Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases

Michel Ferreira Ricardo Rocha Sabrina Silva
DCC-FC & LIACC, University of Porto
Rua do Campo Alegre, 823, 4150-180 Porto, Portugal
{michel,ricroc}@ncc.up.pt ssilva@mat.ua.pt

Abstract

Logic programming and relational databases have common foundations based on First Order Logic. By coupling both paradigms, we can combine the efficiency and safety of databases in dealing with large amounts of data with the higher expressive power of logic and, thus, build more powerful systems. Although much work has been developed and described in the area of logic programming and relational databases, there are very few references which regard implementation alternatives in coupling a logic system with a relational database. In this work, we study and evaluate the impact of using different approaches for coupling the Yap Prolog system with the MySQL relational database system. Our results show that indexing and view level transformations are fundamental to achieve scalability.

1 Introduction

Logic programming is a programming paradigm based on Horn Clause Logic, a subset of First Order Logic. The axiomatic knowledge of a logic program can be represented *extensionally* in the form of facts, and *intensionally* in the form of rules. Program execution tries to prove theorems (*goals*) and if a proof succeeds the variable bindings are returned as a solution. Relational databases can also be considered as a simpler First Order Logic model [4]. The axiomatic knowledge is now only represented extensionally in the form of database relations and the theorems to be proved correspond to (SQL) *queries*.

There are two main differences between a logic programming system and a relational database model. A first difference is the evaluation mechanism which is employed in logic systems and in relational database systems. Logic systems, such as Prolog, are based on a *tuple-oriented* evaluation that uses unification to bind variables with atomic values that correspond to an attribute of a *single tuple*. On the other hand, the relational model uses a *set-oriented* evaluation mechanism. The result of applying a relational algebra operation, such as projection, selection or join, to a relation is also a relation, which is a *set of tuples*. A second difference, is the expressive power of each language. While every relational operator can be represented as a logic clause, the inverse does not happen. Recursive rules cannot be expressed as a sequence of relational operators. Thus the expressive power of Horn clause systems is greater than that of the relational database model.

Combining logic with relational databases would provide the efficiency and safety of database systems in dealing with large amounts of data with the higher expressive power of logic systems. This combination aims at representing the extensional knowledge through database relations and the intensional knowledge through logic rules. A major problem when combining both is the efficient evaluation of queries written in logic involving database relations.

In the specific field of deductive databases [7], a restriction of logic programming, Datalog [14], is commonly used as the query language. Datalog encapsulates the set-at-a-time evaluation strategy and imposes a first normal form compliance to the attributes of predicates associated to database relations. Datalog queries are evaluated by combining top-down goal orientation with bottom-up redundant computation checking. Redundant computations are resolved using two main approaches: the magic-sets rewriting technique [1] and tabling [6], a technique of memoisation successfully implemented in XSB Prolog [11], the most well known tabling Prolog system. A tabling engine largely based on the original ideas of XSB is also available in the Yap Prolog system [9].

The existing prototypes which combine logic with relational databases, such as LDL [13], CORAL [8], Aditi [15] and XSB [12], can be classified in two main categories: integrated systems and coupled systems. Coupled systems, such as CORAL and XSB, have the advantage of keeping the deductive engine and the relational database management system separate. The interface between the two systems in done by translating the query, or parts of the query, written in logic to the language understood by the database system, SQL.

Although much work has been done in these systems, there are no references in the literature that we know that actually discuss and compare the several implementation strategies that may be used in the coupling of a logic system and a relational database. Our goal in this work is to study and evaluate the impact of using some of these strategies. We consider three main approaches: **(i)** asserting database tuples as Prolog facts; **(ii)** accessing database tuples through backtracking; and **(iii)** transferring unification to the database engine. We present a detailed step-by-step description of each approach and for that we use Yap Prolog [2] and MySQL [16] as the base systems for implementing them. Despite the fact that we have chosen these particular systems, we believe that our implementation and results will be of interest for others that intend to couple similar systems.

The remainder of the paper is organized as follows. First, we briefly introduce the set of development tools used. Next, we describe the three alternative approaches. We then evaluate the performance of each approach and finalize by outlining some concluding remarks.

2 Development Tools

To implement the interface between the Yap Prolog and the MySQL systems we took advantage of their client libraries that allow us to write external modules in the C language. To optimize the translation of queries between both systems we used the Prolog to SQL compiler written by Draxler [3]. We next briefly describe the main assets of each tool.

2.1 The C Language interface to Yap Prolog

As many other Prolog systems, Yap provides an interface for writing predicates in other programming languages, such as C, as external modules. We will use a small example to briefly explain how it works. Assume that the user requires a predicate `my_random(N)` to unify `N` with a random number. To do so, first a `my_rand.c` module with the following C code should be create.

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

void init_predicates() {
   YAP_UserCPredicate("my_random",c_my_random,1);
}

int c_my_random(void) {
   YAP_Term number = YAP_MkIntTerm(rand());
   return(YAP_Unify(YAP_ARG1,number));
}
```

Next the module should be compiled to a shared object and then loaded under Yap by calling the `load_foreign_files()` routine. After that, each call to `my_random(N)` will unify `N` with a random number. Despite its small size, the example shows the key aspects about the Yap interface. The include statement makes available the macros for interfacing with Yap. The `init_predicates()` procedure tells Yap the predicates being defined in the module. The function `c_my_random()` is the implementation of the desired predicate. Note that it has no arguments even though the predicate being defined has one. 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. In our example, the function uses just one local variable of type `YAP_Term`, the type used for holding Yap terms, where the integer returned by the standard Unix function `rand()` is stored as an integer term (the conversion is done by `YAP_MkIntTerm()`). Then it calls `YAP_Unify()`, to attempt the unification with `YAP_ARG1`, and returns an integer denoting success or failure.

Table 1 lists the complete set of the available primitives to test, construct and destruct Yap terms. Terms, from the C point of view, can be classified as: *uninstantiated variables*, *instantiated variables*, *integers*, *floating-point numbers* (floats), *atoms* (symbolic constants), *pairs*, and *compound terms*. Integers, floats and atoms are respectively denoted by the primitives `YAP_Int`, `YAP_flt` and `YAP_Atom`. A pair is a term which consists of a tuple of two terms, designated as the *head* and the *tail* of the term. Pairs are most often used to build lists. A compound term consists of a *functor* and a sequence of terms with length equal to the arity of the functor. A functor, denoted in C by `YAP_Functor`, consists of an atom (functor name) and an integer (functor arity).

Term	**Test**	**Construct**	**Destruct**
uninst var	YAP_IsVarTerm()	YAP_MkVarTerm()	(none)
inst var	YAP_NonVarTerm()		
integer	YAP_IsIntTerm()	YAP_MkIntTerm()	YAP_IntOfTerm()
float	YAP_IsFloatTerm()	YAP_MkFloatTerm()	YAP_FloatOfTerm()

atom	YAP_IsAtomTerm()	YAP_MkAtomTerm()
		YAP_LookupAtom()
			YAP_AtomOfTerm()
			YAP_AtomName()

pair	YAP_IsPairTerm()	YAP_MkNewPairTerm()
		YAP_MkPairTerm()
			YAP_HeadOfTerm()
			YAP_TailOfTerm()

compound term	YAP_IsApplTerm()	YAP_MkNewApplTerm()
		YAP_MkApplTerm()
			YAP_ArgOfTerm()
			YAP_FunctorOfTerm()

		YAP_MkFunctor()	YAP_NameOfFunctor()
			YAP_ArityOfFunctor()

Table 1: Primitives for manipulating Yap terms

Building interesting modules cannot be accomplished without two extra functionalities. One is to call the Prolog interpreter from C. To do so, first we must construct a Prolog goal `G`, and then it is sufficient to perform `YapCallProlog(G)`. The result will be `FALSE`, if the goal failed, or `TRUE` otherwise. When this is the case, the variables in `G` will store the values they have been unified with. The other interesting functionality is how we can define predicates. Yap distinguishes two kinds of predicates: *deterministic predicates*, which either fail or succeed but are not backtrackable, like the one in our module; 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 [2].

2.2 The MySQL C API

MySQL provides a client library written in C for writing client programs that access MySQL databases. This library defines an application programming interface that includes the following facilities: connection management procedures, to establish and terminate sessions with a server; procedures to construct, send and process the results of queries; and error handling procedures.

Usually, the main purpose of a client program that uses the MySQL C API is to establish a connection to a database server in order to process a set of queries. Thus, in general, the code skeleton of these programs is as follows.

```
#include <mysql.h>    // header file for the MySQL C API

int main() {
   MYSQL *conn;    // connection handler
   MYSQL_RES *res_set;    // result set
   MYSQL_ROW row;    // row contents
   char *query;    // SQL query string
   conn = mysql_init(...);   // obtain and initialize a connection handler
   mysql_real_connect(conn, ...);   // establish a connection to a server
   while(...) {
      query = ...;    // construct the query
      mysql_query(conn, query);    // issue the query for execution
      res_set = mysql_store_result(conn);    // generate the result set
      while ((row = mysql_fetch_row(res_set)) != NULL) {   // fetch a row
         ...   // do something with row contents
      }
      mysql_free_result(res_set);   // deallocate result set
   }
   mysql_close(conn);   // terminate the connection
}
```

Initially, we allocate a connection handler (represented by the `MYSQL` data type) and try to establish a connection to the desired server. This is done by calling the `mysql_real_connect()` procedure which includes, among others, arguments to define the name of the host to connect to, the database to use, and the name and password of the user trying to connect. Next, we communicate (possibly many times) with the server to process a single query or several queries. At last, we terminate the connection.

Processing a query involves the following steps: **(i)** construct the query; **(ii)** send the query to the server for execution; and **(iii)** handle the result set (represented by the `MYSQL_RES` data type). The result set includes the data values for the rows and also meta-data about the rows, such as the column names and types, the data values lengths, the number of rows and columns, etc.

Handling the result set also involves three steps: **(i)** generate the result set; **(ii)** fetch each row (represented by the `MYSQL_ROW` data type) of the result set to do something with it; and **(iii)** deallocate the result set. Note that each row is implemented as a pointer to an array of strings representing the values for each column in the row. Thus, when treating a value as, for instance, a numeric type, we need to convert the string beforehand.

In our example, we used the `mysql_store_result()` procedure to generate the result set. An alternative is to use the `mysql_use_result()` procedure. They are similar in that both take a connection handler and return a result set, but their implementations are quite different. The `mysql_store_result()` fetches the rows from the server and stores them in the client. Subsequent calls to `mysql_fetch_row()` simply return a row from the data structure that already holds the result set. On the other hand, `mysql_use_result()` does not fetch any rows itself. It simply initiates a row-by-row communication that must be completed by calling `mysql_fetch_row()` for each row. `mysql_store_result()` has higher memory and processing requirements because the entire result set is maintained in the client. `mysql_use_result()` only requires space to a single row at a time, and this can be faster because no complex data structures need to be setting up or handled. On the other hand, `mysql_use_result()` places a great burden on the server, which must hold rows of the result until the client fetches them all. For a complete description on these topics and how to take fully advantage of the MySQL C API please refer to [16].

2.3 Prolog to SQL Compiler

The interface between Prolog programs and database management systems is normally done via the SQL language. A particular Prolog predicate is assigned to a given relation in a database and its facts are made available through the tuples returned by a SQL query. Normally, it is also possible to write Prolog predicates which define *views* over one or more relations. With some restrictions, these views can also be translated to a single SQL query.

This Prolog to SQL translation as been well describe in the literature [5]. An important implementation of a generic Prolog to SQL compiler is the work done by Draxler [3]. It includes the translation of conjunctions, disjunctions and negation of goals, and also of higher-order constructs, such as grouping and sorting. Another important aspect of this work is the notion of *database set predicates*, which allows embedding the set-oriented evaluation of database systems into the standard tuple-oriented evaluation of Prolog, using Prolog itself to navigate this set structure.

Draxler's Prolog to SQL compiler defines a `translate/3` predicate, where the database access language is defined to be a restricted sublanguage of Prolog equivalent in expressive power to relational calculus (no recursion is allowed). The first argument to `translate/3` defines the projection term of the database access request, while the second argument defines the database goal which expresses the query. The third argument is used to return the correspondent SQL select expression. Because this compiler is entirely written in Prolog it is easily integrated in the pre-processing phase of Prolog compilers.

3 Coupling Approaches

In this section, we present and discuss our three alternative approaches for coupling logic programming with relational databases. To develop running examples of each approach, we used Yap and MySQL as the base systems, and we took advantage of their client libraries to implement the interface level. Figure 1 shows how we structured the interface between both systems.

The `yap2mysql.c` is our main module. It defines the low-level communication predicates and it uses the Yap and MySQL interfaces to the C language to implement them. The `sqlcompiler.pl` is Draxler's Prolog to SQL compiler. It will be necessary to implement the third approach. The `yap2mysql.pl` is the Prolog module that the user should interact with. It defines the high-level predicates to be used and abstracts the existence of the other modules. After consulting the `yap2mysql.pl` module, the user starts by calling the `db_open/5` predicate to define a connection to a database server. Then, it calls `db_import/3` to map database relations into Prolog predicates. We also allow, on the third approach, the definition of database views based on these predicates

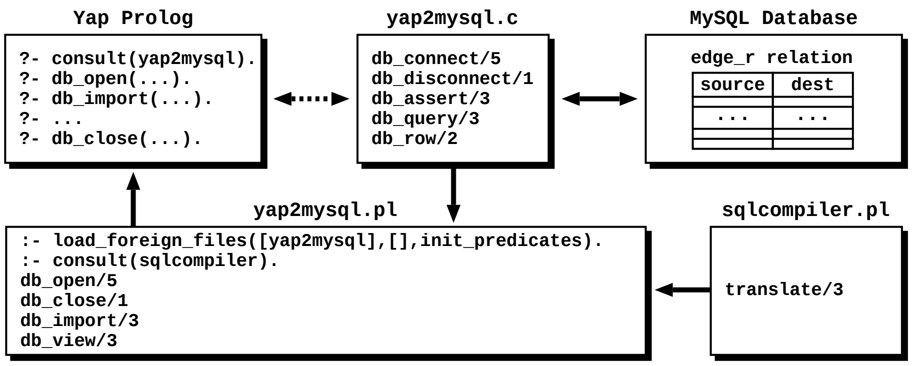

Figure 1: Interface layout

by the use of db_view/3. Next, it uses the mapped predicates to process query goals and, at last, it calls db_close/1 to terminate the session with the database.

In order to allow the user to define multiple connections we use the db_open/5 and db_close/1 predicates to abstract the low-level predicates db_connect/5 and db_disconnect/1 implemented in the yap2mysql.c module.

```
db_open(Host,User,Passwd,Database,ConnName) :-
   db_connect(Host,User,Passwd,Database,ConnHandler),
   set_value(ConnName,ConnHandler).

db_close(ConnName) :-
   get_value(ConnName,ConnHandler),
   db_disconnect(ConnHandler).
```

As we will see, with this module structure, we only need to change the way we define the db_import/3 predicate to implement each approach. In what follows we will use a MySQL relation, edge_r, with two attributes, source and dest, where each tuple represents an edge of a directed graph. The Prolog predicate associated with this relation will be referred as edge/2.

3.1 Asserting Database Tuples as Prolog Facts

A first approach for mapping database relations into Prolog predicates is to assert the complete set of tuples in a relation as Prolog facts. To do so, we only need to connect to the database once and fetch the complete set of tuples. After that, we can simply use the asserted facts as usual. This approach minimizes the number of database communications and can benefit from the Prolog indexing mechanism to optimize certain subgoal calls. On the other hand, it has higher memory requirements because it duplicates the entire set of tuples in the database as Prolog facts. Moreover, real time modifications to the database done by others are not visible to the Prolog system. Even Prolog modifications to the set of asserted tuples can be difficult to synchronize with the database.

To implement this approach we use the following db_import/3 definition.

```
db_import(RelationName,PredName,ConnName):-
   get_value(ConnName,ConnHandler),
   db_assert(ConnHandler,RelationName,PredName).
```

The c_db_assert() procedure implements the db_assert/3 predicate. First, it constructs a 'SELECT * FROM <RelationName>' query in order to fetch the complete set of tuples. Then, for each row, it calls the Prolog interpreter to assert it as a Prolog fact. To do so it constructs Prolog terms of the form 'assert(f_pred(t_args[0],...,t_args[arity-1]))', where f_pred is the predicate name for the asserted facts and t_args[] are the data values for each row.

```
int c_db_assert(void) {
   ...   // auxiliary variables
   YAP_Functor f_pred, f_assert;
   YAP_Term t_pred, *t_args, t_assert;
   MYSQL *conn = (MYSQL *) YAP_IntOfTerm(YAP_ARG1);
   sprintf(query,"SELECT * FROM %s",YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2)));
   mysql_query(conn, query);
   res_set = mysql_store_result(conn);
   arity = mysql_num_fields(res_set);   // get the number of column fields
   f_pred = YAP_MkFunctor(YAP_AtomOfTerm(YAP_ARG3), arity);
   f_assert = YAP_MkFunctor(YAP_LookupAtom("assert"), 1);
   while ((row = mysql_fetch_row(res_set)) != NULL) {
      for (i = 0; i < arity; i++) {   // test each column data type to ...
         ...; t_args[i] = YAP_Mk...(row[i]); ...;   // ... construct the appropriate term
      }
      t_pred = YAP_MkApplTerm(f_pred, arity, t_args);
      t_assert = YAP_MkApplTerm(f_assert, 1, &t_pred);
      YAP_CallProlog(t_assert);   // assert the row as a Prolog fact
   }
   mysql_free_result(res_set);
   return TRUE;
}
```

3.2 Accessing Database Tuples Through Backtracking

The next approach takes advantage of the Prolog backtracking mechanism to access the database tuples. For that, when mapping a database relation into a Prolog predicate it uses the Yap interface functionality that allows defining backtrackable predicates, in such a way that every time the computation backtracks to such predicates, the tuples in the database are fetched one-at-a-time.

With this approach, we concentrate all the data in a single repository. This simplifies its manipulation and allows us to see real time modifications done by others. Moreover, this also minimizes memory requirements. Note however that if we use `mysql_store_result()` to generate the result set, we will duplicate the entire set of tuples on the client side. On the other hand, for non generic calls (calls with not all arguments unbound) we may have to pay the cost of unnecessarily fetch all the tuples from the database. Prolog unification will select the matching tuples.

To implement this approach we changed the `db_import/3` definition. It still receives the same arguments, but now it dynamically constructs and asserts the clause for the predicate being mapped. Consider, for example, that we call `db_import(edge_r,edge,my_conn)`. For this case the following clause will be constructed.

```
edge(A,B) :-
   get_value(my_conn,ConnHandler),
   db_query(ConnHandler,'SELECT * FROM edge_r',ResultSet),
   db_row(ResultSet,[A,B]).
```

To fully implement the process, we need to define the predicates `db_query/3` and `db_row/2` in the `yap2mysql.c` module. The predicate `db_query/3` simply generates the result set for the given query ('`SELECT * FROM edge_r`' in the example). Predicate `db_row/2` is a backtrackable predicate that fetches one row-at-a-time and tries to unify the predicate arguments (`[A,B]` in the example) with the data values in the row. Note that the unification process may fail. For example, if we call `edge(A,1)`, this turns `B` ground when passed to the `c_db_row()` procedure.

```
int c_db_query(void) {
   ...
   MYSQL *conn = (MYSQL *) YAP_IntOfTerm(YAP_ARG1);
   char *query = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2));
   mysql_query(conn, query);
   res_set = mysql_store_result(conn);
   return(YAP_Unify(YAP_ARG3, YAP_MkIntTerm((int) res_set)));
}
```

```
int c_db_row(void) {
   ...
   MYSQL_RES *res_set = (MYSQL_RES *) YAP_IntOfTerm(YAP_ARG1);
   if ((row = mysql_fetch_row(res_set)) != NULL) {
      YAP_Term head, list = YAP_ARG2;
      for (i = 0; i < arity; i++) {
         head = YAP_HeadOfTerm(list);
         list = YAP_TailOfTerm(list);
         if (!YAP_Unify(head, YAP_Mk...(row[i]))) return FALSE;
        }
      return TRUE;
   }
   mysql_free_result(res_set);
   YAP_cut_fail();
}
```

3.3 Transferring Unification to the Database Engine

The last approach uses the `translate/3` predicate from Draxler's compiler to transfer the Prolog unification process to MySQL. Instead of using Prolog unification to select the matching tuples for a non generic call, we dynamically construct specific SQL queries to match the call. By doing this, we discard beforehand the tuples that will not succeed when performing unification. To implement this last approach we need to extend the `db_import/3` definition to include the `translate/3` predicate. If we consider the previous example, the following clauses will now be asserted.

```
edge(A,B) :-
   get_value(my_conn,ConnHandler),
   translate(proj_term(A,B),edge(A,B),QueryString),
   db_query(ConnHandler,QueryString,ResultSet),
   db_row(ResultSet,[A,B]).

relation(edge_r,edge,2).
attribute(1,edge_r,source,integer).
attribute(2,edge_r,dest,integer).
```

When we call `edge(A,1)`, the `translate/3` predicate uses the `relation/3` and `attribute/4` facts to construct a specific query to match the call: '`SELECT source, 1 FROM edge_r WHERE dest=1;`'.

Assume now that we define a `direct_cycle/2` predicate that calls twice the `edge/2` predicate:

```
direct_cycle(A,B) :- edge(A,B), edge(B,A).
```

For the first goal, `translate/3` generates a query '`SELECT * FROM edge_r`', that will access all tuples sequentially. For the second goal, it gets the bindings of the first goal and generates a query of the form '`SELECT 800, 531 FROM edge_r WHERE source=800 AND dest=531`'. These queries return 1 or 0 tuples, and are efficiently executed thanks to the MySQL index associated to the primary key of relation `edge_r`. However, this approach has a substantial overhead of generating, running and storing a SQL query for each tuple of the first goal. To avoid this we can also benefit from the `translate/3` predicate and transfer the joining process to the MySQL engine. To do so, we can create a view using `db_view((edge(A,B),edge(B,A)),direct_cycle(A,B),my_conn)` and the following clause will be constructed.

```
direct_cycle(A,B) :-
   get_value(my_conn,ConnHandler),
   translate(proj_term(A,B),(edge(A,B),edge(B,A)),SqlQuery),
   db_query(ConnHandler,SqlQuery,ResultSet),
   db_row(ResultSet,[A,B]).
```

If later we call `direct_cycle(A,B)`, only a single query will be generated: '`SELECT A.source, A.dest FROM edge_r A, edge_r B WHERE B.source=A.dest AND B.dest=A.source`'.

These two approaches of executing conjunctions of database predicates are usually referred in the literature as *relation level* (one query for each predicate as in the first `direct_cycle/2` definition) and *view level* (an unique query with all predicates as in the constructed `direct_cycle/2` clause). A step forward will be to automatically detect, when consulting a Prolog file, the clauses that contain conjunctions of database predicates and use view level transformations, as in the example above, to generate more efficient code.

4 Performance Evaluation

In order to evaluate the performance of our three approaches, we used Yap 4.4.3 and MySQL server 3.23.52 versions running on the same machine, an AMD Athlon 1400 with 512 Mbytes of RAM. The `edge_r` relation was created in the MySQL DBMS using the following SQL declaration:

```
CREATE TABLE edge_r (
   source  SMALLINT NOT NULL,
   dest    SMALLINT NOT NULL,
   PRIMARY KEY (source,dest));
```

We have used two queries over the `edge_r` relation. The first query was to find all the solutions for the `edge(A,B)` goal, which correspond to all the tuples of relation `edge_r`. The second query was to find all the solutions of the `edge(A,B),edge(B,A)` goal, which correspond to all the direct cycles. We measured the execution time using the `walltime` parameter of the statistics built-in predicate, in order to correctly measure the time spent in the Yap process and in the MySQL process. In the tables that follow, timing results are always presented in seconds.

Table 2 presents the results for the different coupling approaches, with relation `edge_r` having 1,000 vertices and populated with 5,000, 10,000 and 50,000 random tuples.

Coupling Approach/Query	**Tuples**		
	5,000	**10,000**	**50,000**
Asserting Approach			
assert time	0.05	0.30	2.06
`edge(A,B)`	< 0.01	< 0.01	0.02
`edge(A,B),edge(B,A)`	7.17	30.10	753.80
Backtracking Approach			
`edge(A,B)` (*store_result*)	0.02	0.04	0.19
`edge(A,B)` (*use_result*)	0.02	0.04	0.19
`edge(A,B),edge(B,A)` (*store_result*)	91.23	359.40	9,410.7
`edge(A,B),edge(B,A)` (*use_result*)	n.a.	n.a.	n.a.
Backtracking + SQL Unify Approach			
`edge(A,B)` (*store_result*)	0.02	0.04	0.19
`edge(A,B),edge(B,A)` (*relation level*)	0.98	2.20	19.70
`edge(A,B),edge(B,A)` (*view level*)	0.02	0.04	0.28

Table 2: Execution times of the different approaches

For the asserting approach we measured the two queries mentioned above and also the assert time of the involved tuples. This assert time is relevant because the other approaches of coupling do not have this overhead time. The assert time, despite involving multiple context switching between Prolog and C, is fast and can be used with large relations. Using the method described, asserting 50,000 tuples takes about 2 seconds. For comparison, if we dump the relation to a file in the form of Prolog facts and consult this file, Yap takes about 0.6 seconds, which is around 3 times faster. The `edge(A,B)` query involves sequentially accessing all the facts that have been asserted. This is done almost instantly, taking only 0.02 seconds for 50, 000 facts. On the query `edge(A,B),edge(B,A)` this approach shows large difficulties. Even for 5, 000 facts Yap already takes more than 7 seconds and the growth is exponential, taking several minutes for 50, 000 facts. This is due to the fact that, by default, Yap does not index dynamic predicates, such as the asserted edge facts. For each `edge(A,B)`, Prolog execution mechanism will have to access *all* the facts to see if they unify with `edge(B,A)`, because the absence of indexing cannot use the first goal variable bindings to limit the search space. Finally, we would like to note that this asserting approach reduces the overhead of communication with the MySQL server to the initial assert, and the resolution of the queries has no communication with the MySQL server.

To evaluated our second approach we use both `mysql_store_result()` and `mysql_use_result()` procedures. No relevant differences were detected, mainly because Yap and MySQL server were running on the same machine. We could not use `mysql_use_result()` with the `edge(A,B),edge(B,A)` query because this version of MySQL does not allow multiple results sets on the server side (this should be possible with the latest version of MySQL, 4.1). Query `edge(A,B)` takes 0.19 seconds to return the 50, 000 solutions. The overhead of communicating with the MySQL result set structure tuple by tuple causes a slowdown of around 10 times as compared to the previous asserting strategy. This 10 times factor is also reflected on the execution time of `edge(A,B),edge(B,A)`. For both edge goals, a '`SELECT * FROM edge_r`' query is generated and the join is computed by Yap using the two MySQL result sets. We should note that, on this approach, there are no indices on Yap that can be used to speed-up the query, as the `edge_r` tuples only exist in MySQL structures. Also, the difficulties explained for the asserting approach remain, as the indices existing on the MySQL server for the `edge_r` relation are of no use since the queries are '`SELECT * FROM edge_r`'.

The last approach of coupling, which tries to transfer unification to the SQL engine, gives exactly the same results for query `edge(A,B)`, as the query generated by `translate/3` is exactly the same of the previous approach ('`SELECT * FROM edge_r`'). Regarding query `edge(A,B),edge(B,A)` there are very significant differences. For this query we consider a *relation level* access where `translate/3` is used for each goal, and a *view level* access where `translate/3` is used to generate a SQL query which computes the join of the two goals.

For the relation level access the speed-up obtained is of around 100 times for 5, 000 tuples and, more important, allows the increase in execution time to become linear in the number of tuples. Note that the execution times of this approach are not faster because there is a large overhead of communication with MySQL, involving running one query and storing the result on the client for each tuple of the first goal. For view level access `translate/3` generates a single query and Yap just sequentially accesses the returned result set. For 50, 000 tuples the execution time is of 0.28 seconds. This represents a speed-up of more than 2, 500 times over the asserting approach and of more than 30, 000 times over the backtracking approach.

Index performance is fundamental to interpret the results obtained. Note that the asserting approach relies on the logic system indexing capabilities, while the other approaches rely on the database system indexing capabilities. The asserting approach can be improved if indexing can be performed over the dynamic predicate asserted. Yap can index on the first argument of dynamic predicates if we declare the update semantics of dynamic predicates to be *logical* instead of the default *immediate* (`dynamic_predicate(edge/2,logical)`). Dynamic predicates with logical update semantics achieve similar performance when compared with static compiled predicates.

Relational database management systems have extended indexing capabilities as compared to Prolog systems. Every relational database system allows the declaration of several types of indices on different attributes. To evaluate the impact of changing the indexing approach on MySQL we dropped the primary key index of relation `edge_r`: '`ALTER TABLE edge_r DROP PRIMARY KEY`'. We also evaluated performance using a secondary index just on the first attribute of `edge_r`: '`ALTER TABLE edge_r ADD INDEX ind_source (source)`' (this is the traditional indexing approach of Prolog systems). We next compare on Table 3 the performance of these different indexing schemes.

Coupling Approach/Indexing Scheme	**Tuples**		
	50,000	**100,000**	**500,000**
Asserting Approach			
no index	753.80	5270.27	> 2 hours
index on first argument (source)	0.59	2.40	12.88
Backtracking + SQL Unify Approach			
no index	487.35	1997.36	> 2 hours
secondary index on (source)	0.54	1.93	10.25
primary key index on (source,dest)	0.28	0.67	3.81

Table 3: Index performance for query `edge(A,B),edge(B,A)`

Table 3 presents the execution times for 50,000, 100,000 and 500,000 tuples (for 500,000 tuples we used 5,000 vertices) using asserting and backtracking with SQL unification in view level for query `edge(A,B),edge(B,A)`. By observing the table we can see the dramatic impact of indexing when compared with no indexing. An interesting comparison is the time taken with the asserting approach by Yap without indexing (753.80 and 5270.27 seconds for 50,000 and 100,000 tuples), and the time taken by MySQL also without indexing (487.35 and 1997.36 seconds for 50,000 and 100,000 tuples). Yap is about 1.5 to 2.5 times slower than MySQL dealing with no indexed data. Another interesting comparison is the time that Yap and MySQL using an equivalent index on the same argument take. They show almost the same performance, with a small overhead for Yap in this particular query. As expected, best results are obtained for MySQL when using a primary key index on both attributes (0.28, 0.67 and 3.81 seconds for 50,000, 100,000 and 500,000 tuples). Further evaluation must be done for different programs and queries.

5 Concluding Remarks

We studied and evaluated the impact of using alternative approaches for coupling Yap Prolog with MySQL. Through experimentation, we observed that it is possible to couple logic systems with relational databases using approaches based on tuple-at-a-time communication schemes. Our results show however that, in order to be efficient, we need to explore view level transformations when accessing the database. Results also show that indexing is fundamental to achieve scalability. Indexing is important on the database server for view level access and on the Prolog system when tuples are asserted as facts. For Yap, further evaluation should experiment with the current development version of this system, Yap 4.5, where indexing has been improved and can build indices using more than just the first argument. We also plan to perform further evaluation and take advantage of the advanced features of OPTYap [10], such as tabling and or-parallelism, namely in the evaluation of recursive and concurrent queries.

Acknowledgements

This work has been partially supported by APRIL (POSI/SRI/40749/2001) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSI.

References

[1] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1987.

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

[3] C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates*. PhD thesis, Zurich University, 1991.

[4] H. Gallaire and J. Minker, editors. *Logic and Databases*. Plenum, 1978.

[5] M. Jarke, J. Clifford, and Y. Vassiliou. An Optimizing Prolog Front-End to a Relational Query System. In *ACM SIGMOD International Conference on the Management of Data*, pages 296–306. ACM Press, 1984.

[6] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[7] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan-Kaufmann, 1987.

[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] R. Rocha, F. Silva, and V. Santos Costa. On a Tabling Engine that Can Exploit Or-Parallelism. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 43–58. Springer-Verlag, 2001.

[11] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[12] 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.

[13] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data Language. In *International Conference on Very Large Data Bases*, pages 33–41. Morgan Kaufmann, 1986.

[14] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[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.