The MYDDAS Programmer's Manual

Tiago Soares, Michel Ferreira, Ricardo Rocha

Technical Report Series: DCC-2005-10

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

The MYDDAS Programmer's Manual

Tiago Soares, Michel Ferreira, Ricardo Rocha
DCC-FC & LIACC
University of Porto, Portugal
{tiagosoares,michel,ricroc}@ncc.up.pt

version 0.1

Abstract

This document describes the MySQL/Yap Deductive Databese System (MYDDAS) interface predicates. It includes description of the native MySQL interface and ODBC interface that can be used to couple Yap Prolog with a number of relational database systems.

1 Requirements and Instalation Guide

To use this interface we must have installed at least one of the following development libraries: the MySQL developement library or the developement library for ODBC (mysql-devel and libodbc-devel, respectively). Also, we must manually edit the `Makefile.in` file, to include the flag `MYDDAS_MYSQL` or the flag `MYDDAS_ODBC`, or even both, if we want to enable both MYSQL and ODBC on YAP. This is done by editing the line:

```
YAP_EXTRAS=@YAP_EXTRAS@
```

to

```
YAP_EXTRAS=@YAP_EXTRAS@ -DMYDDAS_MYSQL -DMYDDAS_ODBC
```

Finally, we must enable the `cut_c` option on YAP compilation by given to configure the option `--enable-cut-c`.

2 MYDDAS MySQL interface

Next we describe the MyDDAS MySQL interface.

2.1 Introduction

The YAP - MySQL interface provides the programmer with two levels of interaction. The first, relation level interface, offers a tuple-at-a-time retrieval of information from the MySQL relations. The second, view level interface, can translate an entire Prolog clause into a single SQL query to the MySQL, including joins and aggregate operations.

This interface allows MySQL relations to be accessed from YAP's environment as though they existed as facts. All database accesses are done on the fly allowing YAP to sit alongside other concurrent tasks.

2.2 Using the Interface

Begin by starting YAP and loading the library:

```
?- use_module(library(myddas_mysql)).
```

This library already includes the Prolog to SQL Compiler described in [2] and [1].

2.3 Connecting to and disconnecting from MySQL Server

```
db_my_open(+,+,+,+,+).
db_my_close(+).
```

Assuming the MySQL server is running, we have an account, we can login to MySQL by invoking `db_my_open/5` as:

```
?- db_my_open(Host,User,Password,Db,Conn).
```

If the login is successful, there will be a response of yes. For example

```
?- db_my_open('kheque.ncc.up.pt',db_user,keyWord,guest,con1).
```

indicates to the runtime system that we want to contact an MySQL server instance on the host `kheque.ncc.up.pt`, to the `guest` database, whose user is `db_user` that as `keyWord` has password. The `con1` atom, will be the identifier of this connection.

To disconnect one connection use:

```
?- db_my_close(Conn).
```

where `Conn` is the identifier of the connection given by db_open/5.

2.4 Accessing an MySQL Relation: Relational Level Interface

```
db_my_import(+,+,+).
```

Assuming we have access permission for the relation you wish to import, you can use `db_my_import/3` as:

```
?- db_my_import(RelationName,PredName,Conn).
```

where `RelationName`, is the name of relation we wish to access, `PredName` is the name of the predicate we wish to use to access the relation from YAP. `Conn`, is the connection identifier. For example, if we wanted to access the relation `Hello World`, using the predicate `helloWorld/N` [1] make:

```
?- db_my_import('Hello World',helloWorld,con1).
yes

?- helloWorld(Number,Name,Letter).
Letter = 'T',
Name = 'Tiago',
Number = 18 ?
yes
```

In this example we assume that `con1` is the identifier for the connection. The `Hello World` relation as three fields. Backtracking can then be used to retrieve the next row of the relation `Hello World`.

Records with particular field values may be selected in the same way as in Prolog. (In particular, no mode specification for database predicates is required). For example:

```
?- helloWorld(Number,'Tiago',Letter).
Letter = 'T',
Number = 18 ?
yes
```

[1] The arity of this predicate would be automatically calculated by `db_my_import`

generates the query

```
SELECT A.Number , "Tiago" , A.Letter
FROM "Hello World" A
WHERE A.Name = "Tiago"
```

2.5 View Level Interface

```
db_my_view(+,+,+).
```

The view level interface can be used for the definition of rules whose bodies includes only imported database predicates (by using the relation level interface) described above.

One can use the view level interface through the predicate `db_my_view/3`:

```
?- db_my_view(PredName(Arg_1,...,Arg_n),DbGoal,Conn).
```

All arguments are standard Prolog terms. Arg_1 through Arg_n defines the attributes to be retrieved from the database, while DbGoal defines the selection restrictions and join conditions. Conn is the connection identifier.

The compiler is the 'Prolog to SQL Compiler' implemented by Christoph Draxler.

In the following, we show the definition of a simple join view between the two database predicates `helloWorld` and `goodbyeWorld`.

Assuming the declarations:

```
?- db_my_import('Hello World',helloWorld,con1).
yes

?- db_my_import('Goodbye World',goodbyeWorld,con1).
yes
```

use:

```
?- db_my_view(middleWorld(A,B),(helloWorld(A,B,C),goodbyeWorld(A,B,C)),con1).
yes

?- middleWorld(Arg1,Arg2).
```

generates the SQL statement:

```
SELECT A.Number , A.Name
FROM "Hello World" A , "Goodbye World" B
WHERE B.Number = A.Number AND B.Name = A.Name AND B.Letter = A.Letter
```

Backtracking, as in relational level interface, can be used to retrieve the next row of the view.

The view interface also supports aggregate functions predicates sum, avg, count, min and max. For example:

```
?- db_my_view(avg(X),(X is avg(Number,A ^ B ^ helloWorld(Number,A,B))),con1).
```

generates the query :

```
SELECT AVG(A.Number)
FROM "Hello World" A
```

2.6 Connecting to an SQL query

```
db_my_sql_select(+,+,?).
```

It is also possible to connect to any SQL query using the `db_my_sql_select/3` predicate which takes an SQL string as its input and returns a list of field values. For example:

```
?- db_my_sql_select(con1,'SELECT * FROM HELLO WORLD',LA).
LA = [18,'Tiago','T'];
LA = etc ...
```

2.7 Insertion of rows

```
db_my_insert(+,+).
```

Assuming you have imported the related base table using `db_my_import/2`, you can insert to that table by using `db_my_insert/2` predicate. The first argument is the imported database predicate and the second argument is the connection identifier. The first argument must be declared with with all of its arguments bound to constants. For example assuming helloWorld is imported through `db_my_import/2`:

```
?- db_my_import('Hello World',helloWorld,con1).
yes

?- db_my_insert(helloWorld('A','Ana',31),con1).
yes
```

This, would generate the following query `INSERT INTO helloWorld VALUES ('A','Ana',31)`, which would insert into the helloWorld, the following row: A,Ana,31. If we want to insert NULL values into the relation, we call `db_my_insert/2` with a uninstanciated variable in the data base imported predicate. For example, the following question on the YAP-prolog system,

```
?- db_my_insert(helloWorld('A',NULL,31),con1).
yes
```

Would insert the row: A,null value,31 into the relation `Hello World`, assuming that the second row allows null values.

```
db_my_insert(+,+,+).
```

This predicate would create a new database predicate, which will insert any given tuple into the database.

```
?- db_my_insert(RelationName,PredName,Conn).
```

This would create a new predicate with name `PredName`, that will insert tuples into the relation `RelationName`. `is` the connection identifier. For example, if we wanted to insert the new tuple :'A',null,31 into the relation `Hello World`, we do:

```
?- db_my_insert('Hello World',helloWorldInsert,con1).
yes
?- helloWorldInsert('A',NULL,31).
yes
```

2.8 Knowing the fields attributes types

```
db_my_get_attributes_types(+,+,?).
```

You also can use the predicate `db_my_get_attributes_types/3`, to know what are the names and attributes types of the fields of a given relation. For example:

```
?- db_my_get_attributes_types('Hello World',con1,LA).
LA = ['Number',integer,'Name',string,'Letter',string] ?
yes
```

where `Hello World` is the name of the relation and `con1` is the connection identifier.

2.9 Knowing the number of fields

```
db_my_number_of_fields(+,+,?).
```

You also can use the predicate `db_my_number_of_fields/3`, to know what is the arity of a given relation. Example:

```
?- db_my_number_of_fields('Hello World',con1,Arity).
Arity = 3 ?
yes
```

where `Hello World` is the name of the relation and `con1` is the connection identifier.

2.10 Describing a relation

```
db_my_describe(+,+).
```

The `db_my_describe/2` predicate dosen't reality returns any value. It simply prints to the screen the result of the MySQL describe command, the same way as describe in the MySQL prompt would.

```
?- db_my_describe('Hello World',con1).
+----------+----------+------+-----+---------+-------+
| Field    | Type     | Null | Key | Default | Extra |
+----------+----------+------+-----+---------+-------+
| Number   | int(11)  | YES  |     | NULL    |       |
| Name     | char(10) | YES  |     | NULL    |       |
| Letter   | char(1)  | YES  |     | NULL    |       |
+----------+----------+------+-----+---------+-------+
yes
```

```
db_my_describe(+,+,?).
```

The `db_my_describe/3` predicate does the same action as `db_my_describe/2` predicate but with one major difference. The results are returned by backtracking. For example, the last query done:

```
?- db_my_describe('Hello World',con1,Term).
Term = tableInfo('Number',int(11),'YES','',null(0),'') ? ;
Term = tableInfo('Name',char(10),'YES','',null(1),'') ? ;
Term = tableInfo('Letter',char(1),'YES','',null(2),'') ? ;
no
```

2.11 Knowing the relations in the MySQL Schema

`db_my_show_tables(+).`

If we need to know what relations exists in a given MySQL Schema, we use the `db_my_show_tables/1` predicate. As `db_my_describe/2`, dosen't returns any value, but prints to the screen the result of the `show tables` command, the same way as it would be in the MySQL prompt.

```
?- db_my_show_tables(con1).
+-----------------+
| Tables_in_guest |
+-----------------+
|  Hello World    |
+-----------------+
yes
```

`db_my_show_tables(+,?).`

The `db_my_show_tables/2` predicate does the same action as `db_my_show_tables/1` predicate but with one major difference. The results are returned by backtracking. For example, the last query done:

```
?- db_my_show_tables(con1,Table).
Table = table('Hello World') ? ;
no
```

2.12 Verifying if is a database predicate

`db_my_is_database_predicate(+,+,+).`

After a while it could be useful to know if a given predicate was asserted by one of the previous predicate. For example, `db_import/3` or `db_view/3`. This predicate tells if a given predicate is a database one.

```
?- db_my_is_database_predicate(helloWorld,3,user).
yes
?- db_my_is_database_predicate(helloWorld,2,user).
no
```

In the example above, we can see that the predicate `user:helloWorld/3` exists, but the predicate `user:helloWorld/2` dosen't.

`db_my_describe(+,+).`

2.13 Other properties

`db_my_verbose(+).`

When we ask a question to YAP, using a predicate asserted by `db_my_import/3`, or by `db_my_view/3`, this would generate a SQL QUERY. If we want to see that query, we must to this at a given point in our session on YAP.

```
?- db_my_verbose(1).
yes
?-
```

If we want to disable this feature, we must call the `db_my_verbose/1` predicate with any other value than 1.

`db_my_module(?).`

When we create a new database predicate, by using `db_my_import/3` or `db_my_view/3` or `db_my_insert/3`, that predicate will be asserted by default on the `user` module. If we want to change this value, we can use the `db_my_module/1` predicate to do so.

```
?- db_my_module(lists).
yes
?-
```

By executing this predicate, all of the predicates asserted by the predicates enumerated earlier will created in the lists module.

If we want to put back the value on default, we can manually put the value `user`. Example:

```
?- db_my_module(user).
yes
?-
```

We can also see in what module the predicates are being asserted by doing:

```
?- db_my_module(X).
X=user
yes
?-
```

`db_my_result_set(?).`

The MySQL C API permits two modes for transfering the data generated by a query to the client, in our case Yap. The first mode, and the *default mode* used by the MYDDAS-MySQL, is `store_result`. This mode copies all the information generated to the client side.

```
?- db_my_result_set(X).
X=store_result
yes
```

The other mode that we can use is `use_result`. This one uses the result set created directly from the server. If we want to use this mode, he simply do

```
?- db_my_result_set(use_result).
yes
```

After this command, all of the database predicates will use `use_result` by default. We can change this by doing again `db_my_result_set(store_result).`

3 MYDDAS ODBC interface

Next we describe the MyDDAS ODBC interface.

3.1 Introduction

The YAP - ODBC interface provides the programmer with two levels of interaction. The first, relation level interface, offers a tuple-at-a-time retrieval of information from the databases relations, witch are connected to a ODBC Driver. The second, view level interface, can translate an entire Prolog clause into a single SQL query to the ODBC, including joins and aggregate operations.

This interface allows ODBC relations to be accessed from YAP's environment as though they existed as facts. All database accesses are done on the fly allowing YAP to sit alongside other concurrent tasks.

3.2 Using the Interface

Begin by starting YAP and loading the library:

```
?- use_module(library(myddas_odbc)).
```

This library already includes the Prolog to SQL Compiler described in [2] and [1].

3.3 Connecting to and disconnecting from ODBC Driver

```
db_odbc_open(+,+,+,+).
db_odbc_close(+).
```

Assuming the ODBC Driver is installed, we have an account, we can login to ODBC by invoking `db_odbc_open/4` as:

```
?- db_odbc_open(ODBCDriver,User,Password,Conn).
```

The ODBCDriver variable must the name of the ODBC Driver. In Linux based systems, this can consulted in the file `/etc/odbc.ini`. If the login is successful, there will be a response of yes. For example

```
?- db_odbc_open(myodbc3,db_user,keyWord,con1).
```

indicates to the runtime system that we want to contact an ODBC Driver instance `myodbc3`, whose user is `db_user` that as `keyWord` has password. The `con1` atom, will be the identifier of this connection.

To disconnect one connection use:

```
?- db_odbc_close(Conn).
```

where `Conn` is the identifier of the connection given by db_odbc_open/5.

3.4 Accessing a Relation Throw a ODBC Driver: Relational Level Interface

```
db_odbc_import(+,+,+).
```

Assuming we have access permission for the relation you wish to import, you can use `db_odbc_import/3` as:

```
?- db_odbc_import(RelationName,PredName,Conn).
```

where `RelationName`, is the name of relation we wish to access, `PredName` is the name of the predicate we wish to use to access the relation from YAP. `Conn`, is the connection identifier. For example, if we wanted to access the relation `Hello World`, using the predicate `helloWorld/N` [2] make:

```
?- db_odbc_import('Hello World',helloWorld,con1).
yes

?- helloWorld(Number,Name,Letter).
Letter = 'T',
Name = 'Tiago',
Number = 18 ?
yes
```

[2]The arity of this predicate would be automatically calculated by `db_odbc_import`

In this example we assume that `con1` is the identifier for the connection. The `Hello World` relation as three fields. Backtracking can then be used to retrieve the next row of the relation `Hello World`.

Records with particular field values may be selected in the same way as in Prolog. (In particular, no mode specification for database predicates is required). For example:

```
?- helloWorld(Number,'Tiago',Letter).
Letter = 'T',
Number = 18 ?
yes
```

generates the query

```
SELECT A.Number , "Tiago" , A.Letter
FROM "Hello World" A
WHERE A.Name = "Tiago"
```

3.5 View Level Interface

`db_odbc_view(+,+,+).`

The view level interface can be used for the definition of rules whose bodies includes only imported database predicates (by using the relation level interface) described above.

One can use the view level interface through the predicate `db_odbc_view/3`:

```
 ?- db_dobc_view(PredName(Arg_1,...,Arg_n),DbGoal,Conn).
```

All arguments are standard Prolog terms. Arg_1 through Arg_n defines the attributes to be retrieved from the database, while DbGoal defines the selection restrictions and join conditions. Conn is the connection identifier.

The compiler is the 'Prolog to SQL Compiler' implemented by Christoph Draxler.

In the following, we show the definition of a simple join view between the two database predicates `helloWorld` and `goodbyeWorld`.

Assuming the declarations:

```
?- db_odbc_import('Hello World',helloWorld,con1).
yes

?- db_odbc_import('Goodbye World',goodbyeWorld,con1).
yes
```

use:

```
?- db_odbc_view(middleWorld(A,B),(helloWorld(A,B,C),goodbyeWorld(A,B,C)),con1).
yes

?- middleWorld(Arg1,Arg2).
```

generates the SQL statement:

```
SELECT A.Number , A.Name
FROM "Hello World" A , "Goodbye World" B
WHERE B.Number = A.Number AND B.Name = A.Name AND B.Letter = A.Letter
```

Backtracking, as in relational level interface, can be used to retrieve the next row of the view.

The view interface also supports aggregate functions predicates sum, avg, count, min and max. For example:

```
?- db_odbc_view(avg(X),(X is avg(Number,A ^ B ^ helloWorld(Number,A,B))),con1).
```

generates the query :

```
SELECT AVG(A.Number)
FROM "Hello World" A
```

3.6 Connecting to an SQL query

`db_odbc_sql_select(+,+,?).`

It is also possible to connect to any SQL query using the `db_odbc_sql_select/3` predicate which takes an SQL string as its input and returns a list of field values. For example:

```
?- db_sql_select(con1,'SELECT * FROM HELLO WORLD',LA).
LA = [18,'Tiago','T'];
LA = etc ...
```

3.7 Insertion of rows

`db_odbc_insert(+,+).`

Assuming you have imported the related base table using `db_odbc_import/2`, you can insert to that table by using `db_odbc_insert/2` predicate. The first argument is the imported database predicate and the second argument is the connection identifier. The first argument must be declared with with all of its arguments bound to constants. For example assuming helloWorld is imported through `db_odbc_import/2`:

```
?- db_odbc_import('Hello World',helloWorld,con1).
yes

?- db_odbc_insert(helloWorld('A','Ana',31),con1).
yes
```

This, would generate the following query `INSERT INTO helloWorld VALUES ('A','Ana',31)`, which would insert into the helloWorld, the following row: A,Ana,31. If we want to insert NULL values into the relation, we call `db_dobc_insert/2` with a uninstanciated variable in the data base imported predicate. For example, the following question on the YAP-prolog system,

```
?- db_insert(helloWorld('A',NULL,31),con1).
yes
```

Would insert the row: A,null value,31 into the relation `Hello World`, assuming that the second row allows null values.

`db_odbc_insert(+,+,+).`

This predicate would create a new database predicate, which will insert any given tuple into the database.

```
?- db_odbc_insert(RelationName,PredName,Conn).
```

This would create a new predicate with name `PredName`, that will insert tuples into the relation `RelationName`. `is` the connection identifier. For example, if we wanted to insert the new tuple :'A',null,31 into the relation `Hello World`, we do:

```
?- db_odbc_insert('Hello World',helloWorldInsert,con1).
yes
?- helloWorldInsert('A',NULL,31).
yes
```

3.8 Knowing the fields attributes types

```
db_odbc_get_attributes_types(+,+,?).
```

You also can use the predicate `db_odbc_get_attributes_types/3`, to know what are the names and attributes types of the fields of a given relation. For example:

```
?- db_odbc_get_attributes_types('Hello World',con1,LA).
LA = ['Number',integer,'Name',string,'Letter',string] ?
yes
```

where `Hello World` is the name of the relation and `con1` is the connection identifier.

3.9 Knowing the number of fields

```
db_odbc_number_of_fields(+,+,?).
```

You also can use the predicate `db_odbc_number_of_fields/3`, to know what is the arity of a given relation. Example:

```
?- db_odbc_number_of_fields('Hello World',con1,Arity).
Arity = 3 ?
yes
```

where `Hello World` is the name of the relation and `con1` is the connection identifier.

3.10 Other properties

```
db_odbc_verbose(+).
```

When we ask a question to YAP, using a predicate asserted by `db_odbc_import/3`, or by `db_odbc_view/3`, this would generate a SQL QUERY. If we want to see that query, we must to this at a given point in our session on YAP.

```
db_odbc_verbose(1).
```

If we want to disable this feature, we must call the `db_odbc_verbose/1` predicate with any other value than 1.

```
db_odbc_module(?).
```

When we create a new database predicate, by using `db_odbc_import/3` or `db_odbc_view/3` or `db_odbc_insert/3`, that predicate will be asserted by default on the `user` module. If we want to change this value, we can use the `db_odbc_module/1` predicate to do so.

```
?- db_odbc_module(lists).
yes
?-
```

By executing this predicate, all of the predicates asserted by the predicates enumerated earlier will created in the lists module.

If we want to put back the value on default, we can manually put the value `user`. Example:

```
?- db_odbc_module(user).
yes
?-
```

We can also see in what module the predicates are being asserted by doing:

```
?- db_odbc_module(X).
X=user
yes
?-
```

References

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

[2] C. Draxler. Accessing Relational and NF^2 Databases Through Database Set Predicates. In *Third UK Annual Conference on Logic Programming*, Workshops in Computing, pages 156–173. Springer Verlag, 1992.