The MyYapDB Deductive Database System

Michel Ferreira and Ricardo Rocha*

DCC-FC & LIACC, University of Porto
Rua do Campo Alegre, 823, 4150-180 Porto, Portugal
Tel. +351 226078830, Fax. +351 226003654
{michel,ricroc}@ncc.up.pt

Abstract. We describe the MyYapDB, a deductive database system coupling the Yap Prolog compiler and the MySQL DBMS. We use our OPTYap extension of the Yap 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 simplified description of the implementation and present a performance comparison of using static facts or accessing the facts as MySQL tuples for a simple example.

1 Introduction

Logic programming and relational databases have common foundations based on First Order Logic [4]. The motivation for combining logic with relational databases is to 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 more efficiently the extensional knowledge through database relations and the intensional knowledge through logic rules.

In the specific field of deductive databases [6], a restriction of logic programming, Datalog [9], 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 [5], a technique of memoisation successfully implemented in XSB Prolog [8], the most well known tabling Prolog system, and also in the OPTYap Prolog system [7].

The main concern in MyYapDB is in performance. Both Yap and MySQL are systems known for their performance. MyYapDB explores specific features of Yap and MySQL to build an external module which uses the C API's of each system to obtain an efficient deductive database coupled engine. OPTYap is also the first available system that can exploit parallelism from tabled logic programs, which seems interesting to further improve performance through the concurrent evaluation of database queries. Applications of our system include areas such as Knowledge Based Systems, Model Checking or Inductive Logic Programming.

* 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.

2 Basic Description of the System

In coupled deductive database systems, the communication between the Prolog engine and the relational database is usually done via a SQL query. MySQL answers to a SQL query with a structure called a *result set*, which includes the selected tuples and meta-data about these tuples. Following MySQL alternatives, MyYapDB allows for the result set to be copied to the Yap client process, or to be left in the MySQL server [3]. The tuples of this local or remote result set are then made available as Prolog facts in a tuple-at-a-time basis via backtracking.

A very important issue in terms of performance is to be able to transfer as much *unification* as possible from the Prolog engine to the database engine in the evaluation of database goals. Relational database engines traditionally have more powerful indexing schemes than Prolog engines and thus are able to solve more efficiently relational operations such as selections and joins. Dynamic SQL query generation based on the bindings of logical variables is thus fundamental in order to select exactly the tuples that unify with the Prolog goal calling pattern. Conjunctions and disjunctions of database goals should also be translated into a single SQL query, replacing a less efficient *relation-level* access for what is known as *view-level* access. MyYapDB allows for the explicit declaration of views and we plan to automatically replace constructs such as database goals conjunctions by compiler created views.

In MyYapDB the dynamic SQL query generation is done using a generic Prolog to SQL compiler written by Draxler [2]. 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.

In MyYapDB, the association between a Prolog predicate and a database relation is defined using a directive such as '`:- db_import(edge_r,edge,my_conn)`', where `edge_r` is a MySQL relation, `edge` is a Prolog predicate and `my_conn` is a connection to a MySQL server. This directive asserts the following Prolog clause:

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

Predicates `db_query/3` and `db_row/2` are external predicates written in C. The first is a deterministic predicate that sends a SQL query to MySQL and stores the result set. The later is a backtrackable predicate that fetches a tuple at a time from the result set and unifies the tuple with a list of variables.

When we call '`edge(A,1)`', the `translate/3` predicate constructs a specific query to match the call: '`SELECT source,1 FROM edge_r WHERE dest=1`', where `source` and `dest` are the attributes names of relation `edge_r`.

The definition of views is similar. When programmers use a directive such as '`:- db_view((edge(A,B),edge(B,A)),direct_cycle(A,B),my_conn)`', the following clause is asserted:

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

If later we call '`direct_cycle(A,B)`', `translate/3` constructs the query: '`SELECT A.source,A.dest FROM edge_r A,edge_r B WHERE B.source=A.dest AND B.dest=A.source`'. This is clearly more efficient than if we define a predicate `direct_cycle/2` in Prolog using relation level access:

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

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(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), edge(Z,Y).
```

3 Performance Evaluation

We want to evaluate the overhead of accessing Prolog facts in the form of MySQL tuples compared to statically compiled Prolog facts. We also want to evaluate the advantages allowed by using MySQL indexes schemes compared to the simple indexing scheme of Prolog. 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. We have used two queries over the `edge_r` relation of the examples above. 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.

Table 1 presents execution times (in seconds) for Yap with `edge/2` facts as statically compiled facts and indexed on the first argument (this is the available indexing scheme in Yap 4.4.3), and for MyYapDB with edge/2 facts fetched from the `edge_r` relation with a secondary index on *(source)* and a primary index on *(source,dest)*. Note that first argument indexing is the only available indexing scheme on almost all Prolog systems. XSB is one of the most well-know exceptions. The current development version of Yap, version 4.5, is also being improved to build indices using more than just the first argument. Further evaluation should experiment with these systems.

As expected, Table 1 confirms that view-level access is much more efficient than relation-level access. For queries that access sequentially a set of tuples, the

System/Query	**Tuples (Facts)**		
	50,000	**100,000**	**500,000**
Yap (*index on first argument*)			
`edge(A,B)`	0.02	0.03	0.17
`edge(A,B),edge(B,A)`	5.97	24.10	132.15
MyYapDB (*secondary index on (source)*)			
`edge(A,B)`	0.18	0.37	1.95
`edge(A,B),edge(B,A)` (*relation level*)	39.88	119.84	1,779.26
`edge(A,B),edge(B,A)` (*view level*)	6.94	26.18	142.14
MyYapDB (*primary index on (source,dest)*)			
`edge(A,B)`	0.22	0.44	2.18
`edge(A,B),edge(B,A)` (*relation level*)	23.29	69.81	1,272.81
`edge(A,B),edge(B,A)` (*view level*)	0.35	0.82	4.78

Table 1. Performance evaluation

overhead of MyYapDB compared to Yap accessing compiled facts is a factor of 10. For queries which take advantage of indexing schemes, an interesting comparison is the time taken by Yap and by MyYapDB using an equivalent indexing scheme. Results show a small overhead of 10% on MyYapDB using view-level access. The most interesting result for potential users of MyYapDB is the ability to use the available indexing capabilities of MySQL on the database predicates, which can allow very important speed-ups, like a factor of 25 for our example by using a primary index on both attributes. Further evaluation must also be done for different programs and queries.

References

1. C. Beeri and R. Ramakrishnan. On the Power of Magic. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1987.
2. C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates.* PhD thesis, Zurich University, 1991.
3. 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*, 2004. To appear.
4. H. Gallaire and J. Minker, editors. *Logic and Databases.* Plenum, 1978.
5. D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.
6. Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan-Kaufmanm, 1987.
7. 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.
8. 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.
9. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.