Pruning Extensional Predicates in Deductive Databases

Tiago Soares   Ricardo Rocha   Michel Ferreira
DCC-FC & LIACC, University of Porto
Rua do Campo Alegre, 823, 4150-180 Porto, Portugal
{tiagosoares,ricroc,michel}@ncc.up.pt

Abstract

By coupling Logic Programming with Relational Databases, we can combine the higher expressive power of logic with the efficiency and safety of databases in dealing with large amounts of data. An important drawback of such coupled systems is the fact that, usually, the communication architecture between both systems is not *tight* enough to support a completely transparent use of extensionally (or relationally) defined predicates in the logic program. Such an example is the *cut operation*. This operation is used quite often in Prolog programs, both for efficiency and semantic preservation. However, its use to prune choice-points related to database predicates is discouraged in existing coupled systems. Typically, SQL queries result sets are kept outside WAM data structures and the cut implementation leaves the database result set pointer (cursor) open and the result set data structure allocated in memory, instead of closing the cursor and deallocating memory. In this work we focus on the transparent use of the cut operation over database predicates, describing the implementation details in the context of the coupling between the YapTab system and the MySQL RDBMS. Our approach can be generalised to handle not only database predicates but also any predicate that requires generic actions upon cuts.

1 Introduction

The similarities between logic based languages, such as Prolog, and relational databases have long been noted. There is a natural correspondence between relational algebra expressions and Prolog predicates, and between relational tuples and Prolog facts [7]. The main motivation behind a deductive database system, based on the marriage of a logic programming language and a relational database management system (RDBMS), is the combination of the inference capabilities of the logic language with the ultra-efficient data handling of the RDBMS. The implementation of such deductive database systems follows the following four general methods [1, 14, 6]: **(i)** coupling of an existing logic system implementation to an existing RDBMS; **(ii)** extending an existing logic system with some facilities of a RDBMS; **(iii)** extending an existing RDBMS with some features of a logic language; and **(iv)** tightly integrating logic programming techniques with those of RDBMS's.

An example of a tightly integrated implementation is the Aditi system [15], and examples of coupled implementations are the Coral [8] and XSB [10] systems. Regarding coupled systems, the literature usually distinguishes between *tightly* and *loosely* coupled systems. This classification is far from being clear, though. Sometimes the tight or loose adjectives are used regarding the degree of integration from the programming perspective, while sometimes the terms are used regarding the transparency of the use of logic predicates defined by database relations.

The use of explicit SQL queries in logic predicates would characterize a loosely coupled system, while the specification of database queries transparently in logic syntax would characterize a tightly coupled system. The usual meaning of the *tight* and *loose* terms in the computer industry is, however, different. Systems are tightly coupled if they cannot function separately and loosely coupled if they can. Under this definition, the XSB system coupled with a RDBMS results in a loosely coupled system.

The coupling approach, by keeping the deductive engine separated from the RDBMS, offers a number of advantages: deductive capabilities can be used with arbitrary RDBMS's and the deductive system can profit from future and independent developments of the RDBMS; furthermore, there are also no extra overheads for programs that do not access data stored in a relational database. The coupling approach also presents, however, some important drawbacks as compared with integrated systems. The most relevant is the communication overhead to get external data from the database server to be unified with logic goals. This overhead is extremely significant and makes impracticable the use of *relation-level access* (where SQL queries are generated for each logic goal) with all but toy applications. *View-level access* (where relational join operations of logic goals are executed by the RDBMS) very importantly reduces this communication overhead, particularly if it allows the efficient use of the RDBMS indices [4].

Another important drawback of loosely or tightly coupled systems is the fact that the communication architecture between both systems is normally based on an interface layer which lacks important features necessary to improve both the efficiency and the transparent integration of the logic system and the database system. For instance, the application programming interfaces of RDBMS's do not provide adequate mechanisms to send *sets* of tuples back and forth from the logic system to the database server, and this has been shown to be crucial to attain good performance [5].

In addition to efficiency concerns, the existing communication architectures between coupled systems are also not *tight* enough to support a completely transparent use of relationally defined predicates in the logic program. Consider, for example, the *cut operation*. This operation is extremely used in Prolog programs, both for efficiency and semantic preservation. However, its use after a database defined predicate can have undesired effects, with the current interface architectures of coupled systems. The undesired effects can be so significant that systems such as XSB clearly state on the programmers' manual that cut operations should be used very carefully with relationally defined predicates [11]:

> *"The XSB-ODBC interface is limited to using 100 open cursors. When XSB systems use database accesses in a complicated manner, management of open cursors can be a problem due to the tuple-at-a-time access of databases from Prolog, and due to leakage of cursors through cuts and throws. Often, it is more efficient to call the database through set-at-a-time predicates such as findall/3, and then to backtrack through the returned information."*

In this work we focus on the transparent use of the cut operation over database predicates, describing the implementation details in the context of the coupling between the YapTab system [9] and the MySQL RDBMS [16]. YapTab is a tabling system that extends Yap's engine [12] to support tabled evaluation for definite programs. The remainder of the paper is organized as follows. First, we briefly describe the cut semantics of Prolog and the problem arising from its use to prune database predicates. Next, we present how Yap interfaces MySQL through their C API's. Then, we describe the needed extension to the interface architecture in order to deal with pruning for database predicates. At the end, we outline some conclusions.

2 Pruning Database Predicates

Ideally, it should be possible to use predicate facts defined extensionally in database relations exactly as predicate facts defined in a Prolog program. In particular, for the cut operation, it should be possible to prune predicates independently of how they are defined. In this section we show why this is a problem for current coupled systems.

2.1 Cut Semantics

Cut is a system built-in predicate that is represented by the symbol '!'. Its execution results in pruning all the branches to the right of the cut scope branch. The *cut scope branch* starts at the current node and finishes at the node corresponding to the predicate containing the cut.

Figure 1 gives a general overview of cut semantics by illustrating the left to right execution of an example with cuts. The query goal `a(X)` leads the computation to the first alternative of predicate `a/1`, where `!(a)` means a cut with scope node `a`. Predicate `b(X)` is then called and suppose that it succeeds in its first alternative binding `X` with `b1`. Next, `!(a)` gets executed and all the right branches until node `a` are pruned. As a consequence, the nodes for `a` and `b` can be removed.

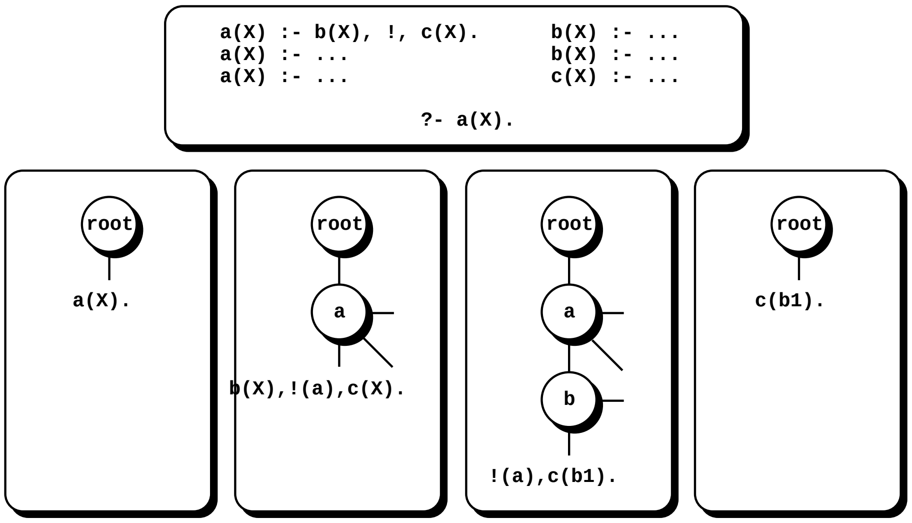

Figure 1: Cut semantics

2.2 Pruning a Query Result Set

In the coupling interface between a logic system and a database system, logic goals are usually translated into SQL queries, which are then sent to the database system. To improve performance, a number of logic goals can be combined on a single SQL query, replacing a relation-level access for what is known as view-level access. The database system then receives the query, processes it, and sends the resulting tuples back to the logic system.

MySQL offers two alternatives for sending these resulting tuples to the client program that generated the query: **(i)** store the set of tuples on a data structure on the database server side and send each tuple *tuple-at-a-time* to the client program; or **(ii)** store the set of tuples on a data structure on the client side sending the all set of tuples at once. Logic systems also have two alternatives to deal with these sets of tuples: **(i)** use them *tuple-at-a-time*, exactly as is done for normal facts; or **(ii)** use them *set-at-a-time*, typically in a Prolog list structure [3]. Building this list for thousands or millions of tuples can lead to memory allocation problems and is very time consuming. Furthermore, the transparent use of facts defined in database relations is lost if they imply a set-at-a-time use.

For tuple-at-a-time, the obvious method of accessing the tuples in the result set of a query is to use the backtracking mechanism, which iteratively increments the database result set pointer (cursor) and fetches the current tuple. Using this tuple-at-a-time access, the deallocation of the data structure holding the result set, whether on the server or on the client side, is performed when the last tuple on the result set has been reached.

The problem is when, during the tuple-at-a-time navigation, a cut operation occurs before reaching the last tuple. If this happens, the result set cannot be deallocated. This can cause a lack of cursors and, more important, a lack of memory due to a number of very large non-deallocated data structures. Consider the example described in Fig. 1 and assume now that predicate `b/1` is a database predicate. The computation of `b(X)` will query the database for the correspondent relation and bind `X` with the first tuple that matches the query. Next, we execute `!(a)` and, as mentioned before, it will disable the action of backtracking for node `b`. The result set with the facts for `b` will remain in memory, although it will never be used.

To solve this problem, we propose an extension to the interface architecture that allows to associate cut procedures with database predicates, in such a way that the system transparently executes them when a cut operation occurs. With this functionality, we can thus use these procedures to deallocate the result set for the pruned database predicates and therefore avoid the problems discussed above. In what follows we briefly describe the Yap interface and then we detail our approach to extend the interface in order to deal with pruning for database predicates.

3 The C Language Interface to Yap Prolog

Like other Prolog Systems, Yap provides an interface for writing predicates in other programming languages, such as C, as external modules. An important feature of this interface is how we can define predicates. Yap distinguishes two kinds of predicates: *deterministic predicates*, which either fail or succeed but are not backtrackable, and *backtrackable predicates*, which can succeed more than once.

Deterministic predicates are implemented as C functions with no arguments which should return zero if the predicate fails and a non-zero value otherwise. They are declared with a call to `YAP_UserCPredicate()`, where the first argument is the name of the predicate, the second the name of the C function implementing the predicate, and the third is the arity of the predicate.

For backtrackable predicates we need two C functions: one to be executed when the predicate is first called, and other to be executed on backtracking to provide (possibly) other solutions. They are similarly declared, but using instead `YAP_UserBackCPredicate()`. When returning the last solution, we should use `YAP_cut_fail()` to denote failure, and `YAP_cut_succeed()` to denote success. The reason for using `YAP_cut_fail()` and `YAP_cut_succeed()` instead of just returning a zero or non-zero value, is that otherwise, when backtracking, our function would be indefinitely called. For a more exhaustive description on how to interface C with Yap please refer to [13].

3.1 Writing Backtrackable Predicates in C

To explain how the C interface works for backtrackable predicates we will use a small example from the interface between Yap and MySQL. We present the `db_row(+ResultSet,?ListOfArgs)` predicate, which fetches tuples from a query result set, tuple-at-a-time through backtracking, and unifies the list in `ListOfArgs` with the current tuple. To do so, first a `yap_mysql.c` module should be created. Next the module should be compiled to a shared object and then loaded under Yap by executing `load_foreign_files([yap_mysql],[],init_predicates)`. After that, the `db_row/2` predicate becomes available. The code for this module is shown next in Fig. 2.

```
#include "Yap/YapInterface.h"                    // header file for the Yap interface to C

void init_predicates() {
  YAP_UserBackCPredicate("db_row", c_db_row, c_db_row, 2, 0);
}

int c_db_row(void) {                                       // db_row: ResultSet -> ListOfArgs
  int i, arity;
  YAP_Term arg_result_set, arg_list_args, head;
  MYSQL_ROW row;
  MYSQL_RES *result_set;

  arg_result_set = YAP_ARG1;
  arg_list_args = YAP_ARG2;
  result_set = (MYSQL_RES *) YAP_IntOfTerm(arg_result_set);
  arity = mysql_num_fields(result_set);
  if ((row = mysql_fetch_row(result_set)) != NULL) {                    // get next tuple
    for (i = 0; i < arity; i++) {
      head = YAP_HeadOfTerm(arg_list_args);
      arg_list_args = YAP_TailOfTerm(arg_list_args);
      if (!YAP_Unify(head, YAP_MkAtomTerm(YAP_LookupAtom(row[i]))))
        return FALSE;
    }
    return TRUE;
  } else {                                                              // no more tuples
    mysql_free_result(result_set);
    YAP_cut_fail();
    return FALSE;
  }
}
```

Figure 2: The C code for the `db_row/2` predicate

Figure 2 shows some of the key aspects about the Yap interface. The include statement makes available the macros for interfacing with Yap. The `init_predicates()` procedure tells Yap, by calling `YAP_UserBackCPredicate()`, the predicate defined in the module. The function `c_db_row()` is the implementation in C of the desired predicate. We can define a function for the first time the predicate is called and another for calls via backtracking. In this example the same function is used for both calls. Note that this function has no arguments even though the predicate being defined has two. In fact the arguments of a Prolog predicate written in C are accessed through the macros `YAP_ARG1`, ..., `YAP_ARG16` or with `YAP_A(N)` where `N` is the argument number.

The `c_db_row()` function starts by converting the first argument (`YAP_ARG1`) to the correspondent pointer to the query result set (`MYSQL_RES *`). The conversion is done by the `YAP_IntOfTerm()` macro. It then fetches a tuple from this result set, through `mysql_fetch_row()`, and checks if the last tuple as been already reached. If not, it calls `YAP_Unify()` to attempt the unification of values in each attribute of the tuple (`row[i]`) with the respective elements in `arg_list_args`. If unification fails it returns `FALSE`, otherwise returns `TRUE`. On the other hand, if the last tuple has been already reached, it deallocates the result set, as mentioned before, calls `YAP_cut_fail()` and returns `FALSE`.

For simplicity of presentation, we omitted type checking procedures over MySQL attributes that must be done to convert each attribute to the appropriate term in Yap. For some predicates it is also useful to preserve some data structures across different backtracking calls. This can be done by calling `YAP_PRESERVE_DATA()` to associate such space and by calling `YAP_PRESERVED_DATA()` to get access to it later. With these two macros we can easily share information between backtracking. This example does not need this preservation, as the cursor is maintained in the result set structure.

3.2 The Yap Implementation of Backtrackable Predicates

In Yap, a backtrackable predicate is compiled using two WAM-like instructions, `try_userc` and `retry_userc`, as follows:

```
try_userc c_first arity extra_space
retry_userc c_back arity extra_space
```

Both instruction have three arguments: the `c_first` and `c_back` arguments are pointers to the C functions associated with the backtrackable predicate, `arity` is the arity of the predicate, and `extra_space` is the memory space used by the `YAP_PRESERVE_DATA()` and `YAP_PRESERVED_DATA()` macros.

When Yap executes a `try_userc` instruction it uses the choice point stack to reserve as much memory as given by the `extra_space` argument, next it allocates and initializes a new choice point (see Fig. 3), and then it executes the C function pointed by the `c_first` argument. Later, if the computation backtracks to such choice point, the `retry_userc` instruction gets loaded from the `CP_AP` choice point field and the C function pointed by the `c_back` argument is then executed.

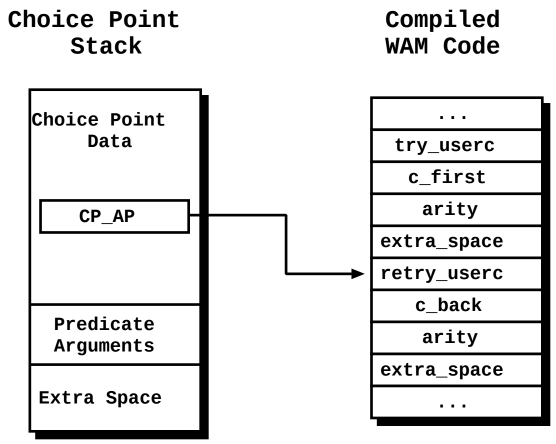

Figure 3: Yap support for backtrackable predicates

In order to repeatedly execute the same `c_back` function when backtracking to this choice point, the `retry_userc` instruction maintains the `CP_AP` field pointing to itself. This is the reason why we should use `YAP_cut_succeed()` or `YAP_cut_fail()` when returning the last solution for the predicate, as otherwise the choice point will remain in the choice point stack and the `c_back` function will be indefinitely called.

The execution of the `YAP_PRESERVE_DATA()` and `YAP_PRESERVED_DATA()` macros in the C functions corresponds to calculate the starting position of the reserved space associated with the `extra_space` argument. For both functions, this is the address of the current choice point pointer plus its size and the arity of the predicate.

4 Extending the Yap C Interface to Deal with Pruning

In this section, we discuss how we can solve the problem of pruning database predicates. We consider two different approaches. In the first approach it is the user that is responsible to specify an action to be executed just before a cut operation over a database predicate. The second approach is our original proposal where the system transparently executes a cut procedure when a cut operation occurs. As we will see, this approach can be used, not only, to handle database predicates but also to handle any predicate that requires a generic action over a cut.

4.1 Handling Cuts Explicitly

In this approach it is the user that has to explicitly call a predicate to be executed when a cut operation is performed over special predicates. The idea is as follows: before executing a cut operation that potentially prunes over databases predicates, the user must execute a predicate that releases beforehand the result sets for the predicates to be pruned.

Consider again the example from Fig. 1 and assume that `b(X)` is a database predicate. We might think of a simple solution: every time a database predicate is first called, we store the pointer for its result set in a stack frame. Then, we could implement a C predicate, `db_free_result/0` for example, that deallocates the result set in the top of the stack. Having this, we could adapt the code for the first clause of predicate `a/1` to:

```
a(X) :- b(X), db_free_result, !, c(X).
```

To use this approach, the user must be careful to always include a call to `db_free_result/0` before a cut operator. A possible alternative, would be to define a new predicate, `db_cut/0` for example, and use it to replace the `db_free_result/0` and the cut.

```
a(X) :- b(X), db_cut, c(X).

db_cut :- db_free_result, !.
```

But, unfortunately, this would not work because the cut operator in the `db_cut/0` definition will not prune the choice point for `b(X)` in the first clause of `a/1`. Another problem with the `db_free_result/0` approach occurs if we call more than one database predicates before a cut. Consider the following definition for the predicate `a/1`, where `b1/1` and `b2/1` are database predicates.

```
a(X) :- b1(X), b2(Y), db_free_result, !, c(X).
```

The `db_free_result/0` will only deallocate the result set for `b2/1`, leaving the result set for `b1/1` pending. A possible solution for this problem is to *mark* beforehand the range of predicates to protect. We can thus implement a C predicate, `db_cut_mark/0` for example, that *marks* were to cut to, and change the `db_cut/0` predicate to call recursively the `db_free_result/0` predicate for all the result sets within the mark left by the last `db_cut_mark/0` predicate.

```
a(X) :- db_cut_mark, b1(X), b2(Y), db_cut, !, c(X).
```

This final solution solves the problem of cursor leaks when pruning database predicates. However, in this approach the transparency in the use of relationally defined predicates is lost. The user has to explicitly include extra annotations in the code to control the range of predicates to cut.

4.2 Handling Cuts Transparently

We next present our approach to handle cuts transparently. As we shall see, this requires minor changes to the Yap interface and engine. First, we have extended the procedure used to declare backtrackable predicates, `YAP_UserBackCPredicate()`, to include an extra C function. Remember that for backtrackable predicates we used two C functions: one to be executed when the predicate is first called, and another to be executed upon backtracking. The extra function is where the user should declare the function to be executed in case of a cut, which for database predicates will involve the deallocation of the result set. Declaring and implementing this extra function is the only thing we need to do to take advantage of this approach. Thus, from the user's point of view, dealing with standard predicates or relationally defined predicates is then equivalent.

With this extra C function, the compiled code for a backtrackable predicate now includes a new WAM-like instruction, `cut_userc`, which is used to store the pointer to the extra C function, the `c_cut` argument.

```
try_userc c_first  arity extra_space
retry_userc c_back arity extra_space
cut_userc c_cut arity extra_space
```

When Yap executes a `try_userc` instruction it now also allocates space for a cut frame data structure (see Fig. 4). This data structure includes two fields: `CF_previous` is a pointer to the previous cut frame on stack and `CF_inst` is a pointer to the `cut_userc` instruction in the compiled code for the predicate. A top cut frame variable, `TOP_CF`, always points to the youngest cut frame on stack. Frames form a linked list through the `CF_previous` field.

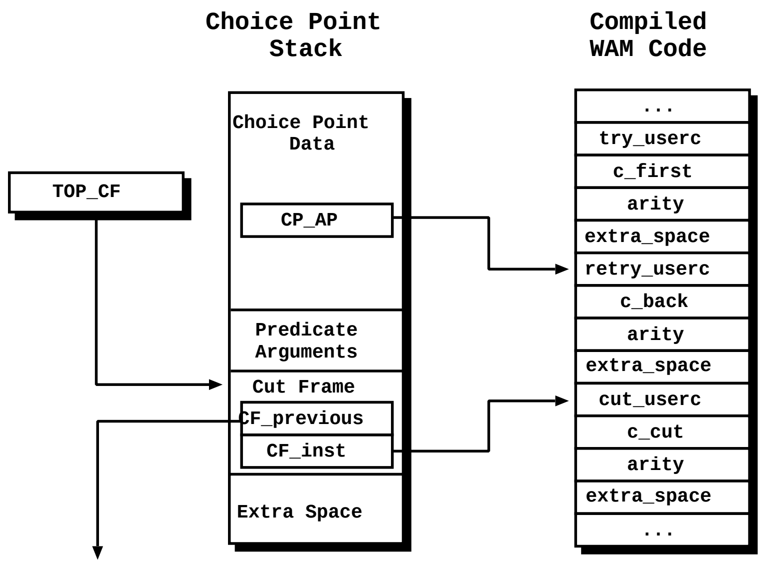

Figure 4: Yap support for handling cuts transparently with backtrackable predicates

By putting the cut frame structure below the associated choice point, we can easily detect the predicates being pruned when a cut operation occurs. To do so, we extended the implementation of the cut operation to start by executing a new `userc_cut_check()` procedure (see Fig. 5). Remember that a cut operation receives as argument the choice point to cut to. Thus, starting from the `TOP_CF` variable and going through the cut frames, we can check if a cut frame will be pruned. If so, we load the `cut_userc` instruction stored in the correspondent `CF_inst` field in order to execute the cut function pointed by the c_cut argument. In fact, in our implementation, the cut_userc instruction is never executed. As an optimization, the CF_inst field points directly to the cut function.

```
void userc_cut_check(choiceptr cp_to_cut_to) {
  while (TOP_CF < cp_to_cut_to) {
    execute_c_function(TOP_CF->CF_inst);
    TOP_CF = TOP_CF->CF_previous;
  }
  return;
}
```

Figure 5: The pseudo-code for the userc_cut_check() procedure

The process described above is done before executing the original code for the cut instruction, that is, before updating the global registry B (pointer to the current choice point on stack). This is important to prevent the following situation. If the cut function executes a YapCallProlog() macro to call the Prolog engine from C, this might has the side-effect of allocating new choice points on stack. Thus, if we had updated the B register beforehand, we will potentially overwrite the cut frames stored in the pruned choice points and avoid the possibility of executing the correspondent cut functions.

As a final remark, note that the cut function can also call the YAP_PRESERVED_DATA() macro to access the data store in the extra space. We thus need to access the extra space from the cut frames. This is why we store the cut frames above the extra space. The starting address of the extra space is thus obtained by adding to the pointer of the current cut frame its size.

To show how the extended interface can be used to handle cuts transparently, we next present in Fig. 6 the code for generalising the db_row/2 predicate to perform the cursor closing upon a cut.

First, we need to define the function to be executed when a cut operation occurs. An important observation is that this function will be called from the cut instruction, and thus it will not be able to access the Prolog arguments, YAP_ARG1 and YAP_ARG2, as described for the c_db_row() function. However, we need to access the pointer to the correspondent result set in order to deallocate it. To solve this, we can use the YAP_PRESERVE_DATA() macro to preserve the pointer to the result set. As this only needs to be done when the predicate is first called, we defined a different function for this case. The YAP_UserBackCPredicate() macro was thus changed to include a cut function, c_db_row_cut(), and to use a different function when the predicate is first called, c_db_row_first(). The c_db_row() function is the same as before (see Fig. 2). The last argument of the YAP_UserBackCPredicate() macro defines the size of the extra space for the YAP_PRESERVE_DATA() and YAP_PRESERVED_DATA() macros.

The c_db_row_first() function is an extension of the c_db_row() function. The only difference is that it uses the YAP_PRESERVE_DATA() macro to store the pointer to the given result set in the extra space for the current choice point. On the other hand, the c_db_row_cut() function uses the YAP_PRESERVED_DATA() macro to be able to deallocate the result set when a cut operation occurs. With these two small extensions, the db_row/2 predicate is now protect against cuts and can be safely pruned by further cut operations.

5 Concluding Remarks

We discussed the problem of pruning database predicates in logic programming with relational database coupled systems. The existing communication architectures between coupled systems

```
void init_predicates() {
  YAP_UserBackCPredicate("db_row", c_db_row_first, c_db_row,
                         c_db_row_cut, 2, sizeof(MYSQL_RES *));
}

int c_db_row_first(void) {
  MYSQL_RES **extra_space;
  ...                                          // the same as for the c_db_row function
  result_set = (MYSQL_RES *) YAP_IntOfTerm(arg_result_set);
  YAP_PRESERVE_DATA(extra_space, MYSQL_RES *);             // initialize the extra space
  *extra_space = result_set;                         // store the pointer to the result set
  ...                                          // the same as for the c_db_row function
}

int c_db_row(void) {
  ...                                                                 // the same as before
}

void c_db_row_cut(void) {
  MYSQL_RES **extra_space, *result_set;

  YAP_PRESERVED_DATA(extra_space, MYSQL_RES *);
  result_set = *extra_space;                           // get the pointer to the result set
  mysql_free_result(result_set);
  return;
}
```

Figure 6: The C code for protecting the db_row/2 predicate against cuts

either do not handle cuts or rely on the user to explicitly protect the database predicates from potential cut operations. This is a relevant problem as the existing coupling architectures between a logic system and a RDBMS do not deallocate result sets upon a cut operation over and extensional predicates. We tested a simple program were a cut was executed over a large database extensional predicate, using XSB and Ciao Prolog [2] coupled to MySQL and we rapidly reached memory overflow because of non-deallocated result sets.

In this work, we proposed a new approach where pruning operators can be used independently of the predicates being pruned, whether they are defined extensionally in database relations or as Prolog facts. In our proposal the Prolog engine transparently executes a user-defined function when a cut operation occurs. To do so, we have extended the Yap interface to include an extra C function when declaring backtrackable predicates. This extra function is where the user defines the actions to be executed when a cut operation prunes the predicate. The Prolog engine then transparently executes the function when a cut operation prunes over the predicate. Thus, from the user's point of view, it only needs to declare that function once when defining the predicate. This approach is applicable to any predicate that requires protection against pruning.

Acknowledgements

This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

[1] M. Brodie and M. Jarke. On Integrating Logic Programming and Databases. In *Expert Database Workshop*, pages 191–207. Benjamin Cummings, 1984.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. Lpez, and G. Puebla. *Ciao Prolog System Manual.* Available from `http://clip.dia.fi.upm.es/Software/Ciao`.

[3] C. Draxler. Accessing Relational and NF^2 Databases Through Database Set Predicates. In *UK Annual Conference on Logic Programming*, Workshops in Computing, pages 156–173. Springer Verlag, 1992.

[4] M. Ferreira, R. Rocha, and S. Silva. Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases. In *Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 71–82, 2004.

[5] J. Freire. Practical Problems in Coupling Deductive Engines with Relational Databases. In *International Workshop on Knowledge Representation meets Databases*, 1998.

[6] F. Maier, D. Nute, W. Potter, J. Wang, M. J. Twery, H. M. Rauscher, P. Knopp, S. Thomasma, M. Dass, and H. Uchiyama. PROLOG/RDBMS Integration in the NED Intelligent Information System. In *Confederated International Conferences DOA, CoopIS and ODBASE*, volume 2519 of *LNCS*, page 528. Springer-Verlag, 2002.

[7] K. Parsaye. Logic Programming and Relational Databases. *IEEE Database Engineering Bulletin*, 6(4):20–29, 1983.

[8] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: A Deductive Database Programming Language. In *International Conference on Very Large Data Bases*. Morgan Kaufmann, 1992.

[9] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[10] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[11] K. Sagonas, D. S. Warren, T. Swift, P. Rao, S. Dawson, J. Freire, E. Johnson, B. Cui, M. Kifer, B. Demoen, and L. F. Castro. *XSB Programmers' Manual.* Available from `http://xsb.sourceforge.net`.

[12] V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 261–267. Springer-Verlag, 1999.

[13] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual.* Available from `http://www.ncc.up.pt/~vsc/Yap`.

[14] P. Singleton and P. Brereton. Storage and Retrieval of First-order Terms Using a Relational Database. In *British National Conference on Databases*, volume 696 of *LNCS*, pages 199–219. Springer-Verlag, 1993.

[15] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask, and J. Harland. The Aditi Deductive Database System. Technical Report 93/10, School of Information Technology and Electrical Engineering, Univ. of Melbourne, 1993.

[16] M. Widenius and D. Axmark. *MySQL Reference Manual: Documentation from the Source.* O'Reilly Community Press, 2002.