently used by a relation (for instance, to be used in query optimization).

Secondly, in certain uses of the database system updating is very frequent. To cater for this in an efficient way, instead of deleting and recording the modified term, the predicate

```
    modify_db(T,UpdateGoal,NewT)
```

was introduced. It replaces a term T with the term NewT after executing the Prolog goal UpdateGoal. Note that this predicate is applied only to the first term matching T for which the update goal succeeds. It fails when no such term is found. A similar predicate is available to modify all terms matching the specification. Again, variations of this predicates, using different matching criteria, are provided.

The database integrity can be kept in multi-user applications by using the `modify_db/3` predicate since it guarantees exclusive access to the term being modified. To see how this predicate works, consider the case where we want to interchange the second argument (the first is the key argument and cannot be changed) of two terms with the main functor `income/2`. This can be achieved through the following code using two nested calls of the `modify_db/3` predicate.

```
modify_db( income(zp,X),
          modify_db(income(pr,Y),true,income(pr,X)),
          income(zp,Y))
```

3 Implementation Decisions

A database with the characteristics described in the previous section must have certain restrictions in the choice of the access mechanisms:

1) Apart from the indexed access that is naturally expected, the database must provide a default search, to guarantee, in all cases, the order of recovery.

2) In the indexed access:

a) - No requirement on the uniqueness of the key (as they are not required for Prolog clauses).

b) - The terms with the same key arguments must be *sorted* by their recording order.

c) - Variables can occur in terms in arguments used as keys. The search method must allow it and garantee that terms are recovered in the correct order.

3) The search method must be compatible with backtracking and consistent when used with it.

The methods used in the first versions of the database are the hash addressing in its basic form, combined with linear search as default. They were chosen for their simplicity and comparative efficiency when used with medium sized stable relations. As we shall see, they meet all the previous specifications.

Other index addressing methods with the characteristics discussed above could be implemented, in particular to deal with large or dynamic relations. A method of the B-Tree family seems to be particularly recommended for this purpose and its implementation is expected in future versions.

The type of applications for which this database was designed will generally need a fast access to the recorded data. With this system we expect to have an access time of the same order of magnitude of the memory resident databases, i.e. databases using clauses or an internal database; at least with small and medium sized relations. To achieve this goal, it is necessary not only to have a fast access method but also to use a caching mechanism to maintain the information in memory as long as possible.

The UNIX file system offers a service of that kind that could partially solve this problem, but we have found it inappropriate for our purposes.

In the remainder of this section the implementation of this topics will be discussed in more detail.

3.1 General access mechanism

The fact that the database must provide an access to terms through unification and must retrieve terms in the order in which they were recorded, precludes, in general, the use of the traditional database access mechanisms. It is obvious that, in general, we must resort to a linear search method to retrieve the terms that match a given term. Since unification is an expensive operation we use a technique known as "pre-unification" that speeds up linear search by avoiding the need to access the complete representation of terms within a relation. This technique, which is also used in the internal database of YAP, and was previously described in [Futo et al.], works as follows.

We associate with each term recorded in the database two codes. One is based on the concatenation of the hash codes of up to the first 8 arguments of the term. The hash code for each argument takes into consideration the type and value of the argument. The second code is a binary number with zeros on positions corresponding to variables.

These codes can be used as a very fast test to decide whether or not two terms are unifiable. For that purpose, the bitwise conjunction of the arguments code of the first term and the variables code of the second term, and vice-versa are compared. The terms can not unify if the results are not the same.

The pre-unification test is used as follows.

For each relation, we keep a linear table at the end of which we add, every time a term is recorded, a pointer to the representation of the term, and the codes described above.

When searching for terms pre-unifying with a given term we compute two similar codes for that term, and make a linear scan of the table for those entries for which the unification test succeeds. Only after the test succeeds we fetch the term from the disk and perform the unification. This process reduces both the number of unifications and accesses to disk.

3.2 Index Access Mechanism

In the current database implementation the index access mechanism is based on hashing. We shall see that this method meets all predefined specifications.

It is trivial to verify that this method accepts more than one term with the same key.

The term insertion order is preserved, provided the terms in each hash-bucket are maintained in the same order in which they were recorded. Note, in passing, that methods closely related to this one for partial-match retrieval, such as those proposed in [Chomicki et al.] , could not be used directly as a general access method, since they would not respect the order in which the terms were recorded, and would require the uniqueness of the key arguments.

Terms with variables in the key arguments have a replica in every bucket. In this way, not only a certain locality of the search - which increases its efficiency - results, but also the appropriate order of retrieval is ensured. It can be assumed that there will not be too many terms with variables in the key arguments, so that the cost of this redundancy is not expected to be very high.

As mentioned in the previous section the user can employ the **relation_usage/3** predicate to specify one or more sets of arguments to be used as (ground) indices. If none is specified by the user the system will use the set consisting of just the first argument.

Of the several indexing sets specified for a relation, we will distinguish one as the preferred indexing set and will refer to the others as the secondary indices.

The tuples of a relation are divided into buckets using an hash function on the arguments specified by the preferred indexing set. In a bucket directory we will find, for each possible hash-code, a pointer to a sequence of disk pages containing the representation of the terms having that particular hash-code. Note that under each hash bucket, the terms are kept in the order in which they were recorded.

A similar organization is used for secondary indices but the information under each bucket consists of pointers to the representation on the terms instead of the terms themselves.

To look up a term we start by checking if, for any of the indexing sets, all the relevant arguments are non-variables. If is not the case we use the general access method previously described.

Then, an hash function is applied to the key arguments in order to compute the bucket's number and the bucket directory is consulted to find the sequence of terms with that particular hash-code. Finally every term in that sequence is checked for a matching with the initial term.

The hash function of a term is computed from the partial hash functions of each key argument. The hash value is the remainder of the division of the sum of those values by the number of buckets. In a numerical argument, the partial hash function result is its value. In all other cases the partial hash function is computed from the string of characters of the argument external representation. The sum of the characters code shifted to the left by their position in the string has been found to be a well distributed hash function for a wide range of applications.

When a new relation is created a certain number of hash-buckets is assigned to it. If the number of tuples of a relation grows so much that the number of pages of an hash-buckets exceeds a pre-defined value – usually 1 – the number of buckets in the bucket directory will be doubled in order to keep the average number of terms in each bucket within reasonable bounds.

3.3 Backtracking

As we have seen in the introduction, the backtracking mechanism will access a database term at a time. The information needed to handle backtracking is maintained in the execution stack: as records (or pointers to them) are clustered by their insertion order it is enough to save in the stack the position of the record (pointer) where the last matching term was found. Actually more information is kept in stack to avoid unnecessary calculations and searches during backtracking.

We had a special concern about the consistence of backtracking when a relation is updated before the next backtrack. A particular problem could arise in the indexed search. Consider the execution of a goal were `recorded_db/1` predicate has succeeded n times and the subsequent insertion of a new term causes an automatic relation restructuring before the next backtracking, possibly moving all terms with the relevant key value to another page. For instance, the execution of the goal

```
:- recorded_db(f(a,X)), record_db(f(b,X)), fail.
```

could lead to an inconsistency of that kind.

To avoid this kind of inconsistency, when a relation restructuring has occurred before the next backtrack, the search will continue in the new page and the first n matching terms will be skipped.

3.4 Buffering Mechanism

To improve the database efficiency and reduce the number of disk accesses, the system uses a pool of buffers as a cache mechanism for disk pages.

All the access to database pages is done through these buffers which are kept in memory as long as possible. When a buffer is needed to access a database page and there is none free, the database releases the least recently used one. This decision is made taking into account both the number of accesses to the pages and the "time" of the last access.

Even with this mechanism a page that is currently buffered may have to be re-read. If the database file is in use by several users and is updated then all other users having a private copy of the updated page must refresh their buffers.

In this situation (multi-user access) the database must be locked to prevent the introduction of inconsistency when different processes are updating the same relation simultaneously. Locks are maintained over file pages.

3.5 Physical organization of the Database

The database is stored in a single file. The file is organized in pages of a fixed size, which is currently 4K.

The first page of the file is used to keep global information. In particular, it contains the following information:

- The chain of free pages.
- The location of the atom table.
- The location of the relation directory.

The atom table contains an hash table, giving, for each possible hash code, the address of the first of the sequence of linked pages where the actual ASCII representation of the atoms with that particular hash code are stored.

The relation directory contains the following information for each relation present in the database:

- The name and arity.
- The location of the chain of pages containing, for every term of the relation recorded in the database, the pre-unification codes for the term and pointers to the actual representation of the term.

- The location, for each indexing set in use for the relation, of the chain of pages containing the hash buckets for the indexing set in question.

Each chain of data pages is used in a linear fashion and can grow as needed. All the data pages provide a field for linking to a continuation page. The actual information stored in a chain, such as the representation of a recorded term, can be split among several continuation pages.

3.6 Representation of terms

Prolog terms are encoded as sequences of 32-bit cell in prefix form. Atomic terms (numbers and atoms) and variables are encoded using only one cell while structured terms (functor-terms and lists) use a sequence of contiguous cells.

In a cell encoding an atomic term as well as in the first cell of a sequence encoding a structured term, the 4 leftmost bits determine the type of the term. The remaining bits, or the following cells, record the term itself.

When encoding an atom the remaining bits record a pointer to an entry in the atom table were the atoms external representation (a string of characters) is kept. The unification process is more efficient using this method and the redundancy of recording repeatedly used names is avoided.

Variable terms are encoded as an offset to their previous occurrence in the sequence of cells, or a special value in their first occurrence. Note that variables will only appear in structured terms, a variable alone cannot be saved in the database.

The rightmost bits of a functor term record its functor - name and arity - and the following cells record its argument in order.

In general a list of terms is implemented as a list of pairs where each pair is a special functor (the dot) with arity 2. For lists with only atomic elements - string lists - is used a special compacted format: the first cell will record the size of the list and the following cells will record the elements in order. The general case has to be kept to allow an efficient unification in all circumstances.

4 Conclusion

The value of this work can only be fully appreciated after a good number of realistic applications having used it, specially those with large knowledge bases or dictionaries - the kind of applications this database was written for.

For the moment, this database was used with SPIRAL, a natural language interface to databases [Filgueiras]. This kind of application need to store two kinds of information: the database itself and the dictionary used for lexical analysis. The former kind of predicates could belong to a relational database as they possess all the characteristics of the relational model. The latter kind uses more complex predicates that do not fit well in the relational model: predicates with linked variables, fields with different kinds of structures, more then one tuple with the same primary key (the same word may have different meanings).

Since the characteristics of this database are very similar to those of the clausal database of Prolog, which was previously used to store all relations, this adaptation was very straightforward.

5 Acknowledgements

The work described in this paper was partially supported by Junta Nacional de Investigação Científica, under contract no. 87.366, and by Instituto Nacional de Investigação Científica.

References

[Chomicki et al.]	J. Chomicki, Wlodzimiers Grudzinski "A Database Support System For Prolog", in L. Moniz Pereira, L. Monteiro, A. Porto, M. Filgueiras (eds.), Proceedings of the Logic Programming Workshop 83, Universidade Nova de Lisboa, 1983.
[Damas et al.]	L. Damas, V. Costa, R. Azevedo, R. Reis, "Yap Reference Manual", Centro de Informática, Universidade do Porto.
[Date]	C. J. Date, "An introduction to Database Systems", Addinson-Wesley Publishing Company, 1986
[Filgueiras]	M. Filgueiras, "Cooperating Rewrite Process for Natural Language Analysis", Jornal of Logic Programming, vol 3 no. 4, 1986.
[Futo et al.]	I. Futo, F. Darvas and P. Szeredi, "The Application Of Prolog to the Development of QA and DBM Systems", Logic and Databases.
[Parsaye]	K. Parsaye, "Database Management, Knowledge Base Management and Expert System Development in Prolog", in L. Moniz Pereira, L. Monteiro, A. Porto, M. Filgueiras (eds.), Proceedings of the Logic Programming Workshop 83, Universidade Nova de Lisboa, 1983.
[Robinson]	J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", JACM, Vol 12, No 1 (January 1965), pp. 23-41.
[Ullman]	J, D. Ullman, "Database Systems", Pitman Publishing Limited, 1980.