COUPLING OPTYAP WITH A DATABASE SYSTEM

Michel Ferreira Ricardo Rocha
DCC-FC & LIACC, University of Porto
Rua do Campo Alegre, 823, 4150-180 Porto, Portugal
{michel,ricroc}@ncc.up.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. In this work, we study and evaluate the impact of coupling OPTYap with the MySQL relational database management system. We use the OPTYap extension of the Yap Prolog compiler, which is the first available system that can exploit parallelism from tabled logic programs. We describe the major features of the system, give a detailed description of the implementation and present a performance comparison of using asserted facts or accessing the facts as external relational tuples. Our results show that indexing and view level transformations are fundamental to achieve scalability.

KEYWORDS

Deductive Databases, Coupled Systems Implementation, Performance.

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 (Gallaire H. and Minker J., 1978). 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 (Minker, J., 1987), a restriction of logic programming, Datalog (Ullman, J. D., 1989), 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 technique (Beeri, C. and Ramakrishnan, R., 1987) and tabling (Michie, D., 1968), a technique of memoisation successfully implemented in XSB Prolog (Sagonas, K. and Swift, T., 1998), the most well known tabling Prolog system. Perhaps the most significant difference between XSB and conventional implementations based on rewriting techniques is its tuple-at-a-time evaluation strategy. Tabling relies less on rewriting techniques and on alternate control strategies than is usual in bottom-up approaches. This allows XSB to maintain an operational interpretation of a program in almost as elegant manner as Prolog.

A tabling engine largely based on the original ideas of XSB is also available in the OPTYap system (Rocha, R. et al., 2001). OPTYap is a state-of-the-art system that builds on the high-performance Yap Prolog system (Santos Costa, V. 1999) and combines the power of tabling with support for implicit parallel execution of goals in shared-memory machines. Previous results showed that OPTYap is able to speedup execution and achieve scalability for a wide range of applications using tabling (Rocha, R. et al., 2002).

Our goal in this work is to study and evaluate the impact of coupling OPTYap with the MySQL relational database management system. The interface between the two systems in done by translating the goals, or parts of them, written in logic to the language understood by the database system, SQL. We consider two main approaches: asserting database tuples as Prolog facts, and accessing facts as external relational tuples. We present a detailed step-by-step description of each approach and for that we use the C client libraries of Yap and MySQL. 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 our coupling approaches, detailing their implementation differences. We then evaluate their performance on a simple relational schema. We finalize by outlining some concluding remarks.

2. DEVELOPMENT TOOLS

To implement the interface between OPTYap and MySQL we took advantage of their client libraries that allows writing external modules in C. To optimize the translation of queries between both systems we used a Prolog to SQL compiler. 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. The arguments of a Prolog predicate written in C are accessed through the macros `YAP_ARG1`, `YAP_ARG2`, etc. A C variable can have type `YAP_Term`, the type used for holding Yap terms, and the conversion is done by macros such as `YAP_MkIntTerm()`. Unification between Yap terms is done in C by calling `YAP_Unify()`. It is also possible 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.

Another 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; 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. For backtrackable predicates we need two C functions: one to be executed when the predicate is first called, and another to be executed on backtracking to provide (possibly) other solutions. When returning the last solution, we should use `YAP_cut_fail()` to denote failure, and `YAP_cut_succeed()` to denote success. This is necessary as otherwise, when backtracking, our function would be indefinitely called.

2.2 THE MYSQL C API

MySQL provides a client library written in C for writing client programs that access MySQL databases. 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. Initially, we allocate a connection handler and try to establish a connection to the desired server. 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. 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 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.

2.3 PROLOG TO SQL COMPILER

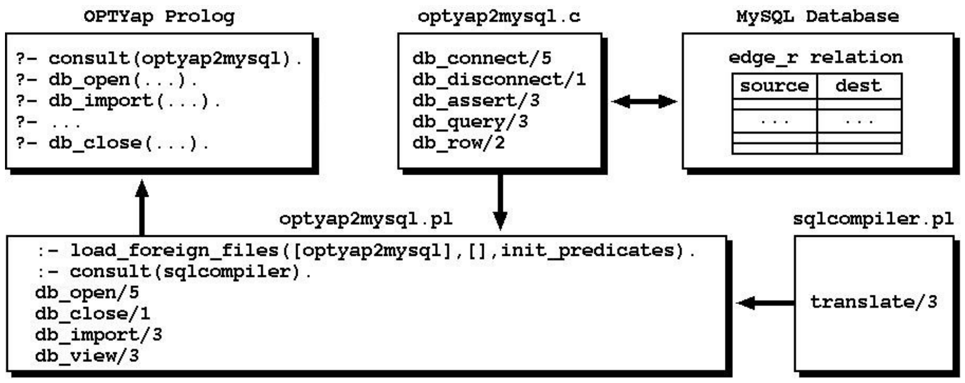
The interface between Prolog and database systems is normally done via the SQL language. A particular Prolog predicate is assigned to a given relation in a database and its facts are made available through the tuples returned by a SQL query. Normally, it is also possible to write Prolog predicates which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

An important implementation of a generic Prolog to SQL compiler is the work done by Draxler (Draxler C., 1991). This 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 two alternative approaches for coupling OPTYap with the MySQL database management system. The architecture of this interface is presented next in Figure 1.

Figure 1. Interface layout



The `optyap2mysql.c` is our main module. It uses the Yap and MySQL interfaces to the C language to implement the low-level communication predicates. The `sqlcompiler.pl` is Draxler's Prolog to SQL compiler. It will be necessary to implement the second approach. The `optyap2mysql.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 `optyap2mysql.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 second approach, the definition of database views based on these predicates 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 `optyap2mysql.c` module.

```
db_open(Host,User,Password,Database,ConnName) :-
  db_connect(Host,User,Password,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 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 asserting 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 tuple, it calls the Prolog interpreter to assert it as a Prolog fact. To do so it constructs Prolog terms of the form `assert(fp(arg[0],...,arg[arity-1]))`, where `fp` is the predicate name for the asserted facts and `arg[]` are the data values for each tuple.

```
int c_db_assert(void) {
  ...                                                  // auxiliary variables
  YAP_Functor fp, fa;
  YAP_Term *arg, tp, ta;
  MYSQL *conn = YAP_IntOfTerm(YAP_ARG1);               // connection handler
  char *rel = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2));
  sprintf(query,"SELECT * FROM %s", rel);              // construct the query
  mysql_query(conn, query);                            // issue the query for execution
  rset = mysql_store_result(conn);                     // get the result set
  arity = mysql_num_fields(rset);
  fp = YAP_MkFunctor(YAP_AtomOfTerm(YAP_ARG3), arity);
  fa = YAP_MkFunctor(YAP_LookupAtom("assert"), 1);
  while ((row = mysql_fetch_row(rset)) != NULL) {      // process each row
    for (i = 0; i < arity; i++) {
      ...; arg[i] = YAP_Mk...(row[i]); ...;            // construct appropriated term
    }
    tp = YAP_MkApplTerm(fp, arity, arg);
    ta = YAP_MkApplTerm(fa, 1, &tp);
    YAP_CallProlog(ta);                                // assert tuple as a Prolog fact
  }
  mysql_free_result(rset);
  return TRUE;
}
```

3.2 ACCESSING FACTS AS EXTERNAL RELATIONAL TUPLES

This second 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, 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.

To implement this approach we changed the `db_import/3` definition. It now 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 asserted.

```
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 `db_query/3` and `db_row/2` predicates in the `optyap2mysql.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(1,B)`, this turns `A` ground when passed to the `c_db_row()` procedure.

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

int c_db_row(void) {
  ...
  MYSQL_RES *rset = YAP_IntOfTerm(YAP_ARG1);
  if ((row = mysql_fetch_row(rset)) != 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(rset);
  YAP_cut_fail();
}
```

Performing this unification with the Prolog engine is clearly a bad idea. Not only we pay the cost of unnecessarily fetch all the tuples from the database, selecting then just those that unify, but also we could expect this unification to be performed faster on the MySQL engine, thanks to more powerful indexing capabilities. An important improvement is to be able to generate a query based on the current instantiations of the arguments of the database goal, rather than the generic '`SELECT * FROM edge_r`' query.

For this dynamic SQL generation we use the `translate/3` predicate from Draxler's compiler to transfer the unification process to MySQL, and we discard beforehand the tuples that will not unify with the calling pattern of the database goal. We 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),Query),
  db_query(ConnHandler,Query,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(1,B)`, the `translate/3` predicate uses the `relation/3` and `attribute/4` facts to construct a specific query to match the call: ‘`SELECT 1,dest FROM edge_r WHERE source=1`’. Assume now that we define a `direct_cycle/2` predicate that calls twice the `edge/2` predicate, and call it with unbound variables: `direct_cycle(A,B):- edge(A,B), edge(B,A).`

For the first goal, translate/3 generates a 'SELECT * FROM edge_r' query, that will access all tuples sequentially. For the second goal, it gets the bindings of the first goal and generates queries 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 define views by using the db_view/3 definition. For example, if we call db_view((edge(A,B),edge(B,A)), direct_cycle(A,B),my_conn) the following clause will be asserted.

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

When later we call direct_cycle(A,B), only a single query is 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 the clauses that contain conjunctions of database predicates and use view level transformations, as in the example above, to generate more efficient code.

3.3 USING TABLING TO SOLVE RECURSIVE QUERIES

As mentioned previously, the expressive power of Datalog is greater than the expressive power of relational algebra because of the ability to write recursive queries. Recursive queries evaluated using Prolog SLD resolution can often be non-terminating and tend to recompute the same answer. Rewriting techniques have been developed to solve these problems of lack of finiteness and redundant computation. The tabling technique offers an alternative to solve these problems, and compares favorably in terms of performance, as demonstrated by the XSB system (Sagonas et al., 1994).

The basic idea behind tabling is straightforward: calls to tabled predicates are evaluated by storing intermediate answers in an appropriate data space, called the *table space*, so that they can be reused when a repeated call appears. Execution proceeds as follows. Whenever a tabled goal *G* is first called, an entry for *G* is allocated in the table. This entry will collect all the answers found for *G*. Repeated calls to *variants* of *G* are resolved by consuming the answers already stored in the table instead of being re-evaluated against the predicate clauses. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant calls. Using the table directive of OPTYap allows the efficient evaluation of recursive predicates including database goals. For example, assuming edge/2 defined as above, the following tabled predicate computes its transitive closure.

```
:- table path/2.
path(A,B) :- edge(A,B).
path(A,B) :- path(A,C), edge(C,B).
```

If we use Prolog SLD resolution, a call like path(1,B) will lead to an infinite computation because it calls itself recursively in the second clause. On the other hand, the OPTYap tabling mechanism easily detects the recursive call and avoids its re-evaluation. Moreover, if a parallel shared-memory machine is available, we can take full advantage of the OPTYap system to speedup the execution by exploiting implicit or-parallelism. To exploit parallelism no further annotations need to be made to the program.

4. PERFORMANCE EVALUATION

In order to evaluate the performance of our two coupling approaches, we used Yap 4.4.3 and MySQL server 4.1.1-alpha versions running on the same machine, an AMD Athlon 1400 with 512 Mbytes of RAM.

The `edge_r` relation was created in MySQL using a primary key on (*source,dest*). To evaluate accessing MySQL tuples using an equivalent indexing scheme as used in Prolog systems, we also used a schema were the primary key index was dropped and replaced by a secondary index on attribute *source*.

We have used two queries over the `edge_r` relation. The first query is '`:-edge(A,B),fail.`', traversing all tuples of relation `edge_r`. The second query is '`:-edge(A,B),edge(B,A),fail.`', traversing all the solutions that represent direct cycles. This second query is evaluated using both relation-level access and view-level access, as explained previously. We also present results for a recursive query '`:-path(1,B),fail.`', executed a first time, where the solutions table is created, and a second time, after solutions are tabled. Table 1 presents the execution times in seconds for these queries.

Table 1. Performance evaluation

Coupling Approach/Query	**Tuples**		
	50,000	**100,000**	**500,000**
Asserting (*index on first argument*)			
Assert time	2.12	17.05	90.89
`:- edge(A,B),fail.`	0.02	0.03	0.16
`:- edge(A,B),edge(B,A),fail.`	0.54	2.17	10.94
`:- path(1,B),fail.` (*first call*)	0.15	0.34	2.20
`:- path(1,B),fail.` (*second call*)	< 0.01	< 0.01	< 0.01
SQL + Backtracking (*primary index on (source,dest)*)			
`:- edge(A,B),fail.`	0.22	0.44	2.18
`:- edge(A,B),edge(B,A),fail.` (*relation level*)	23.29	69.81	1,272.81
`:- edge(A,B),edge(B,A),fail.` (*view level*)	0.35	0.82	4.78
`:- path(1,B),fail.` (*first call*)	0.53	0.82	3.99
`:- path(1,B),fail.` (*second call*)	< 0.01	< 0.01	< 0.01
SQL + Backtracking (*primary index on (source)*)			
`:- edge(A,B),fail.`	0.18	0.37	1.95
`:- edge(A,B),edge(B,A),fail.` (*relation level*)	39.88	119.84	1,779.26
`:- edge(A,B),edge(B,A),fail.` (*view level*)	6.94	26.18	142.14
`:- path(1,B),fail.` (*first call*)	0.61	0.91	4.92
`:- path(1,B),fail.` (*second call*)	< 0.01	< 0.01	< 0.01

For the asserting approach we measured the queries mentioned above and also the assert time of the involved tuples. This assert time is relevant because the other approach of coupling does 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, OPTYap takes about 0.6 seconds, which is around 3 times faster.

The '`:-edge(A,B),fail.`' query involves sequentially accessing all the facts that have been asserted. As there is no communication with MySQL, this is done almost instantly, taking only 0.16 seconds for the 500,000 in-memory facts. On the query '`:-edge(A,B),edge(B,A),fail.`' OPTYap is able to use indexing to limit the search space on the second `edge/2` goal, executing the query quite efficiently and with linear execution time growth on the number of tuples. For the recursive query '`:-path(1,B),fail.`', the first call builds the solutions table without any interaction with MySQL. A second call to `path(1,B)` will just traverse the stored solutions in the table, and is executed instantly.

On the second coupling approach, query '`:-edge(A,B),fail.`' takes 2.18 and 1.95 seconds to return the 500,000 solutions using the two indexing schemes. Note that indices are not used for this query. 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.

The use of relation-level access to evaluate query '`:-edge(A,B),edge(B,A),fail.`' clearly shows the poor performance of this strategy. With relation-level access there is a very large overhead of communication with MySQL, involving running one query and storing the result on the client for each tuple of the first `edge/2` goal. Results in table 1 show the importance of using view-level access, that is able to transfer the joining process to the MySQL engine, and process a single query and the associated result set in OPTYap. For 500,000 tuples view-level access takes 4.78 using a primary index on both attributes and 142.14 seconds using a secondary index on attribute *source*. The availability of creating more powerful indices on MySQL allows significant speed-ups, like the 2.2 times factor of MySQL using a primary index over OPTYap using indexing on the first argument. On the other hand, the speed-up of OPTYap using an equivalent indexing scheme (a 13 times factor) can only be explained by the execution mechanism of Prolog, that for this particular query is more efficient than the joining mechanism applied by MySQL using a secondary index.

The recursive '`:-path(1,B),fail.`' presents a slow-down factor of around 2 times over the asserting approach, during the first call where the solutions table is being constructed. Again this is due to the communication overhead with MySQL and the processing of the result set tuple-at-time, as the table is constructed based on SQL queries. Clearly, subsequent calls are also instantly executed, as the same in-memory table structure is used by both approaches.

5. CONCLUDING REMARKS

We studied and evaluated the impact of coupling OPTYap with MySQL using approaches based on tuple-at-a-time communication schemes. Our results show that, in order to be efficient, we need to explore view-level transformations when accessing the database via dynamic SQL queries. Results also show that indexing is fundamental to achieve scalability. The available indexing schemes of database management systems, together with view-level access, can overcome the slowdown caused by accessing and processing result sets from external database systems, as compared to in-memory Prolog facts. For OPTYap, further evaluation should experiment with the current development version of Yap, version 4.5, where *just-in-time* indexing based on instantiated arguments is used. We also plan to perform further evaluation for more complex queries that can, in particular, explore the or-parallelism of OPTYap on parallel shared-memory machines.

ACKNOWLEDGEMENT

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

REFERENCES

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

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

Gallaire H. and Minker J., 1978. *Logic and Databases*. Plenum.

Michie, D., 1968. Memo Functions and Machine Learning. *Nature*, 218:19—22.

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

Rocha, R. et al., 2001. On a Tabling Engine That Can Exploit Or-Parallelism. *In International Conference on Logic Programming*, number 2237 in LNCS, pp 43—8. Springer-Verlag.

Rocha, R. et al., 2002. Achieving Scalability in Parallel Tabled Logic Programs. *In International Parallel and Distributed Processing Symposium*. IEEE Computer Society.

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

Sagonas et al., 1994. XSB as an Efficient Deductive Database Engine. *In ACM SIGMOD International Conference on the Management of Data*, pp 442—453. ACM Press.

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

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