QDL DB

Introduction

The QDL database extension. A module that allows access to various databases and has an extremely simple syntax – there are 6 functions and that's it. (You cold argue there are 4, but that's a philosophical point.) This is designed not to be a full fledged database application, but a tools module that allows for all the basic access to various databases with the grunt work of converting between types as well as having a way to work seamlessly with native SQL types. Typically you would write your database application using this module.

Loading the module

To load the module, issue something like

```
db := j_load('db)
```

Supported functions

Name	Description
`batch_execute(stmt, args.)`	Run the same prepared statement with a stem of arguments.
`batch_read(stmt, args.{,flatten})`	Run the same read repeatedly using a stem of arguments. Optionally trying to simplify the output.
`connect(cfg.)`	Open a connection to the database. All other requests will fail until this is called
`execute(stmt{,args.})`	Execute a statement with no result
`read(stmt{, args.})`	Execute a statement with a result
`update(stmt{,args.})`	Update an entry in the database, getting back the rows changed.

Notes

- Execute, read and update can be run without any argument, or may take a scalar as the argument too. This is interpreted to mean there is a single parameter

Variables

`sql_types.` - a stem of names and integer values,

```
 db#sql_types.
{
 NUMERIC:2,
```

```
 FLOAT:6,
 BLOB:2004,
 LONGVARCHAR:-1,
 CLOB:2005,
 ARRAY:2003,
 BINARY:-2,
 CHAR:1,
 BIGINT:-5,
 TIME:92,
 BIT:-7,
 DATE:91,
 REF:2006,
 SQLXML:2009,
 SMALLINT:5,
 TIMESTAMP:93,
 VARCHAR:12,
 REAL:7,
 VARBINARY:-3,
 DOUBLE:8,
 STRUCT:2002,
 TINYINT:-6,
 INTEGER:4
}
```

These are internal values and should not be altered.

Arguments generally

The statement in each of the calls above is a string. It may be either hard coded such as

```
select * from my_table where id='42'
```

which would be issued as

```
db#read('select * from my_table where id=\'42\'');
```

Or it may be prepared with ? signs replacing the arguments and a list of arguments and possibly their types supplied. If there is a scalar, it is used for the only parameter, E.g.

```
db#read('select * from my_table where id=?', 'client_DB864EF6754');
```

QDL tries to be helpful, in that if you supply no SQL type, it will be inferred, so a string will be treated as if it is a string.

That said, databases can have any number of oddities so if you need a specific SQL type (e.g. you have a column that is a tinyint) then by all means specify it. In this case, if the argument were a tiny int, you would issue

```
db#read('select * from my_table where id=?', [42,sql_types.TINYINT]);
```

More generally, a lit of arguments is of the form

```
[a0,a1,…]
```

where a's are either simple types - long, big decimal, string, boolean or null
or are an explicit record

```
[value, type]
```

In which case the type will be asserted (and the value may be changed too).
E.g.

```
['foo',[12223,sql_types.DATE],['3dgb3ty24fgf',sql_types.BINARY]]
```

would assert the first is a string, convert the second into a date and the 3rd is assumed
to be base 64 encoded and would be decoded and asserted as a byte[]

Prepared statements are also extremely useful so you don't have to do a lot of escaping of quotes. For instance to do a search using a regex might look like

```
    db#read('select client_id from oauth2.clients where client_id regexp ?',
['.*123.*'])
{client_id:oa4mp:/client_id/7142f3461239deb57d98ba3a4636}
```

Remember that SQL engines are not really QDL aware, so if you are trying to store your 1000 digit approximation to pi as a number, the database may simply refuse to accept it or it is free to truncate it, mangle it, *etc*.

Finally, **`read`** always returns a list since there are generally assumed to be multiple results. This also holds if you supply a scalar as the second argument.

Responses generally

A ***stem response entry*** is of the form

```
{column0:value0, column1:value1, …}
```

The response from a read will be either a stem response (if there is a single row returned) or a list of them for multiple rows. The type of the value will be one of the basic QDL types, matched up to the response from the server. Default case has the value as a string.

Only a read will return a response, execute and update return a dummy response of **`true`** if it worked or raise an error if it did not. In the example above, including the response we would have

```
    db#read('select client_id from oauth2.clients where client_id regexp ?',
['.*123.*'])
{client_id:oa4mp:/client_id/7142f3461239deb57d98ba3a4636}
```

In this case, the select statement requested a single column, client_id and that is therefore the only key in the response.

Batch processing generally

There are two batch calls, **`batch_execute`** for updates and inserts *i.e.*, that do not return a result from the database, and **`batch_read`** that does return results. While you could loop and get formally the same result, in practice, connecting to a database is often a very expensive operation that can exhaust system resources, especially for a huge update or collection of inserts. It is far better (as in an order of magnitude faster, at least, for sufficiently large sets) to send everything as a single command to the database and databases are generally very well optimized for this.

Function Reference

batch_read

Description

Do multiple queries on the database in a single request.

Usage

```
batch_read(statement, args.{, flatten})
```

Arguments

statement – a (prepared) statement that will be executed

args. - a stem. This may be a stem of scalars if there is a single parameter in the statement or it may be a stem of lists. In a list, the order of the elements is the order they are used in the prepared statement.

Flatten – (optional) a boolean to control the returned result. If args. is a stem of scalars, *and* there is at most one result per query, *then* setting flatten to **true** will try to return a result matching each element of the stem.

Output

A stem conformable to **`args.`** of results. Remember that each query will return a list of results, so the output of this operation might be quite large depending on your query.

Examples

In this example, a simple prepared statement (single parameter) is evaluated for a general stem. Each entry in the stem is a (trivial) list. The response is a stem each of whose entries is a list of results.

```
   stmt := 'select client_id, creation_ts from oauth2.clients where client_id=?'
   args.:={'zero':['oa4mp:/client/234234'], 'one':['oa4mp:/client_id/5667']}
   db#batch_read(stmt, args.);

{
 zero : [{creation_ts:2023-05-19T05:00:00.000Z,client_id:oa4mp:/client/234234}],
```

```
  one : [{creation_ts:2024-03-21T05:00:00.000Z,client_id:oa4mp:/client_id/5667}]
}
```

E.g. With flatten.

In this case, a list of strings is supplied and the flatten argument is set to true, so any list of results that has a single element is flattened to its contents (hence the name). This allows the case that simple cases stay simple.

```
   args1.:=['oa4mp:/client/234234','oa4mp:/client_id/5667']
   db#batch_read(stmt, args1., true)
[
{creation_ts:2023-05-19T05:00:00.000Z,client_id:oa4mp:/client/234234},
{creation_ts:2024-03-21T05:00:00.000Z,client_id:oa4mp:/client_id/5667}
]
```

batch_execute

Description

Execute multiple prepared statements in a single call.

Usage

```
batch_execute(statement, args.)
```

Arguments

statement – a string that is the prepared statement.

args. - A stem of scalars (if there is a single parameter in the statement) or a list of values which will be used to prepare the statement.

Output

At stem conformable to args., where each element is an integer. If the 0 <= value, then this is the number of rows affected by the statement. If negative, this offers one of two conditions:

- **-2** = the operation worked, but no other information is available
- **-3** = the statement failed, but processing continued.

Examples.

E.g.

Let us use the prepared statement

```
stmt := 'UPDATE my_table set accessed=? where id=? AND (access IS NULL or
create_ts<?)'
```

and we have a large list of values. Each element of the list is a list whose values are used in the prepared statement

```
v.:=[[date_ms(), '7D5EF', date_ms()-2419200000], [date_ms(),'C46AB',date_ms()-2419200000]]
```

(just 2 for this example, but it could be thousands). You issue

```
rc. := batch_execute(stmt, v.)
```

The next example will show possible return codes in more detail.

E.g. Mass delete

This will do a mass delete by a unique id. It uses the fact that the function accepts a list of scalars if there is a single parameter. Mass deletes are a fine usage of this because deletes can be extremely resource intensive[1]

```
   stmt = 'DELETE from my_table WHERE id = ?
   ids.:=['ADC745B','B6434F','C984E875',...];
   db#batch_execute(stmt, ids.);
[1,-2,0,5,-3,…]
```

in this case various return codes are in effect showing the number of affected rows (>= 0) or how processing happened. -2 means it worked, but the system cannot report more, -3 means it failed.

connect

Description

Setup the connection to a database. You must connect to the database before issuing commands or they will fail.

Usage

```
connect(cfg.)
```

Arguments

`cfg.` = stem with connection parameters

Supported values are

Name	Description	Comment
`username`	The user name	
`password`	The password	

1 For most databases, the row to be deleted is copied so that if there is an error, it may be rolled back. In addition, any and all indices are updated. For a large entry with several indices, deletes can be very resource intensive and doing them in batches is therefore often very well optimized in most databases. As in, doing them by hand sequentially with, say, a loop may be order***s*** of magnitude slower than using a batch delete.

database	The name of the database	In Derby this is the path
schema	The schema	
host	The host	
port	The port	Standard ports are 3306 for maria DB and mysql and 5432 for postgres. Derby does not use ports
parameters	Specific connection parameters	These are very vendor specific
useSSL	Use SSL for the connection	Make *sure* you have set up SSL correctly first!
bootPassword	The boot password	Derby only
inMemory	Run in memory only	Derby. Note that this database will vanish as soon as you exit QDL.
type	The type of the connection	One of mysql, mariadb, postgres or derby

Output

Ther result is always **`true`** or an error is raised.

Example

```
    cfg.'username' := 'qdl-user';
    cfg.'password' := 'w00fity';
    cfg.'schema' := 'qdl_test';
    cfg.'database' := 'qdl_test';
    cfg.'host' := 'localhost';
    cfg.'port' := 3306;
    cfg.'type' := 'mariadb';
   db#connect(cfg.);
true
```

This indicates that a connection to the database was made. Here is a test to count the rows in one of the tables

```
     db#read('select count(*) from transactions');
{count(*):161}
```

The result is a stem. Note that the database engine itself returned the name of the result as **`'count(*)'`** and this may vary by vendor. In any case, there are 161 entries in the given table for the given database.

How do I close a connection?

You don't. Connecting to the database means creating a pool that makes connections and destroys them as needed, so there is no single connection to dispose of.

execute

Description

The most general way to execute an SQL statement. The read and update functions break down the two return types so you don't have to test for what you got back.

Usage

```
execute(stmt{,args.})
```

Arguments

`stmt` = the SQL statement

`args.` = (optional) the parameters if the stmt is prepared. Note that a scalar may be used if there is exactly one parameter in the prepared statement.

Output

Either

1. A result list if the statement returns a result, *e.g.* it is a **`SELECT`** statement
2. An integer for the number of rows affected, *e.g.*, it is an **`INSERT`**, **`UPDATE`** or **`DELETE`**

Example

```
   a. := db#execute('select * from ' + cfg.schema +
                    'client_approvals where client_id REGEXP \'.*234.*\'');

print(a\[;5]\client_id)
   0 : myproxy:oa4mp,2012:/adminClient/10ef0068ecb49cef2d1f918a22dc860/1655392348601
   1 : myproxy:oa4mp,2012:/adminClient/1b493909913b4348bbc997623176783e/1665765523406
   2 : myproxy:oa4mp,2012:/adminClient/2341e2ad8a643241eadbf05fa801b68c/1669828613898
   3 : myproxy:oa4mp,2012:/adminClient/23479e753498a4d539a11aba3bf9d4a4/1708722611415
   4 : myproxy:oa4mp,2012:/adminClient/2371b66f9832347b8a96353d362c589c/1687546274421
```

Here a possibly large set is selected and only the identifiers of the first 5 are printed. Note that if the function had been **`read`** instead of **`execute`**, this would be the same.

read

Description

Execute an SQL statement that returns a list of results.

Usage

```
read(stmt{, args.})
```

Arguments

`stmt` = the SQL statement

`args.` = (optional) the list of parameters if the stmt is prepared. This may be a scalar if there is exactly one parameter.

Output

A list of results. This list may be empty if there are no results.

Example

```
   id := 'client_id:/3467653';
   db#read('select * from ' + cfg.schema + '.clients where client_id=?',id);
[
   {client_id : 'client_id:/3467653',
   … // lots more!
]
```

update

Description

Update a selection of rows. Note that this does not return a result set, so issuing this with a SELECT clause or some such will fail.

Usage

```
update(stmt{,args.})
```

Arguments

`stmt` = the SQL statement

`args.` = (optional) the list of parameters if the **`stmt`** is prepared. This may be a scalar if there is exactly one parameter.

Output

An integer reflecting the number of rows affected.

Example

This updates a single column to the value {},

```
   db#update('update ' + cfg.schema +
             '.clients set cfg=? where INSTR(cfg, \'{"isSaved\') >0',
             '{}');
93
```

This states that 93 rows in the database were updated. Note that since there was a single parameter, the last argument is just a scalar.