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

Types and conversions

Here is a table of SQL column types (these are pretty standard across various vendors) and the type of value QDL returns on reading. When setting these, type information should be supplied and various conversions will be done. Do note that you should keep in mind the storage type. While you may set a column of type `tinyint` to an arbitrary value in QDL, the database engine might well truncate it or fail,

depending on the vendor. Also the same for numeric or decimal types, since these are usually very constrained in a database, but QDL allows for arbitrary precision. It is up to the programmer to make sure that you are sending sensible things to the database.

Default conversions SQL to/from QDL

SQL	Type	QDL
BIGINT	-5	integer
BINARY	-2	Base 64 string
BIT	-7	boolean
BLOB	2004	Base 64 string
BOOLEAN	16	boolean
CHAR	1	string
CLOB	2005	string
DATE	91	ISO date string
DECIMAL	3	decimal
DOUBLE	8	decimal
FLOAT	6	decimal
INTEGER	4	integer
LONGNVARCHAR	-16	string
LONGVARBINARY	-4	Base 64 string
LONGVARCHAR	-1	string
NCHAR	-15	string
NCLOB	2011	string
NULL	0	null
NUMERIC	2	decimal
NVARCHAR	-9	string
REAL	7	decimal
SMALLINT	5	integer
TIME	92	ISO date string
TIMESTAMP	93	ISO date string
TIMESTAMP_WITH_TIMEZONE	2014	ISO date String
TIME_WITH_TIMEZONE	2013	ISO date String
TINYINT	-6	integer
VARBINARY	-3	Base 64 string
VARCHAR	12	String

Notes:

- Base 64 strings assume that the data returned from the database is indeed binary.
- Date – This zeros out the time portion. Setting the time portion will be lost (for most database engines).
- Time – This sets the date to Jan 1., 1970 and simply has the time portion. On save, the date will be lost

- Time, timestamp with timezone. These will be correctly converted into QDL ISO dates, but it is very hard to have the database engine preserve them. (Most implementations simply suggest you have a second column for the timezone and don't have the database do anything.) Given the vagaries of various database engines handling these, you should probably just avoid setting them.
- The sql_types. Stem has other values that the database engine may recognize. If it is not in the above table, QDL does not support it.

Specific conversions from QDL to/from SQL

QDL also supports specifying the QDL type you want and converting it from SQL on reads and back to the correct SQL type for the database on writes. You create a stem with the conversions you want and pass that as an argument.

Basic types

- string – the QDL is a string. If the if the column type is binary, then the string is stored as bytes. If the column is a string, then no conversion happens.
- date – QDL is an ISO 8601 date, the column type is long, and the date is converted
- json – QDL is a stem, set or list and the column type is a string. The QDL object is converted to JSON. In the case of lists and sets, the result is a JSON array.
- input_form – QDL is arbitrary (incudes stems, sets) and the SQL column type is a string.
- default – (implied at all times) use the default QDL to SQL conversions. This mostly is for completeness.

Binary types

QDL Type	
`json`	Stem → JSON object List → JSON array set → JSON array
`input form`	Stem, list, set → input form
`string`	
`date`	
`none`	

In addition, some calls may take a flag for **`force_column_case`** in the stem. The reason is some database engines (such as Derby, PostgreSQL) allows for case sensitive column names. For the table metadata, if case is not specified, they are usually converted to all lower or upper case (depends on database engine and possibly the release). So that you may work with any case you wish in stems,

setting this true normalizes cases. If you require case sensitive columns,, then this should be *false*, which is the default.

Variables

```
print( db#sql_types.)
                    ARRAY : 2003
                   BIGINT : -5
                   BINARY : -2
                      BIT : -7
                     BLOB : 2004
                     CHAR : 1
                     CLOB : 2005
                     DATE : 91
                  DECIMAL : 3
                   DOUBLE : 8
                    FLOAT : 6
                  INTEGER : 4
             LONGVARBINARY : -4
               LONGVARCHAR : -1
                    NCHAR : -15
                    NCLOB : 2011
                     NULL : 0
                  NUMERIC : 2
                 NVARCHAR : -9
                     REAL : 7
                      REF : 2006
                 SMALLINT : 5
                   SQLXML : 2009
                   STRUCT : 2002
                     TIME : 92
                TIMESTAMP : 93
   TIMESTAMP_WITH_TIMEZONE : 2014
        TIME_WITH_TIMEZONE : 2013
                  TINYINT : -6
                VARBINARY : -3
                  VARCHAR : 12
```

These are internal values and should not be altered.

```
print(db#qdl_types.)
         date : 3
   input_form : 2
         json : 1
         none : -1
       string : 4
```

Prepared statements

A *prepared statement* is one of an SQL DELETE, SELECT, UPDATE, INSERT statement or a stored procedure. A *parameter* is a placeholder denoted with the question mark, ? that represents a variable. You supply these values in a list and as the statement is interpreted, each one is replaced by its value in

order. You may not parameterize database constants like the table name, SQL commands and keywords generally. Good example:

```
SELECT * FROM products WHERE price > ?
```

Bad example:

```
SELECT * FROM ? WHERE ? > ?
```

This fails because the table and column cannot be parameterized. QDL simply passes through and prepared statements to the underlying database engine, allowing for the full complexity to be used.

Advantages

One of the most common attacks on an SQL server is an injection attack – adding malicious code to a string in order to take over or otherwise misuse the server. Prepared statements are frequently much simpler to write, so the chances of having to track down an error in a very complex statement are lessened. Also, prepared statements are usually very well optimized for most database engines, so there can be a noticeable speed improvement using them.

Example: A simple injection attack.

Let us say that you were going to get the user's username and password from the command line and just create the following query manually:

```
enter username> bob
enter password> djHSD94j

my_query := 'SELECT * FROM users WHERE username = \'' + username + '\' AND password
= \'' + password + '\'';

// then send the query to the database
   db#read(my_query);
```

(Assuming username and password contain those values.) This is handed directly to the database. If the user wanted to bypass having a password, they could simple insert an SQL statement like so:

```
enter username> admin
enter password> '' OR 1=1 --'
```

After creating the query it is

```
   my_query
SELECT * FROM users WHERE username = 'admin' AND password = '' OR 1=1 --'
```

What this does is

1. The username is set to admin (they don't know for sure, so this is fishing and they may have to retry this a few times.)
2. first quote `'` ends the statement, so the password is empty

3. `' OR 1=1 --'` adds a condition the is *always* true, then – terminates the rest of the statement (since the attacker does not know what the query is, this just throws the rest away)
4. The statement executes and will return the admin's record with all information, allowing the attacker to take over the system.

By using parameters, the raw quotes (which are syntactically part of the SQL statement) are escaped and therefore not executable. The correct way to do this in QDL is

```
enter username> bob
enter password> djHSD94j

   my_query := 'SELECT * FROM users WHERE username=? AND password=?';
   db#read(my_query, [username, password]);
```

More about arguments.

The statement a call 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 count of how many rows were affected or another code (see entries for those for details) 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.

How arguments and responses are mapped.

If you do a read then the keys of the argument are used as the keys of the list of result stems. So consider the following. It reads with a prepared statement, hence has positional arguments, and the result set column named **`create_date`** requires conversion to being an ISO date:

```
now := date_ms();
query := 'select last_name, first_name, create_date, id from qdl_test.doc_test ' +
       'where ?<create_date AND create_date<?';
args2.'yesterday':= [now - 24*3600*1000, now];       // 1 day = 24 hours ago
args2.'last_week' := [now - 7*24*3600*1000, now];    // 7 days ago
args2.'last_month' := [now - 30*24*3600*1000, now]; // 30 days ago
           qt2.0 := [db#sql_types.'BIGINT', db#qdl_types.'date']; // positional argument 0
           qt2.1 := [db#sql_types.'BIGINT', db#qdl_types.'date']; // positional argument 1
```

```
qt2.'create_date' = db#qdl_types.'date';                              // column name in result


db#read(query, args2., qt2.);
```

Since the structure of the arguments (arg2.) is

```
{'yesterday' : list…
 'last_week' : list..
 'last_month' : list...
}
```

The resulting output will be structured the same as

```
{'yesterday' : list…
 'last_week' : list..
 'last_month' : list...
}
```

but each list here is a result stem, so in more detail it might be

```
{'yesterday' : [{create_date:2026-01-29T22:58:42.617Z, first_name:Bob, id:id-AITSEV, last_name:Smith},
{create_date:2026-01-16T04:39:42.715Z, first_name:Fionna,...}]
 'last_week' : [{...
 'last_month' : [{...
}
```

note the create_date column, stored as a BIGINT in the database, was converted to an ISO string. (Do note that the order of the columns is specified and followed in the SQL query, but there is no canonical ordering in stems, hence printing or displaying them is not guaranteed to preserve column order. On the other hand, updating a database entry would be done by naming the columns, and QDL will manage that so you don’t really have to think about column order.)

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 (*aka* cause a at least the connection to crash), 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_execute

Description

Execute multiple prepared statements in a single call. Remember that in the prepared statement, a **?** represents each argument in order. This is for statements that do not return an entry in the database. For that, use **`batch_read`**.

Usage

```
batch_execute(statement, args.{,conversions.})
```

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.

Conversions. – a list of pairs, **`[sql_type, qdl_type].`** Each element at the given index is applied to the prepared argument of the same index. See the section on Types and Conversions. Sparse lists are allowed, so Both of the following are equivalent

```
 qt. := {4 : [-9, 2]}
qt.4 := [-9, 2]
```

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.

Example using conversions

In this example, we are going to insert a stem which is converted to JSON and stored as a string. There is one value to convert. For simplicity, the stem is the same.

```
stmt := 'insert into qdl_test.xxx (id,current_state) values (?,?)';
qq. := {'a':123,'b':{'c':234,'d':'SPQR'}};
c.1 := [12,1]; // is an SQL string (12), the first element has type JSON (1)

db#batch_execute(stmt,
                 [['a-8',qq.],
                  ['a-9',qq.],
                  ['a-10',qq.],
                  ['a-11',qq.]],
                 c.);
[1,1,1,1]
```

The output shows that for each statement, 1 row in the database was updated.

close

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.

Description

Close a connection. After this call, any attempts to access the database will fail.

---

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.

Usage

```
close()
```

Arguments

None.

Output

A boolean. True if the connection is closed, false otherwise.

Example

```
  db#close()
true
```

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	
`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.

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{.}}{,qt{.}})
```

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.

`qt.` = (optional) a stem of key value pairs for columns and constants as found in the **`qdl_types.`** Variable.

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.

get_table_metadata

Description

Get the metadata for a given table. This will have the column names, their SQL types, primary key and other useful information.

Usage

```
get_table_metadata(table_name)
```

Arguments

`table_name` = the name of the table. The database and schema are in the connection information, so you don't need to fully qualify it.

Output

A stem with the following structure

Key	Type	Description
`auto_commit`	`boolean`	True if auto commit for this table is enabled, false otherwise.
`catalog`	`string`	The catalog for this table (optional)
`columns`	`stem`	A stem of whose keys are the column names and the values are the sql types
`db`	`stem`	Information about the database itself. See note below
`primary_key`	`List`	A list of column names that comprise the key.
`schema`	`string`	The schema for this database (optional)
`table_name`	`string`	The same as the argument. It identifies this table.

The **`db`** stem contains the following information and lets you customize your code to the specific vendor and version you need, since that is a fact of life with databases.

Key	Type	Description
`major_release`	`integer`	The major release number as an integer
`minor_release`	`integer`	The minor release number as an integer
`vendor`	`string`	The name of the vendor, *e.g.*, **`mySQL, derby`**, ...
`version`	`string`	The platform version, *e.g.*, **`8.0.44-0ubuntu0.24.04.1`**. This includes the major an minor release values in the string typically, though parsing it may be nearly impossible, hence the values are directly included in this stem.

```
print(get_table_metadata('my_table'))
   auto_commit : true
       catalog : qdl_test
       columns : {spreadsheet:-4, is_copy:-7, create_date:91, description:-1,
                  authors : -1, comment:-1, last_modified:92, id:12}
   primary_key : [id]
    table_name : my_table
```

Notes. There are several vendor oddities with metadata. Generally MySQL and MariaDB are well behaved. But

PostgreSQL – the catalog is always returned as public (since version 7) to solve a backwards compatibility issue. QDL will not return that (unless it ever changes)

Derby – The metadata will not reliable return the primary key for the database. You may have to supply this. Also, the default schema for every embedded database is the user's name, so if you are issuiong SQL statements that do not, say, fully qualify the tablename, you may get an error claiming the table does not exist in a strange schema.

Examples

qdl_create

Description

Create a new entry in the database from either a stem or list/stem of them (each entry then is a new entry in the database).

Usage

```
qdl_create(table_name | md. {,arg.}{,qt.})
```

Arguments

`table_name` = a string that is the table name.

`md.` = table metadata. This is the output from the **`get_table_metadata`** call. (Derby users may have to set the primary key). General;ly you only need the table name and the system will do the lookup. Pass this in if you have to tweak the metadata somehow. (Some vendors do strange things with the metadata.)

`arg.` = A stem or list of stems. If a stem, then that is assumed to be a complete entry in the database. If a list, then each entry is a stem that corresponds to a single entry in the database.

`qt.` = A stem of (column_name, qdl_type) pairs. If the database requires forcing column case, then it may have the flag **`force_column_case`** set to true.

Output

If the **`arg.`** **i**s a stem, the output is a scalar showing the number of rows affected or other status. If **`arg.`** is a list, each element of the list is the output value for that stem.

Example

The assumption is that you have a stem, a., that contains information about cars:

```
          a.'parts' := {'engine':'V-8',… };
      a.'car_types' := {'sedan', 'compact', …};
   a.'last_updated' := date_iso();
            a.'id' := random_string(16);

   qt. := {'parts':db#qdl_types.'json',
           'car_types':db#qdl_types.'input_form',
           'last_updated':db#qdl_types.'date',
           'force_column_case' : true};

   db#qdl_create('qdl_test.vehicles',a., qt.);
1
```

This creates a new element. Note that the stem contains standard QDL types. The conversion stem, qt., specifies that the car_types is stored in a string that is input form, the stem for parts is converted to JSON, etc. The date is an iso string in QDL, but is stored in the database as a 64-bit integer. It also informs us that that the database allows for case sensitive columns which are not used, but there is no canonical way to tell what case is used when querying the metadata, so it is all normalized. (Remember that the insert statement is created from the stem and introspection on the SQL column types for you, so column case might be important.) This means that you can ignore column case and just use whatever you want.

The result shows a single record was added successfully, as expected.

qdl_to_sql

Description

Convert QDL values to SQL. This is useful if you are creating, say, a batch statement that is complex and need to do the conversions manually.

Usage

```
qdl_to_sql(value, qdl_type, sql_type)
```

Arguments

value – the value to convert

qdl_type = the constant that determines how the conversion is done.

sql_type – the target SQL type

Output

The new (scalar) value that SQL can understand.

Example

```
  db#sql_to_qdl(date_iso(), db#qdl_types.’date’, db#qdl_types’BIGINT’)
1769629736226
```

In this case, an ISO date string is converted to an integer.

qdl_update

Description

Update an existing entry or entries in the database, using a stem or resp. list of stems.

```
qdl_update(table_name | md. {,arg.}{,qt.})
```

Arguments

`table_name` = a string that is the table name.

`md.` = table metadata. This is the output from the **`get_table_metadata`** call. (Derby users may have to set the primary key). General;ly you only need the table name and the system will do the lookup. Pass this in if you have to tweak the metadata somehow. (Some vendors do strange things with the metadata.)

`arg.` = A stem or list of stems. If a stem, then that is assumed to be a complete entry in the database. If a list, then each entry is a stem that corresponds to a single entry in the database. Note that the primary key (may be compound) must be present. However, you do not need to update every field.

`qt.` = A stem of (column_name, qdl_type) pairs. If the database requires forcing column case, then it may have the flag **`force_column_case`** set to true.

Output

If a stem, then the number of rows affected. If a list, a list with the row count per entry.

Example

```
          a.'parts' := {'engine':'V-8',… };
             a.'id' := 'I13dpxYil805Tc0i';

   qt. = {'parts':db#qdl_types.'json',
          'force_column_case' : true};

   db#qdl_update('qdl_test.vehicles',a., qt.);
1
```

In this example, only a parts list is updated. The primary key for the database is **id**, so that must be included to create the insert statement.

read

Description

Execute an SQL statement that returns a list of results.

Usage

```
read(stmt{, arg{.}} {,qdl_type{.} | flatten})
```

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.

`qdl_type{.}` = (optional) a (possibly sparse) list of qdl types to convert the SQL to.

`flatten` = (optional) A flag to unwrap singleton results, so reduce one dimension of the resulting 2xn list. It may also be included in the **`qdl_type.`** argument.

Output

A list of lists or stems. Each entry corresponds to a select query run with parameters. The lists 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!
   ]
]
```

In this case, the query returns a single result. If you want to unwrap the singleton, then add the flag:

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

Note that you may supply a stem and each result will be populated with those keys. Here is a full example, with a QDL type that specifies an SQL big integer (64 bit) in the database should be converted to an ISO 8601 date.

sql_to_qdl

Description

Convert an SQL data type to a QDL type. This permits turning, *e.g.* JSON stored in a string into a stem.

Usage

```
sql_to_qdl(value{.}, qdl_type{.})
```

Arguments

value – a scalar or stem of base values returned from an SQL database

qdl_type – the target type to convert to

Output

The converted QDL values.

Example

update

Description

Update a selection of rows using a prepared statement. Note that this does not return a result set, so issuing this with a SELECT clause or some such will fail. You must construct the statement, unlike **qdl_update** which takes a stem and creates it for you.

Usage

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

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.

qt. = (optional) A stem of (column_name, qdl_type) pairs. If the database requires forcing column case, then it may have the flag **force_column_case** set to true. If there is a single parameter, this may be a scalar.

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.

Appendix: A complete test database example

In this appendix I'll give the setup for a test database and the QDL script to populate it with test data, then query and update it. This is done in MySQL so if you want to use another database, you may need to change something.

SQL creation commands

In this case, a database with schema named qdl_test has been created and a user named qdl_tester exists. To create the test table (used in some examples in this blurb)

```
CREATE TABLE qdl_test.doc_test
(
   id          VARCHAR(128) PRIMARY KEY,
   last_name    VARCHAR(250),
   first_name   VARCHAR(250),
   state        TEXT,
   create_date   BIGINT
);

GRANT All PRIVILEGES ON qdl_test.doc_test TO 'qdl_tester'@'localhost';
```

Populating the table with random data

Creating data

This takes a number, count, and created random entries. These are stored in the variable arg., which should be passed in to the database.

```
count := 10; // total number of entries.
   r. := random_string(6, count); // random strings
  now := date_ms();
    i := 0; // counter;
 arg. := [;count]; // output, allocate space

while[r ∈ r.]
   do[
        date := now - mod(abs(random()), 30*24*3600*1000); //random date last month
          j. := {'a':'a-', 'b' : {'c':'c-'}} + r; // random JSON
      arg.i := ['id-' + r,
                'last-'+r,
                'first-'+r,
                to_json(j.),
                date
               ];
    i++;
]; // end do
```

This creates completely random data and in particular, the date is some random time within the last 30 days. This lets us have somewhat interesting queries.

Random values are in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, hence we make sure it's non-negative with the ***abs*** *function, then reduce them to the proper range between 0 and 2592000000 with the* ***mod*** *function.*

Creating database entries

In this case, cfg. (the database connection configuration) is created from a file. Such a configuration is just a stem that looks like

```
{'database':'qdl_test',
 'host':'localhost',
 'password':'XXXXXXX',
 'port':3306,
 'schema':'qdl_test',
 'type':'mysql',
 'useSSL':false,
 'username':'qdl_tester'
}
```
(Tweak the entries for your specific database.)

Now, load the database module, and connect

```
   db := j_load('db');
   cfg. := interpret(file_read('/home/ncsa/dev/csd/config/mysql-connector.qdl'));
   db#connect(cfg.);
```
Next, we create the actual insert statement. Since this may be a large operation, we use the batch mode.

```
   stmt := 'insert into qdl_test.doc_test (id, last_name,first_name, state,'+
           'create_date) values (?,?,?,?, ?)';

   db#batch_execute(stmt, arg.); // should return list of 1's, for success
[1,1,1,1,1,1,1,1,1,1]
```
Note we broke up the stmt variable over two lines so it is easier to read. Note that we sent along integers as the date which are the type in the database so, in this case, no conversions of any sort are needed.

Having QDL create the items

There is a function named **`qdl_create`** that will take a stem (just a result stem) and, as long as the primary key is unique, create a new database entry from that. If there are multiple entries, it will batch them up. So we could have also written

```
   qq. := {'create_date':'2026-02-05T12:09:40.457Z',
           'first_name':'Robespierre',
           'id':'id321',
           'last_name':'Braithewait'};
   db#qdl_create('qdl_test.doc_test', qq., {'create_date':db#qdl_types.'date'})
1
```
The above loop could be easily retooled to create a list (or even stem) of stems and would create a batch to update. This is great for very generic creation tasks. Again it uses the coloumn names as the keys. Read the specific entry in the documentation for this.

Reading results.

We are going to read a set of results from the database.

Getting the size of the database

```
   db#read('select count(*) from qdl_test.doc_test')
{count(*):25}]
```

This means that the operation (listed as the key) returned a value of 25, so there are 25 items in the database.

A more complex query

This example is intended to show how reading a more complex set of results works.

```
now = date_ms();
query = 'select last_name, first_name, create_date, id from qdl_test.doc_test ' +
        'where ?<create_date AND create_date<?';
```

Create the parameters (which replace the ?'s in order). These are for the last 24 hours, last 7 days and resp. last 30 days.

```
   args2.'yesterday'= [now - 24*3600*1000, now];
   args2.'last_week' = [now - 7*24*3600*1000, now];
   args2.'last_month' := [now - 30*24*3600*1000, now];
```

We want the dates converted to ISO dates (since dates as milliseconds are not really readable)

```
   qt2.'create_date' = db#qdl_types.'date';   // column name in result

   results. := db#read(query, args2., qt2.);
   results.
[yesterday: [{id:id-aJWn9, create_date:2026-02-01T08:44;17.123Z,…],
 last_week: […],
 last_month: […]
]
```

Do note that while the parameters must be in strict order, the result, being a stem, has no canonical order. Had we given args2. as a list instead, the order of the queries would be preserved.

Updating results.

Simplest update

There are a couple of ways to update the database. The simplest is to take a result stem, tweak it, then have the system update it, using the **`qdl_update`** call. You don't have to write the update statement. The way it works is that you supply the table name the system gets the metadata including the SQL types for the columns then constructs the insert statement and runs it.

Example. Update a single result.

In this case, we will just alter the first result by resetting the name and issue the update.

```
   results.'yesterday'.0.'first_name':= 'Bob';
 db#qdl_update('qdl_test.doc_test',
               results.'yesterday'.0,
               {'create_date':db#qdl_types.'date'})
1
```

The output shows a single record was updated. We also supply the conversion constant as the last argument (note it is in a stem, keyed by column name).

Note: you do not need to send the whole entry, but you do need to supply the primary key. If you need a more exotic update, you will have to write it. This allows you to just send the bare minimum. Again, this updates a single record based off the primary key, so anything else you have to do, it's a convenience. Please read the documentation for **`qdl_update`** for more details.