Implementation of a Deductive Database System

Advanced Topics in Databases

Implementation of deductive database systems

- There are two main approaches:
  - Integrated systems – The logic programming system and the database system are developed together in a monolithic and integrated manner.
  - Coupled systems – There are two separate systems, one as a logic programming system and one as a database management system., which are interconnected through communication interfaces.

- In integrated systems, datalog queries can be executed directly, without the need to translate to a language understood by a database management system.
- Contrarily, in coupled systems datalog queries have to be first translated to SQL, that is then sent for execution by the database management system.

Implementation of a coupled deductive database system

- Tools:
  - A logic programming system (Yap Prolog);
  - A translator of Prolog to SQL (SQLCompiler.pl, written by Christoph Draxler);
  - A SQL Database system (MySQL).

- The generic idea of our implementation is to use the C API of MySQL and the C API of Yap to build a C program that implements the interface between the two systems.
- The compiler of Datalog (Prolog) to SQL, written in Prolog, is used at the level of the Prolog/Datalog program we want to compile, translating logic queries to databases into SQL.

The Yap C API

- As many other Prolog systems, Yap Prolog has an external interface to write predicates in other programming languages, such as C, as *external modules*.
- Example:
  - Assume we want to write a predicate my_random(N) that unifies N with a random number.
  - For this purpose we have to create a module my_rand.c with the following C code:

```
#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));
}
```

Example of an external module in C

- The module has to be compiled as a *shared object* e load into Yap through the following directive:

```
:- load_foreign_files([my_rand],[],init_predicates).
```

- After this the goal **?- my_random(N).** unifies **N** with a random number.
- The include declaration enables the macros necessary to make the interface with Yap.
- The init_predicates() function tells Yap, through the call to YAP_UserCPredicate(), which predicates are defined in the module.
- Note that this function has no arguments despite the predicate being defined having 1. The arguments of a Prolog predicate written in C are accessed through the macros YAP_ARG1,..., YAP_ARG16.

Primitives to the manipulation of Yap terms in C

Termo	Teste	Construtor	De-construtor
uninst var	YAP_IsVarTerm()	YAP_MkVarTerm()	(nenhum)
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()

Types of predicates defined in external modules in C

- Yap distinguishes between two types of predicates that can be defined in C external modules:
  - deterministic predicates
  - backtrackable predicates

- Deterministic predicates can succeed or fail, but never invoke backtracking upon failure.
- Backtrackable predicates can succeed several time, invoking backtracking to derive alternative solutions.
- The my_random(N) predicate is an example of a deterministic predicate, which is implemented by a single C function.
- Backtrackable predicates require two functions for their definition, one to the first time the predicate is called and one to the subsequent call through backtracking.

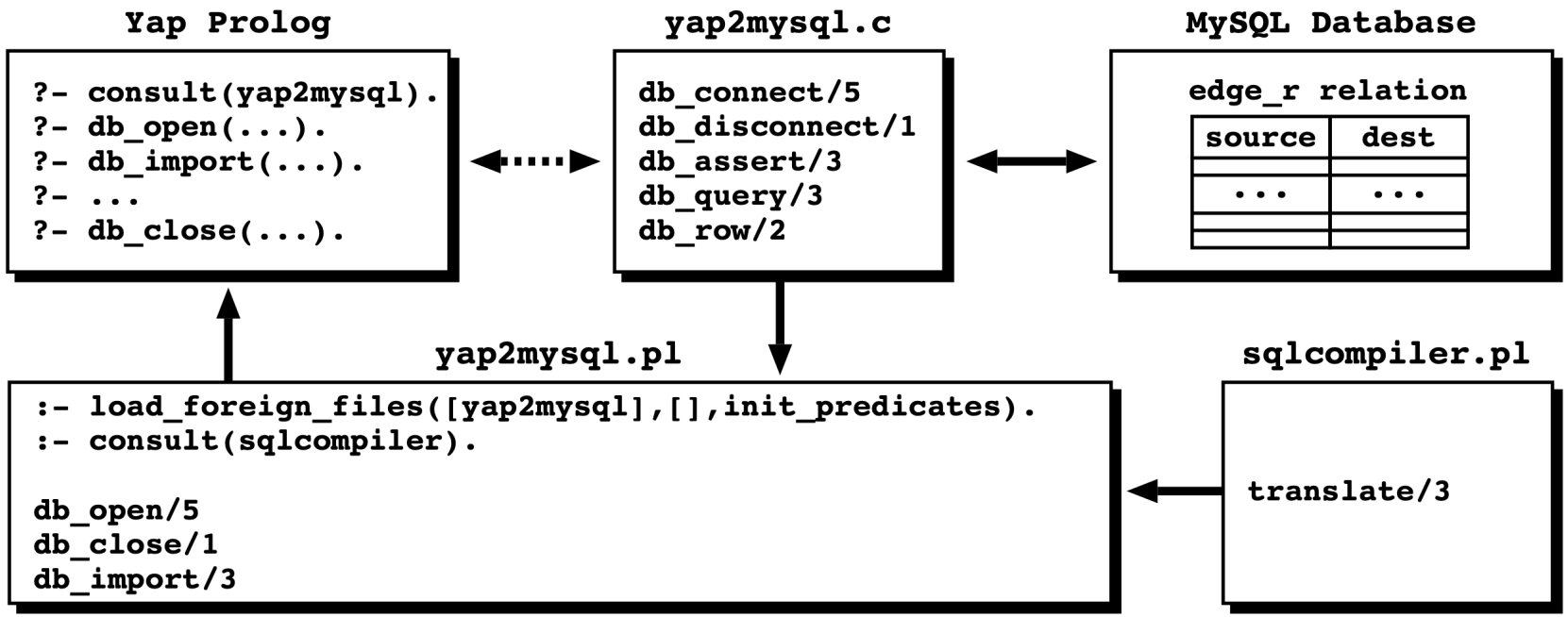
The Prolog to SQL compiler

- The generic compiler of Prolog/Datalog to SQL, written by Christoph Draxler, defines a predicate translate/3 that receives as first argument a projection term, as second argument a database goal (query) and returns in the third argument the SQL representation equivalent to that goal.
- Example:

```
?- translate(ramo(A,B),(ramo(A,B),ramo(B,A)), RSQL),
   queries_atom(RSQL,SQL),
   write(SQL).

SELECT A.origem,A.dest FROM ramo A,ramo B
WHERE B.source=A.dest AND B.dest=A.source;
```

Generic architecture of the implementation



Implemented database predicates

- db_open/5 – initializes a conection with the database server.
- db_import/3 – associates a Prolog predicate with a database relation through an open connection.
- db_close/1 – closes a connection to the database server.

```
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).
```

Implementation of db_connect/5 through the C API

```
int c_db_connect(void) {
  YAP_Term arg_host = YAP_ARG1;
  YAP_Term arg_user = YAP_ARG2;
  YAP_Term arg_passwd = YAP_ARG3;
  YAP_Term arg_database = YAP_ARG4;
  YAP_Term arg_conn = YAP_ARG5;
  MYSQL *conn;
  char *host = YAP_AtomName(YAP_AtomOfTerm(arg_host));
  char *user = YAP_AtomName(YAP_AtomOfTerm(arg_user));
  char *passwd = YAP_AtomName(YAP_AtomOfTerm(arg_passwd));
  char *database = YAP_AtomName(YAP_AtomOfTerm(arg_database));
  conn = mysql_init(NULL);
  if (conn == NULL) {
    printf("erro no init\n");
    return FALSE;
  }

  if (mysql_real_connect(conn, host, user, passwd, database, 0, NULL, 0) == NULL) {
    printf("erro no connect\n");
    return FALSE;
  }
  if (!YAP_Unify(arg_conn, YAP_MkIntTerm((int) conn)))
    return FALSE;
  return TRUE;
}
```

Implementation of db_disconnect/1 through the C API

```
int
c_db_disconnect(void) {
  YAP_Term arg_conn = YAP_ARG1;

  MYSQL *conn = (MYSQL *) YAP_IntOfTerm(arg_conn);

  mysql_close(conn);
  return TRUE;
}
```

Implementation of db_import/3

- Two alternatives:
  - Make an assert of the relational tuples as Prolog facts.
  - Access the tuples tuple-at-a-time through backtracking.

- The first approach is the simplest. Initially the tuples are read from the databases and loaded dynamically as Prolog facts. From that point on, there is no difference between predicates associated with relational tables and the other predicates.
- The second approach generates a SQL query whenever a predicate associated with a relational table, with the associated tuples being passed as solutions to the Prolog predicate through backtracking.
- This second approach can be improved by generating SQL queries that use the current bindings of Prolog variables to select just the tuples that satisfy such bindings. It is even possible to join several database goals in a single SQL query, so that it is the database system that computes such joins instead of the Prolog system (which is slower).

Assert approach

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

int c_db_assert(void) {
   ...                                          // declare 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;}
```

Backtracking approach

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

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)));
}
```

db_row/2

```
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();
}
```

db_import/3

```
db_import(RelationName, PredName, ConnName) :-
                        get_value(ConnName,ConnHandler),
                        db_arity(ConnHandler,RelationName,Arity),
                        functor(F,PredName,Arity),
                        F =..[_ | LArgs],
                        atom_concat('select * from ',RelationName, SQL),
                        assert(':-'(F,','(db_query(ConnHandler,SQL,ResultSet),db_row(ResultSet,LArgs)))).
```

Transferring unification to the DBMS

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

- Joining database goals:

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

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