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 database access. There are 2 layers for access – a database independent layer and a database driver.

The database indipendent 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 driver name and a database dependent 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 a exception is thrown. In the above example it is unhandled, which makes the program to abort. This is not so nice, so we add exceptionhandling 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(col1 int not null primary key,“
      „                col2 int not null)“);
    conn.execute(„insert into t1 values(1, 5)“);
    unsigned n = conn.execute(„update t1 set col1 = col1 + 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 need to fetch 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 a 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 a iterator or through a 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 a exception of type *tntdb::NotFound*, if the query returns no rows at all.

A *tntdb::Value* has a get-template-method to read the actual value into a variable.

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;
    int a;
    std::string b;
    row[0].get(a);  // read column 0 into variable a
    row[1].get(b);  // read column 1 into variable b
    std::cout << „col1=“ << a << „\tcol2=“ << b
      << std::endl;
  }
}

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. The disadvantage is, that the user has to deal with special characters to avoid misinterpretation of data and especially avoid sql injection .

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 serve-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 a 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 databaseaccess as *tntdb::Connect*: *execute*, *select*, *selectRow* and *selectValue*. They work exactly like the methods in *tntdb::Connect*.

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();
}
```

You may also omit the explicit mention of the data type. There is a template method *tntdb::Statement::set*, which determines the right type of the column from the type of the passed parameter. If you later decide to change the type of a variable, you don't need to replace the method used to pass the value to the database. Here is the same example as above using this template method:

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

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

  st.set(„v1“, 1)
    .set(„v2“, „hi“)
    .execute();

  st.set(„v1“, 2)
    .set(„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::string col1;
    std::string col2;
    row[0].get(col1);
    row[1].get(col2);
    std::cout << „col1=“ << col1 << „ col2=
              << col2 << 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.

The *begin*-method of *tntdb::Statement* has a optional parameter *fetchsize* of type *unsigned*, which is passed to the driver. It 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 explicitely 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(„...“).set(„col1“, value).execute();
  trans.commit();

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

Using dates and times

Dates and times are a little special when it comes to databases. There is no standard syntax for

specifying dates and times. Therefore tntdb offers 3 helper classes. *tntdb::Date* can hold a date, *tntdb::Time* a time and *tntdb::Datetime* both. They can be used just like built in types in tntdb. You can set host variables and retrieve the value from results returned from the database.

The classes are simple helper classes which do not have any range checks or other knowledge of the nature of the data. The constructor of *tntdb::Date* takes 3 parameters: the year, the month and the day. The constructor takes 3 or 4 parameters: the hour, the minute, the second and a optional millisecond. *Tntdb::Datetime* is created using 6 or 7 parameters. First the 3 parameters of the date and then the 3 or 4 for the time.

There are also 2 static methods in each of the 3 classes *localtime* and *gmtime*, which create a object with the current date or time.

Lets look for some examples:

```
void dateTimeDemos(tntdb::Connection conn)
{
  // insert dates and times into table:
  tntdb::Statement ins = conn.prepare(
    "insert into mytable (datecolumn, timecolumn, datetimecolumn)"
    "  values(:date, :time, :datetime)");

  ins.set("date", tntdb::Date(2010, 2, 13))
     .set("time", tntdb::Time(23, 22, 30))
     .set("datetime", tntdb::Datetime(2010, 2, 13, 23, 22, 30))
     .execute();

  // insert the current dates and times into the table
  ins.set("date", tntdb::Date::localtime())
     .set("time", tntdb::Time::localtime())
     .set("datetime", tntdb::Datetime::localtime())
     .execute();

  // retrieve dates and times from a table:
  tntdb::Statement sel = conn.prepare(
    "select datecolumn, timecolumn, datetimecolumn"
    "  from mytable");
  for (tntdb::Statement::const_iterator cur = st.begin();
       cur != st.end(); ++cur)
  {
    tntdb::Row row = *cur;
    tntdb::Date mydate;
    tntdb::Time mytime;
    tntdb::Datetime mydatetime;
    row[0].get(mydate);
    row[1].get(mytime);
    row[2].get(mydatetime);
    // now we have the data from the table in our variables
    std::cout << "date=" << mydate.getIso() << "\n"
                 "time=" << mytime.getIso() << "\n"
                 "datetime=" << mydatetime.getIso() << std::endl;
  }
```

Using own types

Tntdb offers also support for custom types. The setter and getter methods in tntdb are actually templates, which uses 2 operators to actually map types to the methods. Lets design a own type, which we would like to store in a database field. To make the example simple, we use a struct with public member variables. Normally you should always make the member variables private or

protected and offer setter and getter methods:

```
struct Myclass
{
  int a;
  int b;
  Myclass() { } // default constructor is mostly good to have
  Myclass(int a_, int b_)  // ctor to initialize both members
    : a(a_), b(b_) { }
};
```

We would like to write the class as a pair of numbers separated with ':' into a database field. So we define a operator, which formats the structure properly:

```
void operator<< (tntdb::Hostvar& hv, const Myclass& myclass)
{
  std::ostringstream s;
  s << myclass.a << ':' << myclass.b;
  hv.set(s.str());
}
```

The operator creates a std::string with the content of the class and passes it to this special helper class *tntdb::Hostvar*, which is defined in the header tntdb/statement.h. This is the only case, where you will explicitly use the class.

To retrieve data from the database we need another operator:

```
bool operator>> (const tntdb::Value& value, Myclass& myclass)
{
  if (value.isNull())
    return false;
  std::string str;
  value.get(str);
  std::istringstream in(str);
  char ch;
  in >> myclass.a
     >> ch       // skip the ':'
     >> myclass.b;
  return true;
}
```

That's all. Now we can use the class just like built ins:

```
void customTypeDemo(tntdb::Connection conn)
{
  // insert myclass into table:
  tntdb::Statement ins = conn.prepare(
    "insert into mytable (mycolumn)"
    "  values(:mycolumn)");

  ins.set("mycolumn", Myclass(17, 45))
     .execute();

  // retrieve myclass from a table:
  tntdb::Statement sel = conn.prepare(
    "select mycolumn from mytable");
  for (tntdb::Statement::const_iterator cur = st.begin();
       cur != st.end(); ++cur)
  {
    tntdb::Row row = *cur;
    Myclass myclass;
    row[0].get(myclass);
    // now we have the data from the table in our variables
    std::cout << "a=" << myclass.a << "\t"
                 "b=" << myclass.b << std::endl;
  }
```

Using serial columns

If tables are designed, you have to create a primary key to identify uniquely a specific row in the table. Sometimes it is enough to use just a unique number. Databases has support for a automatically generated primary key. It actually depends on the database, how to create such columns. Since tntdb has no support for ddl statements (create table …), there is actually no need to abstract the creation of these columns. But in applications it is often needed to know, which number was given to the last inserted row. This is also depends on the actual database software used.

Tntdb helps retrieving that last insert id. There is a method "*long tntdb::Connection::lastInsertId(const std::string& name)*"

There is one problem. Some databases (postgresql, oracle) use named sequences and a single table may have multiple of them. So to fetch the actual id, we need to tell the driver, which sequence we want to know. The sequence is identified by name. Other databases (mysql, sqlite) do not have a identifier. They will simply ignore the passed name.

Here is a example. We assume a table foo with 2 columns: id and name. The id is the primary auto incremented key. In postgresql or oracle the sequence is created with the name "foo_id_seq":

```
void insertData(tntdb::Connection conn)
{
  tntdb::Statement ins = conn.prepare(
    "insert into foo (name)"
    "  values(:name)");

  ins.set("name", "some name")
     .execute();

  std::cout << "id: " << conn.lastInsertId("foo_id_seq") << std::endl;
}
```

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 is 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 offerers a 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 hold 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.