

Tntdb

Author: Tommi Mäkitalo

Introduction.....	1
Connecting.....	1
Execute query.....	2
Selecting data.....	2
Prepared statements.....	3
Working with cursors.....	4
Transactions.....	4
Connectionpool.....	5
Statementcache.....	5

Introduction

Tntdb is a library for simple databaseaccess. There are 2 layers for access – a databaseindependent layer and a databasedriver.

The databaseindependent layer offers easy to use methods for working with the database and also greatly simplifies resource-management. The classes hold reference-counted pointers to the actual implementation. They are copyable and assignable. The user can use the classes just like simple values. The resources they reference are freed, when the last reference is deleted. This happens normally just by leaving the scope. There is normally no reason to instantiate them dynamically on the heap.

The driver-layer contains the actual implementation, which does the work. These classes are database-dependent. The user normally doesn't need to deal with this.

Connecting

A connection is represented by the class `tntdb::Connection`. Tntdb offers a simple function, which connects to the database: `tntdb::connect`. This expects a parameter of type `std::string`, which is the database-url. The database-url consists of the drivename and a databasedependent part divided by a colon.

Example:

```
#include <tntdb/connection.h>
#include <tntdb/connect.h>
int main(int argc, char* argv)
{
    tntdb::Connection conn = tntdb::connect(„sqlite:mydb.db“);
}
```

The example above loads the sqlite-driver-library and opens a connection to the databasefile „mydb.db“. At the end of the program the class `tntdb::Connection` goes out of scope, which closes the connection automatically.

When the database could not be opened an exception is thrown. In the above example it is unhandled, which makes the program to abort. This is not so nice, so we add exception handling in the second example:

```
#include <tntdb/connection.h>
#include <tntdb/connect.h>
int main(int argc, char* argv)
{
    try
    {
        tntdb::Connection conn = tntdb::connect („postgresql:dbname=db“);
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }
}
```

This is a complete example, which just checks, if the database is accessible.

Execute query

To execute a query without selecting data *tntdb::Connection* has a method *execute*. It expects a *std::string* with a sql-statement, which does not return data. It returns the number of affected rows.

Example:

```
#include <tntdb/connection.h>
#include <tntdb/connect.h>
int main(int argc, char* argv)
{
    try
    {
        tntdb::Connection conn = tntdb::connect („postgresql:dbname=db“);
        conn.execute(
            „create table t1(coll int not null primary key,“
            „                col2 int not null)“);
        conn.execute(„insert into t1 values(1, 5)“);
        unsigned n = conn.execute(„update t1 set coll = coll + 1“);
        std::cout << n << „ rows updated“ << std::endl;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }
}
```

Selecting data

A database is not just for storing data, but it also needs to return the data. *Tntdb* offers several ways to read the data from the database. The most general is the method *tntdb::Connection::select()*, which expects a query and returns an object of class *tntdb::Result*.

tntdb::Result is a collection of rows. Rows are represented by the class *tntdb::Row* and these rows are also collections of type *tntdb::Value*. Both collections (*Result* and *Row*) can be accessed with an iterator or through an index. The *Value*-class offers methods for returning the data in different types. *Tntdb* does not tell, which type the column is. *Value* just does its best to convert the data to the requested type. The User has to know, which data the column holds.

Often there are statements, which return exactly one row or only a single value. For convenience *tntdb::Connection* offers the methods *selectRow* and *selectValue*. The former returns the first row of a query and the latter the first value of the first row. Both throw an exception of type *tntdb::NotFound*, if the query returns no rows at all.

A *tntdb::Value* explicit get-methods and implicit converter-operators to access the value.

Example:

```
#include <tntdb/result.h>
#include <tntdb/row.h>
#include <tntdb/value.h>

void someFunc(tntdb::Connection conn)
{
    tntdb::Result result = conn.select("select col1, col2 from table");
    for (tntdb::Result::const_iterator it = result.begin();
        it != result.end(); ++it)
    {
        tntdb::Row row = *it;
        std::cout << "col1=" << row[0] << "\tcol2=" << row[2]
            << std::endl;
    }
}

std::pair<unsigned, unsigned> getMinMax(tntdb::Connection conn)
{
    tntdb::Row r = conn.selectRow(
        "select min(col1), max(col1) from table");
    // we use implicit converters-operators here:
    return std::pair<unsigned, unsigned>(r[0], r[1]);
}

void someOtherFunc(tntdb::Connection conn)
{
    tntdb::Value v = conn.selectValue("select count(*) from table");
    std::cout << "The table 'table' has " << v.getUnsigned()
        << " rows" << std::endl;
}
```

Prepared statements

Most of the time the user needs to parameterize the queries.

Because the query has the type *std::string* they can just be stringed together e.g. with *std::ostringstream*. But this is not recommended normally. The disadvantage is, that the user has to deal with special characters to avoid misinterpretation of data.

Prepared statements solve this by parsing the statement and getting the parameters separately. This also offers sometimes significant performance-advantages, because the user can execute the same statement multiple times with different parameters. The parsing can be done either at the client-side or at the server-side. Tntdb let the driver decide, if the database can parse the query and which placeholders the database needs.

To create a prepared statement *tntdb::Connection* has a method *prepare*, which takes a query as a *std::string* and returns an object of type *tntdb::Statement*. The query can contain parameters. Parameters are named tokens in the query prefixed with a colon. A token can occur multiple times in a query. The *Statement*-class has setter-methods to pass parameter-values with different types.

tntdb::Statement offers the same methods for database access as *tntdb::Connection*: *execute*, *select*, *selectRow* and *selectValue*. They work exactly like the methods in *tntdb::Connection*.

Example:

```
#include <tntdb/statement.h>

void insData(tntdb::Connection conn)
{
    tntdb::Statement st = conn.prepare(
        „insert into table values (:v1, :v2)“);

    st.setInt(„v1“, 1) // the setters return *this, so they can be
                      // chained easily
    .setString(„v2“, „hi“)
    .execute();

    st.setInt(„v1“, 2)
    .setString(„v2“, „world“)
    .execute();
}
```

Working with cursors

Connections and prepared statements offer the method *select()*, which fetches the result and offers random-access to the data. Databases has often more data, than would fit into the memory of the program. To deal with this, the innovators of databases has created cursors. They are like pointers to a window in a resultset, but without holding (and transferring) all data in memory. Tntdb offers this functionality with *const_iterators* in prepared statements. The class *std::Statement::const_iterator* represents a database-cursor. It is a forward-only-iterator, which returns objects of type *tntdb::Row*, when dereferenced.

The *begin*-method of *tntdb::Statement* starts a new iteration of a cursor.

Example

```
#include <tntdb/statement.h>

void printData(tntdb::Connection conn)
{
    tntdb::Statement st = conn.prepare(„select col1, col2 from table“);
    for (tntdb::Statement::const_iterator cur = st.begin();
        cur != st.end(); ++cur)
    {
        tntdb::Row row = *cur;
        std::cout << „col1=“ << row[0].getString() << „ col2=“
                  << row[1].getString() << std::endl;
    }
}
```

In the above example the memory-consumption is low even when the table has millions of rows. When the data would have been fetched with a *tntdb::Result* all rows has to fit into the main-memory.

In contrast to many databases Tntdb allows multiple concurrent iterations for a statement. This is done by allocating a second cursor, when needed. Normally the cursor-handle is reused, when a Statement is iterated multiple times.

The *begin*-method of *tntdb::Statement* has a optional parameter *fetchsize* of type *unsigned*, which is passed to the implementation. The implementation may use it as a hint, how many rows it should fetch at once. The default value is 100.

Transactions

A database wouldn't be a database, if it does not offer transactions. *tntdb::Connection* has 3 methods to deal with it: *beginTransaction*, *commitTransaction* and *rollbackTransaction*. But this is not the recommended way to deal with it. *Tntdb* has more to offer: *tntdb::Transaction*. This class monitors the state of a transaction and closes the transaction automatically, when needed. This offers exception-safety without the danger of open transactions.

tntdb::Transaction are instantiated (just like all *tntdb*-user-classes) as local variables. The constructor starts a transaction and the destructor rolls the transaction back, if the transaction is not explicitly committed. This guarantees, that the transaction is never left open (except when the rollback fails, but this normally happens only, when the connection is broken anyway and there is no way to do any harm to the database any more).

Example:

```
#include <tntdb/transaction.h>

void doSomeModifications(tntdb::Connection conn)
{
    tntdb::Transaction trans(conn);
    // do some modifications in the database here:
    conn.execute(...);
    conn.prepare(...).setString("coll", value).execute();
    trans.commit();

} // no explicit rollback is needed. In case of an exception, the
  // transaction is rolled back automatically here
```

Connectionpool

In a long-running program it is often desirable not to connect and disconnect for every access. One solution is to keep a connection open somewhere and use it as needed. In a multi threaded application the user has to make sure, that there is only one thread at a time accessing the database through a single connection.

To solve this, *Tntdb* offers an automatic connectionpool. When the call to *tntdb::connect* is replaced with *tntdb::connectCached* a special connection is returned. This connection works just like the normal connection (it is the same class), but when destroyed, it does not close the connection, but puts the connection to a free pool. When *tntdb::connectCached* is called again with the same parameter, the connection is reused. When the old connection is still in use, *connectCached* just creates a new one.

Example:

```
std::string url = "mysql:db=mydb;host=192.168.0.1";
tntdb::Connection conn;
conn = tntdb::connectCached(url); // connects to the db
conn = tntdb::Connection();      // puts the connection back to the pool
conn = tntdb::connectCached(url); // fetches the same connection
                                   // (if not already fetched by another thread)
tntdb::dropCached(); // closes all free connections, but not ours,
                     // because it is held by 'conn'
conn = tntdb::Connection();
tntdb::dropCached(); // closes the connection, because we released it
```

Statementcache

As told previously statement-reuse improves performance quite heavily. It is advisable to try to use

prepared statements where possible. In the case of a connectionpool it is quite difficult to maintain prepared statements, because they are specific to the connection.

Tntdb helps here by putting a statementcache into the connection-class. When calls to *tntdb::Connection::prepare* is replaced with *tntdb::Connection::prepareCached*, tntdb looks into the connection, if the same statement is already prepared earlier and returns this when needed and calls *prepare* and fills the statement-cache with this new statement otherwise.