RogueWave SOFTWARE | SourcePro

DB ACCESS MODULE FOR MYSQL ENHANCEMENTS

This white paper describes enhancements made to the DB Access Module for MySQL for the SourcePro C++ 2018.1 release, including bulk insertion, stored procedures, and parameter binding.

OVERVIEW

SourcePro DB provides a layer of abstraction that allows an application to migrate to a different database server with minimal modifications, all while maintaining a high degree of performance. With the 2018.1 release, the DB Access Module for MySQL functionality includes parameter data binding, stored procedures, and bulk insertion.

BINDING OF DATA

In SourcePro 2016.3, all bound data are translated into literal SQL strings when sending data to the server. Data such as *RWDateTime* and *double* values are converted into strings, then converted back into the MySQL datatype during execution of the statement on the server side. In addition, a bound string is also converted into literal SQL text and can be inspected with no obfuscation.

The following code snippet illustrates data binding.

```
RWDBTable table = db.table("table");
RWDBInserter inserter = table.inserter();
RWCString value;

inserter << &value;

value = "secret text";

inserter.execute();
```

For SourcePro 2016.3, the SQL sent to the server would look like this:

```
INSERT INTO <TABLENAME> VALUES("secret text")
```

SourcePro 2018.1 no longer translates input binding of data. All bound data are sent in binary format to the server, avoiding double conversion as well as sending the data to the server un-obfuscated. Given the above example, the SQL sent to the server for 2018.1 would look like this:

```
INSERT INTO <TABLENAME> VALUES(?)
```

STORED PROCEDURES

Stored procedure functionality in the DB Access Module for MySQL now includes parameter definitions, the ability to create, call, and drop stored procedures, and the ability to retrieve stored procedure text. These enhancements further reduce the barrier when switching to MySQL from another database.

The following is an example that illustrates:

- Creating a stored procedure that takes an input and output parameter
- Executing the stored procedure, including binding of the parameters
- Obtaining the result returned by the stored procedure and retrieving the output parameter
- Dropping the stored procedure
- Example output

Create a stored procedure that takes an input and an output parameter

```
RWDBDatabase db = /* initialize database */
RWDBConnection conn = db.connection();

const RWCString body(
    "begin \n"
    "  if barcode = 1 then set price = 'newValue';\n"
    "  else set price = null;\n"
    "  end if;"
    "  select 123456;"
    "end"
);

RWDBColumn barcodeParameter;
barcodeParameter.name("barcode").type(RWDBValue::Int);
barcodeParameter.paramType(RWDBColumn::inParameter);

RWDBColumn priceParameter;
priceParameter.name("price").type(RWDBValue::String);
priceParameter.paramType(RWDBColumn::outParameter);

RWDBSchema params;
params.appendColumn(barcodeParameter);
params.appendColumn(priceParameter);

db.createProcedure("proc", body, params, conn);
```

Execute the stored procedure and bind the parameters

```
RWDBStoredProc proc = db.storedProc("proc");
int barcode = 1;
RWCString price("oldValue");
proc << barcode;
proc << &price;

RWDBTable outTable = proc.execute(conn).table();
```

Obtain the stored procedure's result and retrieve the output parameter

```
RWDBReader reader = outTable.reader();
int output;
while (reader()) {
    reader >> output;
    std::cout << "Procedure result set output: " << output << std::endl;
}
```

```
proc.fetchReturnParams();
std::cout << price << std::endl;
```

Drop the stored procedure

```
proc.drop(conn);
```

Example output

```
Procedure result set output: 123456
newValue
```

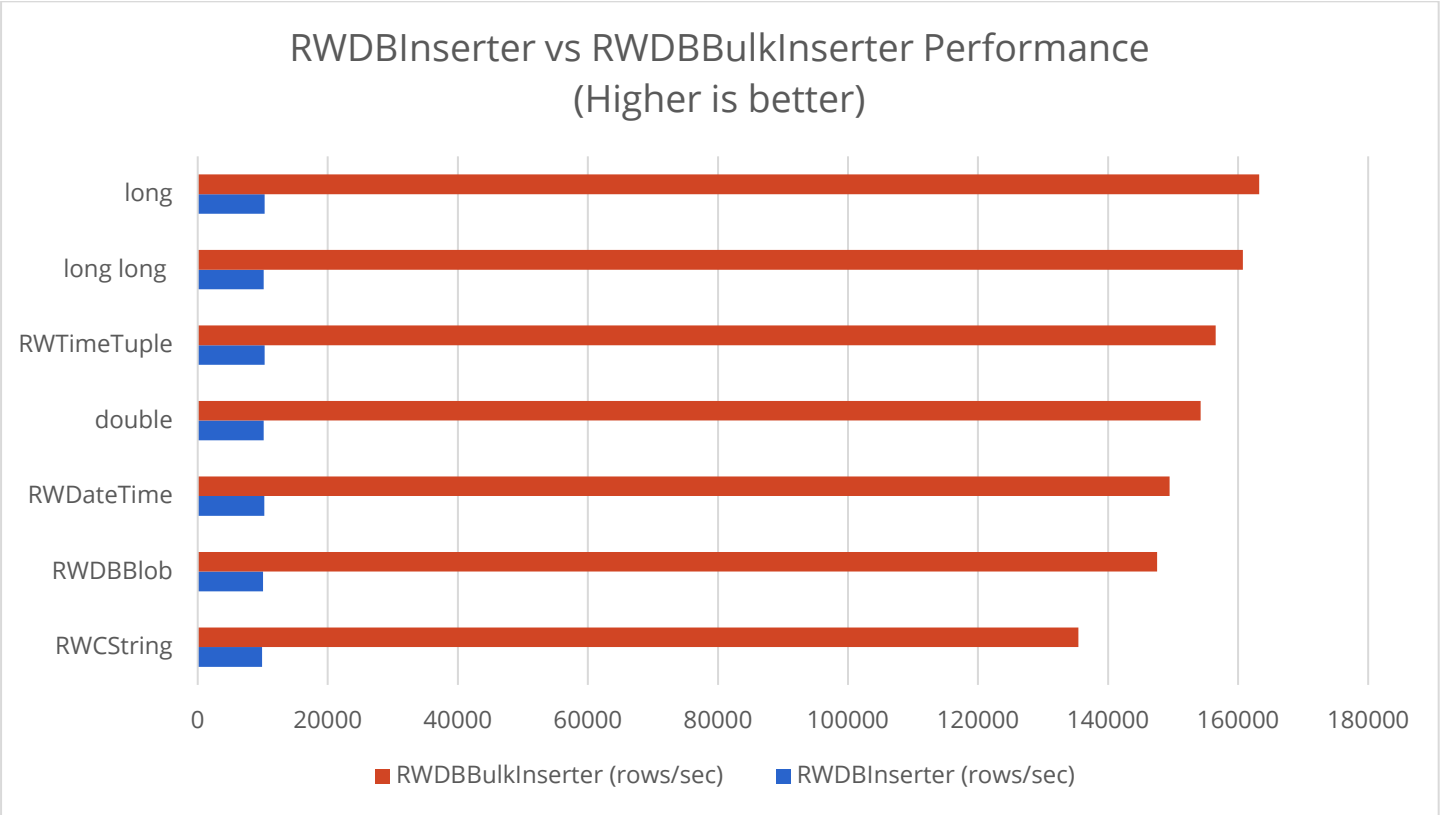
BULK INSERTION

Bulk insertion offered by the *RWDBBulkInserter* provides a significant performance improvement over *RWDBInserter* when inserting a large amount of data. The following code snippet illustrates a simple bulk insertion:

```
RWDBDatabase db = /* initialize database */

RWDBConnection conn = db.connection();
RWDBTable table = db.table("mytable");
RWDBBulkInserter inserter = table.bulkInserter(conn);

int data[10000];
size_t bulkSize = 1000;
size_t numberOfRows = 10000;

RWDBTBuffer<int> buffer(bulkSize);
inserter << buffer;
size_t rows = 0;
do {
    size_t size = min(bulkSize, numberOfRows - rows);
    for (size_t item = 0; item < size; ++item) {
        buffer[item] = data[rows + item];
    }
    RWDBResult res = inserter.execute(size);
    rows += res.rowsAffected();
} while (rows < numberOfRows);
```

The performance when using *RWDBBulkInserter* compared to the regular inserter is improved, in some cases up to 15 times faster.

This graph shows a comparison between the regular inserter compared to the bulk inserter available in SourcePro 2018.1.



Using bulk insertion requires some planning to determine the size of the array to send when invoking the execute function. Typically, the full length of the array can be passed to the execute function. However, if the array size exceeds MySQL's maximum size, the execution call will fail with a "too many placeholders" error. In this case, send a subset of the rows across multiple executions, also known as bulk sizing. The bulk size value should be tuned to your application and environment to maximize performance.

SUMMARY

The DB Access Module for MySQL includes enhanced functionality with SourcePro 2018.1. This includes new bulk insertion functionality via *RWDBBulkInserter* that produces a performance gain of up to 15 times faster than the standard *RWDBInserter*. It also includes stored procedure support via *RWDBStoredProc* to further minimize code changes needed when changing to MySQL from another database. Additionally, it allows sending bound data as binary to reduce unnecessary type conversions between client and server and to provide the added benefit of obfuscation.

APPENDIX

This appendix includes examples used in performance gathering for *RWDBInserter* and *RWDBBulkInserter*.

bulkInsertVsInsert.cpp

```
#include <rw/db/db.h>
#include <rw/db/tbuffer.h>
#include <rw/timer.h>
#include <rw/tvordvec.h>
#include <rw/tools/scopeguard.h>
#include <iostream>

using namespace std;

static const RWCString TABLENAME = "TESTTABLE";
static size_t ROWS = 10000;

void
outputStatus(const RWDBStatus& aStatus)
{
    // Print out the error.
    cerr << "Error code:       " << (int) aStatus.errorCode() << endl
         << "Error message     " << aStatus.message() << endl
         << "Is terminal:      " << (aStatus.isTerminal() ? "Yes" : "No")
         << endl
         << "Vendor error 1:   " << aStatus.vendorError1() << endl
         << "Vendor error 2:   " << aStatus.vendorError2() << endl
         << "Vendor message 1: " << aStatus.vendorMessage1() << endl
         << "Vendor message 2: " << aStatus.vendorMessage2() << endl;
}

void setup(RWDBDatabase& db, RWDBConnection& conn)
{
    RWDBSchema schema;
    schema.appendColumn("c1", RWDBValue::Int);

    RWDBStatus stat;
    if (db.table(TABLENAME).exists(conn)) {
        stat = db.table(TABLENAME).drop(conn);
    }
    stat = db.createTable(TABLENAME, schema, conn);
}

void testBulkInsert(RWDBTable& table, RWDBConnection& conn)
{
    RWDBBulkInserter inserter = table.bulkInserter(conn);

    conn.beginTransaction();
    RWScopeGuard sg = rwtMakeScopeGuardM(conn,

(RWDBStatus(RWDBConnection::*)(void))&RWDBConnection::rollbackTransaction);

    int data[10000];
    size_t bulkSize = 1000;

    RWDBTBuffer<int> buffer(bulkSize);
    inserter << buffer;
    size_t rows = 0;
```

```
    do {
        size_t size = min(bulkSize, ROWS - rows);
        for (size_t item = 0; item < size; ++item) {
            buffer[item] = data[rows + item];
        }
        RWDBResult res = inserter.execute(size);
        rows += res.rowsAffected();
    } while (rows < ROWS);

    conn.commitTransaction();
    sg.dismiss();
}

void timeTestBulkInsert(RWDBDatabase& db, RWDBConnection& conn)
{
    RWDBTable table = db.table(TABLENAME);

    RWClockTimer timer;
    testBulkInsert(table, conn);
    table.deleter().execute(conn);
    timer.start();
    testBulkInsert(table, conn);
    timer.stop();
    table.deleter().execute(conn);
    cout << "test bulk insert with "
         << ROWS << " rows and "
         << "1000 rows insert at a time took "
         << timer.elapsedMilliseconds() << "ms" << endl;
}

void testInsert(RWDBDatabase& db, RWDBConnection& conn)
{
    int value = 1;
    RWDBTable table = db.table(TABLENAME);
    RWDBInserter inserter = table.inserter();

    conn.beginTransaction();
    RWScopeGuard sg = rwtMakeScopeGuardM(conn,

(RWDBStatus(RWDBConnection::*)(void))&RWDBConnection::rollbackTransaction);

    size_t rows = 0;
    while (rows < ROWS) {
        inserter << value;
        RWDBResult res = inserter.execute(conn);
        rows += res.rowsAffected();
    }

    conn.commitTransaction();
    sg.dismiss();
}

void timeTestInsert(RWDBDatabase& db, RWDBConnection& conn)
{
    RWClockTimer timer;
    testInsert(db, conn);
    db.table(TABLENAME).deleter().execute(conn);
    timer.start();
    testInsert(db, conn);
    timer.stop();
```

```
    db.table(TABLENAME).deleter().execute(conn);
    cout << "test insert with " << ROWS << " rows took "
         << timer.elapsedMilliseconds() << "ms" << endl;
}

void tearDown(RWDBDatabase& db, RWDBConnection& conn)
{
    RWDBTable table = db.table(TABLENAME);
    table.drop(conn);
}

int main()
{
    RWDBManager::setErrorHandler(&outputStatus);

    RWDBDatabase db = RWDBManager::database(RWDB_SERVERTYPE,
                                            RWDB_SERVERNAME,
                                            RWDB_USER,
                                            RWDB_PASSWORD,
                                            RWDB_DATABASE,
                                            RWDB_PSTRING);

    RWDBConnection conn = db.connection();

    setup(db, conn);

    timeTestInsert(db, conn);
    timeTestBulkInsert(db, conn);

    tearDown(db, conn);

    return 0;
}
```