Coupling OPTYap with a Database System

Michel Ferreira Ricardo Rocha
{*michel,ricroc*}*@ncc.up.pt*

DCC-FC & LIACC
University of Porto, Portugal

Motivation

Obtain an efficient Deductive Database System by coupling OPTYap with MySQL.

Motivation

Obtain an efficient Deductive Database System by coupling OPTYap with MySQL.

- **Deductive databases** are logic programming systems designed for applications with large amounts of data. Deductive databases generalise relational databases by exploiting the expressive power of (potentially recursive) logical rules, greatly simplifying the task of application programmers.

Motivation

Obtain an efficient Deductive Database System by coupling OPTYap with MySQL.

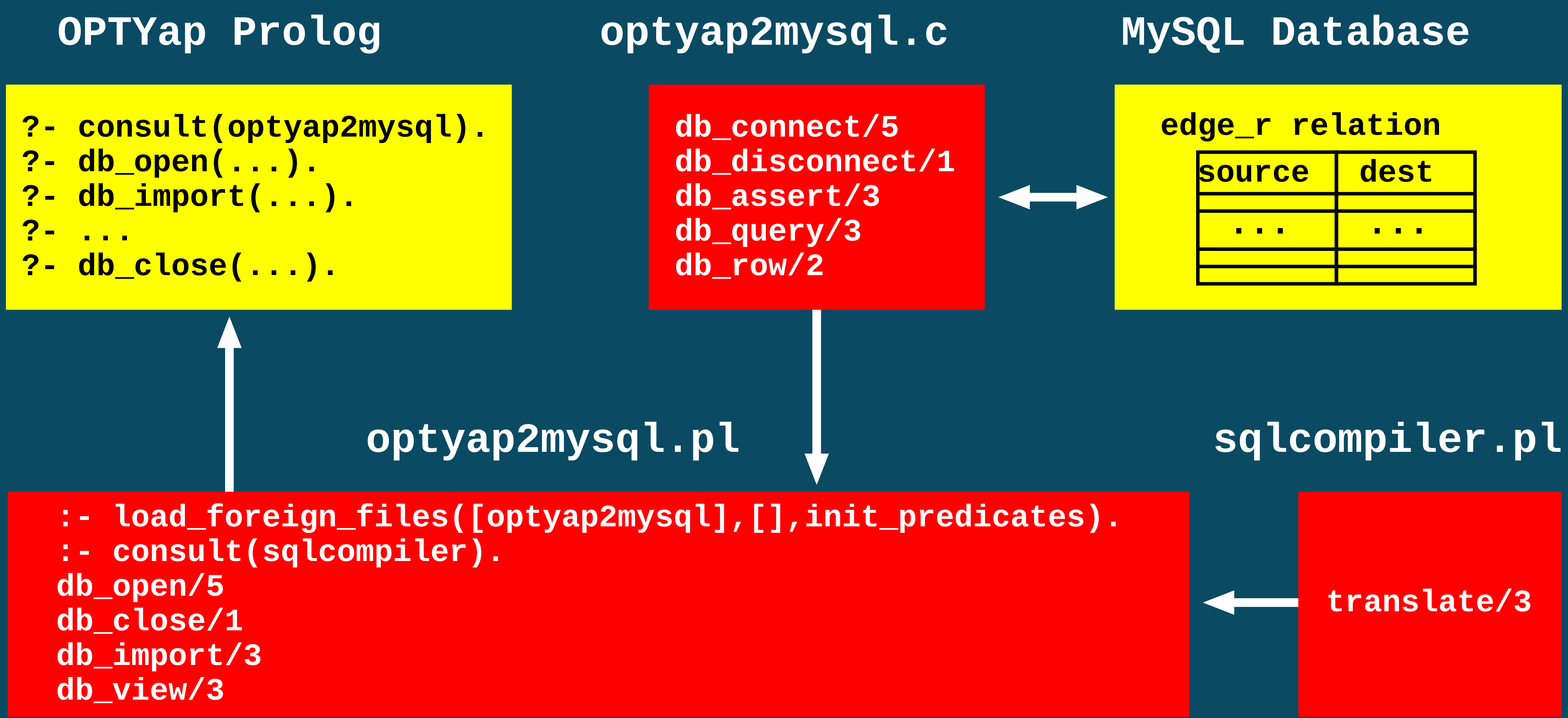
- **Deductive databases** are logic programming systems designed for applications with large amounts of data. Deductive databases generalise relational databases by exploiting the expressive power of (potentially recursive) logical rules, greatly simplifying the task of application programmers.
- **OPTYap** is a state-of-the-art system that builds on the high-performance Yap Prolog compiler and combines the power of tabling with support for implicit parallel execution of Prolog goals in shared-memory machines.
  - OPTYap allows tabled evaluation of Prolog goals, which can support the efficient resolution of recursive queries.
  - OPTYap also supports or-parallel execution of Prolog clauses, which is interesting to concurrently evaluate database queries.

Motivation

Obtain an efficient Deductive Database System by coupling OPTYap with MySQL.

- **Deductive databases** are logic programming systems designed for applications with large amounts of data. Deductive databases generalise relational databases by exploiting the expressive power of (potentially recursive) logical rules, greatly simplifying the task of application programmers.
- **OPTYap** is a state-of-the-art system that builds on the high-performance Yap Prolog compiler and combines the power of tabling with support for implicit parallel execution of Prolog goals in shared-memory machines.
    - OPTYap allows tabled evaluation of Prolog goals, which can support the efficient resolution of recursive queries.
    - OPTYap also supports or-parallel execution of Prolog clauses, which is interesting to concurrently evaluate database queries.
- **MySQL** is a widely used database management system (DBMS), known for its high performance.

Interface between OPTYap and MySQL



- Development tools:
  - Yap Prolog C API
  - MySQL C API
  - Prolog to SQL compiler (Christoph Draxler, 1991)

Accessing Database Tuples Through Backtracking

- When mapping a database relation into a Prolog predicate we use 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**.

Accessing Database Tuples Through Backtracking

- When mapping a database relation into a Prolog predicate we use 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**.

- To implement this approach, we **dynamically construct** the clause for the predicate being mapped. For example, if we call db_import(edge_r,edge,my_conn) the following clause is asserted:

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

Accessing Database Tuples Through Backtracking

- When mapping a database relation into a Prolog predicate we use 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**.

- To implement this approach, we **dynamically construct** the clause for the predicate being mapped. For example, if we call db_import(edge_r,edge,my_conn) the following clause is asserted:

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

- Note that the db_row/2 predicate may fail. For example, if we call edge(1,B), this turns A ground when passed to db_row/2.

Transferring Unification to the Database Engine

- Instead of using Prolog unification to select the matching tuples for the generic SELECT * FROM query, **bindings of goal arguments** are used to dynamically construct **specific SQL queries** to match the call.

Transferring Unification to the Database Engine

- Instead of using Prolog unification to select the matching tuples for the generic SELECT * FROM query, **bindings of goal arguments** are used to dynamically construct **specific SQL queries** to match the call.

- To implement this approach we use the translate/3 predicate from Draxler's compiler. If we consider the previous example, the following clause 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]).
```

- When we call edge(1,B), the translate/3 predicate constructs the specific query SELECT 1, dest FROM edge_r WHERE source = 1.

Transferring Unification to the Database Engine

- We can also **transfer the joining process** of more than one database goal to the MySQL engine. Consider, for example, the following predicate and query goal:

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

?- direct_cycle(A,B).
```

Transferring Unification to the Database Engine

- We can also **transfer the joining process** of more than one database goal to the MySQL engine. Consider, for example, the following predicate and query goal:

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

?- direct_cycle(A,B).
```

- 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 1, 2 FROM edge_r WHERE source = 1 AND dest = 2.
- 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 to define **views**.

View-Level Access

- We can define views by using the db_view/3 predicate. For example, if we call db_view((edge(A,B),edge(B,A)),direct_cycle(A,B),my_conn) the following clause is 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.

Using Tabling to Solve Recursive Queries

- Recursive queries can often be **non-terminating** and tend to **recompute the same answers**. Consider, for example, the predicate path/2 that computes the transitive closure of the edge relation.

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

Using Tabling to Solve Recursive Queries

- Recursive queries can often be **non-terminating** and tend to **recompute the same answers**. Consider, for example, the predicate path/2 that computes the transitive closure of the edge relation.

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

- If we use Prolog standard 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. To evaluate a predicate using tabling we simply need to use the table directive.

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

Some Performance Results

Approach/Query	Tuples (Facts)		
	50,000	**100,000**	**500,000**
Prolog Backtracking (*index on first argument*)			
`edge(A,B),fail.`	0.02	0.03	0.16
`edge(A,B),edge(B,A),fail.`	0.54	2.17	10.94
SQL + Backtracking (*primary index on (source)*)			
`edge(A,B),fail`	0.18	0.37	1.95
`edge(A,B),edge(B,A),fail.`	39.88	119.84	1,779.26
`edge(A,B),edge(B,A),fail.` (*view-level*)	6.94	26.18	142.14
SQL + Backtracking (*primary index on (source,dest)*)			
`edge(A,B),fail`	0.22	0.44	2.18
`edge(A,B),edge(B,A),fail.`	23.29	69.81	1,272.81
`edge(A,B),edge(B,A),fail.` (*view-level*)	0.35	0.82	**4.78**

- The MySQL tuple by tuple communication is around 10 times slower
- View-level access introduces significant speed-ups
- Extended indexing capabilities of MySQL can be very useful

Conclusions and Further Work

- Conclusions
  - Accessing and processing result sets from external database systems can cause a significant slowdown when compared with in-memory Prolog facts.
  - In order to be efficient, we need to explore the available indexing schemes of database management systems, together with view-level transformations when accessing the database.

Conclusions and Further Work

- Conclusions
  - Accessing and processing result sets from external database systems can cause a significant slowdown when compared with in-memory Prolog facts.
  - In order to be efficient, we need to explore the available indexing schemes of database management systems, together with view-level transformations when accessing the database.

- Further Work
  - Automatically detect the clauses that contain conjunctions of database predicates and use view-level transformations to generate more efficient code.
  - Further evaluation with more complex queries/applications that can, in particular, explore the or-parallel component of OPTYap.