N3612: Desiderata of a C++11 Database Interface

Thomas Neumann

Technische Univeristät München
neumann@in.tum.de

2013-03-15

With the recent papers N3415 and N3458 two proposals have been made for a standard database interface in the C++ standard, based upon C++11 features. In addition, a wide range of C++98 database interfaces already exist. Before designing just another interface, we therefore discuss the desirable properties or requirements of a database interface. In the following we classify these requirements in three rough groups, from high-level to low-level. It might not be feasible to fulfill all of these in one unified interface, therefore the group order also roughly implies importance. Afterwards we give a rough classification of existing interfaces according to the desired properties.

Note that while this text contains examples to illustrate certain trade-offs, these examples should not be interpreted as new proposals. They are included for illustrative purposes only, an actual interface might be different.

1 High-Level Requirements

Before describing technical aspects, we briefly discuss the high-level requirements of a database interface, i.e., the kind of functionality it should offer. These are more stylistic requirements, but they describe what a user expects from a database interface.

1.1 A Clean, Concise Interface Query Interface

This is the foremost requirement for a database interface. Access to the database must be easy and simple, and should require as little boilerplate code as possible. As a non-normative example, the following code is concise and simple

```
auto x = db.queryValue<int>("select max(x) from table");
```

while the corresponding ODBC code fragment is not (and error handling would easily double the size of the code)

```
SQLHandle st;
SQLAllocHandle(SQL_HANDLE_STMT,db,&st);
SQLINTEGER x;
SQLBindCol(st,1,SQL_C_SLONG,&x,sizeof(x),nullptr);
SQLExecDirect(st,"select max(x) from table",24);
SQLFetch(st);
```

Ideally, the interface is so simple that the boundary between C++ code and database access is blurred. For example, accessing the database should be (nearly) as simple as iterating over a standard container. And commonly used features should be particularly simple. Queries, i.e., retrieving specific data items from the database, are extremely common, and should be made as simple as possible. Rarely used functionality like defining a new stored procedure must be supported too, of course, but might not warrant specific syntactic sugar.

1.2 Support for Queries, Transactions, and Statements

As indicated in the section title, the database interface has to offer support for queries, transactions, and general statements. This list might sound a bit arbitrary, but there are actually good reasons to consider these three: First, queries, i.e., the programmatic access to data stored within the database, is the core aspect of database usage. Queries are used extremely frequently, and the query result are used by other parts of the application. As such they warrant extra effort to make queries as painless as possible.

The second aspect, transactions, are one of the key characteristics of databases vs. flat files. In a transaction context, database operations are executed atomically, on consistent data, in isolation from other transactions, and with a durable result (the so-called ACID properties). This affects both the database and the C++ program itself, and should therefore be visible in the application. For example using a RAII idiom for transactions seems to be a very good idea

```
{
  transaction trans(db); // begin of atomic transaction block
  db.query(" ... ");
  db.query(" ... ");
  trans.commit(); // end of atomic transaction block, destructor rolls back if no commit
}
```

as now the structure of database transaction is cleanly visible in the C++ code.

Finally, database systems will always support some special purpose functionality that the database interface cannot anticipate. Examples include backup functionality, federation, etc. The database usually exposes these as general statements with side effects. This functionality must be supported, but it will be used much more rarely, therefore it can be handled with a more generic interface than queries.

In some cases statements can behave similar to queries in that they can return one (or even more) results. This is the case for stored procedures, i.e., user code that is embedded inside the database and potentially produces one or more query results. These constructs are used more rarely, but an database interface should support it.

Ideally the result interface would be the same for both regular queries and query result from statements, as then the user can treat both similar, for example like this:

```
auto st=db.statement("execute myprocedure(1,2,2)");
for (auto name:st.next_result<string>())
    cout << name << endl;
for (auto id:st.next_result<int>())
    cout << id << endl;
```

Note that it might be useful to expose more than these three mechanisms to the users, for example stored procedures, but these three are the required minimum.

1.3 Seamless Integration in C++

Just as it should be simple to access the database, the database access could should fit in naturally into the surrounding C++ code. For example it it would seem plausible to treat database queries similar to functions or even lambdas, in the sense that they are "callable" constructs, potentially with parameters, that produce a collection of result tuples, as shown in the following example

```
auto query=db.prepare<int>("select name from table where id = ?");
for (auto name:query<string>(5))
    cout << name << endl;
```

Historically that has not been the case, largely due to C++98 restrictions. Many database systems used a very verbose syntax to couple C++ and the database, which leads to somewhat cumbersome code, as illustrated by the QtSql example below

```
QSqlQuery query;
query.prepare("select  name from table where id = ?");
query.addBindValue(5);
query.exec();
while (query.next())
    cout << query.value(0).toString().toStdString() << endl;
```

Other database libraries invented their own syntax constructs to express queries, as illustrated by the following libsoci example

```
string  name;
statement st = (sql.prepare << "select name from table where id = :val",use(5),into(i));
st.execute();
while (st.fetch())
     cout << name << endl;
```

The libsoci code is quite readable, but the usage of overloaded comma operators is a bit unusual. In addition, libsoci plays games with the lifetime of temporary objects to detect the end of the statement (i.e., the last comma-separated value). These brittle tricks would not be necessary in C++11 when using variadic template functions.

2 Technical Requirements

After these high-level requirements we now look at the technical requirements for a database interface. Note that these are not necessarily hard requirements, a standard

interface might compromise in some of these, but all requirements mentioned here are at least highly desirable.

2.1 Generic Support for Queries/Statements and Data Types

A fundamental assumption that any standard database interface should make is that the database system is a black box. In particular, the interface itself should not try to interpret the queries or statements in any way. The reason for this is that most database vendor implement some non-standard language extensions, and it would be hopeless to try to cope with all of these in a single standard database interface. Thus, all textual statements have to be passed to the database (or more precisely: to the appropriate database driver) uninterpreted. Note that the driver itself can interpret the query, as the driver is database specific. Thus, parameter types, result types, etc., can be retrieved from the driver after passing the textual query/statement.

In particular, one should not assume any kind of query syntax like SQL. Interpreting the query is purely up to the driver. In fact one should not even assume that the underlying database system is relational. Most "relational" database systems are not purely relational any more anyway. The basic contract between the database interface and the underlying database driver/system should be:

- a *query* is an n-ary function that produces a collection of tuples as output
- a *statement* is an n-ary function that changes the state of the database and produces a count as output (traditionally the number of affected rows, but that is implementation defined)

An actual interface might wish to expose more constructs, i.e., for stored procedures, but these are the basic requirements. And no interpretation of the semantics beyond this contract should be done to allow for maximum flexibility.

Ideally this agnosticism towards database functionality contains the type system, too. Many database systems offer proprietary data types for special purposes, and it should be possible to use them in the C++ binding. Of course this requires that the underlying database driver supports the data type, and in a practical implementation that will require template functions to cope with the different types. But unless there are very good reasons to forbid other data types, a database interface should be open to support non-standard data types provided by a specific database driver.

Note that this requirement implicitly rules out approaches like LINQ that embed the query as compiler-interpreted constructs. Interpreting the query at compile time is attractive, as it can catch errors early. But adding similar functionality to C++ would require significant language changes, which is unrealistic. Furthermore, LINQ does make assumptions about the database, and thus cannot handle arbitrary query constructs. As a compromise it would be possible to write a database-specific compiler plugin that checks the textual queries for errors at compile time. But this would be implementation defined and beyond the scope of the standard.

2.2 Support for SQL-92

This requirement somewhat contradicts the previous requirement, namely that the database interface should be database agnostic. But even though we want to support arbitrary queries and types, most real-world database system implement a super-set of SQL-92. As such, it makes sense to ensure that 1) all of the SQL-92 functionality can be mapped conveniently to the proposed interface, and that 2) standard type mappings exists for all SQL-92 data types. Otherwise, different database vendors would implement the mappings differently, which would lead to unnecessary fragmentation.

In a sense the database world is fragmented anyway, and realistically this fragmentation cannot be completely healed by a database interface library, but one should try to minimize the fragmentation. And SQL-92 is an extremely well supported basis that should be handled well. Newer language revisions exist, the current SQL standard is ISO/IEC 9075:2011, but de-facto SQL-92 is only standard revision that is really universally supported.

2.3 Support for both Static and Dynamic Usage

The main purpose of the database interface is to simplify the interaction with the database. As such, it should be easy to couple queries (and in particular, query results) with code that interprets the result. We consider this the *static* use case, an example is shown below.

```
int value;
for (auto row:db.query(querystring,123).into(value))
    cerr << value << endl;
```

Note that the query text itself does not have to be static, it can be an arbitrary string. But both the type of the query parameters (if any) and the type of the result is known at compile time. This is probably the most common use case for a database interface, as application programs will nearly always make assumptions about the result.

But sometimes even the structure of the query result is not known at compile time, for example when implementing a web interface that supports arbitrary user queries. We call this the *dynamic* use case, and it has to be supported, too. An example is shown below

```
auto query=db.prepare_query(queryString);
for (auto& p:userParameters)
    query.bind(p);
for (auto row:query()) {
    for (auto index=0,limit=row.column_count();index!=limit;++index)
        cerr << row.to_string(index) << " ";
    cerr << endl;
}
```

Such a fully dynamic use case is probably more rare, as without any prior knowledge about the result the application cannot do much except pass it to the user as it is. But still there is a need for such functionality.

2.4 Efficiency

Finally, the database interface has to allow for an efficient implementation. Some applications access the database thousands of times per second, and the interface must cope with such transaction rates (given a suitable driver implementation, of course). This effectively implies two consequences: 1) the interface must allow for "prepared queries", i.e., queries that are compiled once by the database and that can the be executed repeatedly without recompilation by just changing the query parameters (if any). Query compilation is very expensive, and can easily take many milliseconds, which is a problem for high query rates. And 2) the interface, must allow for implementations that forgo type conversions. In the static use case all types are known at compile time, and can be passed to the database if needed. Therefore, the database can often produce data in exactly the right format. This is mainly relevant for embedded high-performance databases, network traffic or other communication costs will dominate conversion costs. But still it is a desirable goal.

3 Implementation Requirements

Besides these more high-level requirements, there are also some implementation aspects that have to be considered. These are more like small details, but they can have quite some impact on the way the database interface can be used.

3.1 Exception Safety

The database interface must be able to cope with exceptions, and it should reports its own errors using exceptions, too. While this might sound like an obvious requirement, it is not trivial to implement, as there exists a database state this is coupled with but separate from the application state. And exceptions might occur at any point in time, for example while currently processing the result of an unfinished query. But while exception safety requires some effort from the interface developer, it greatly simplifies the life of the application developer. The same for using exceptions instead of return code. In some of the existing database interfaces error handling is very complicated and bloats the application code, even though errors are rare in practice and therefore the costs of exceptions are negligible.

3.2 Strong Separation between Compilation and Execution

Most database systems have a clear separation between query compilation, i.e., preparing the query for execution, and actually executing the query. Many applications have a fixed core of queries that are executed very frequently, and which are therefore prepared once and then executed multiple times. This separation should also be possible inside the application program. Some of the existing database interfaces offer only an incomplete separation, e.g., by requiring dedicated result variables already at prepare time. This is unfortunate from a usability point of view.

3.3 Minimize Implementation Exposure

Database interfaces are often implemented using database cursors or similar techniques. These implementation details should not be exposed the user, unless there are very compelling reasons to do so. For example the interface should not encourage (or even allow for) accessing a query result in arbitrary order, as this restricts possible implementations. Most users do not need this functionality, and buffering can easily be implemented in application code if needed. As the interface is supposed to be database agnostic, it should not favor one implementation over the other. And for most user code it makes absolutely not difference how a query is implemented internally. Most of the time a user wants simple result comprehension like this

```
int value;
for (auto row:db.query("select value from foo").into(value))
    cerr << value << endl;
```

and none of the internals should be visible. If arbitrary access is required, buffering the *value* entries in the example above would be trivial for the application.

4 Use Cases

In order to get an idea about the practical implications of different library designs, it is insightful to consider use cases. Some use cases are already included in N3415: A Database Access Library by Bill Seymour. As a condensed example, it might be interesting to consider the small pseudo-code fragment below:

```
create table users(id integer not null,name varchar(40) not null);
create table groups(id integer not null,name varchar(40) not null);
create table assignments(user integer not null,group intege not null);
create procedure dropprivileges(user integer not null) ... // database specific code

// parameterized transaction, should be C++ code
begin transaction
select u.name,u.id from users u,assignments a,groups g
where u.id=a.user and a.group=g.id and g.name=? {
    // non-standard syntax, loop over statement result
    execute dropprivileges(a.id) -> privilege {
        print u.name, ' lost ', priviledge
    }
}
commit transaction
```

It assumes tables to manage users and groups, and a stored procedure that drop the priviledges of a given user and reports the dropped priviledges as a query result. Within a database transaction the user code finds all users from a given group, drops their priviledges, and then reports the result back. Note that the pseudo-code should be C++ code in reality, as this is the application logic, but we used pseudo-code here to order to avoid biasing in favor of a certain proposal.

This small code fragments already highlights quite a few concepts (transactions, queries, parameterized queries, statements, statement results), and should cover the

most common database usage scenarios. Therfore it should be a reasonable use case to evaluate existing approaches, and we that future database proposal will consider this example (or something similar) to illustrate their strength and weaknesses.

5 Some Preliminary Findings

In the following we include a brief overview over existing libraries. We mention some of the limitations (by referencing the section numbers from above), and include a short code fragments that prints the name of a certain user. This is only a preliminary survey intended to get an idea of existing systems. Promising candidates should be evaluated more thoroughly considering the use case shown above.

Note that most interfaces violate requirement 2.1, i.e., lack support for arbitrary data types. Of course that could imply that the requirement is too difficult to implement. But some of the interfaces support arbitrary data types. We should therefore not give up on it too early.

ODBC

ODBC: `http://msdn.microsoft.com/en-us/library/ms714562(v=vs.85)`
unixODBC: `http://www.unixodbc.org/`
iODBC: `http://www.iodbc.org/`

Limitations: 1.1, 1.3, 2.4, 3.1, 3.3

ODBC is a standard database interface for C, released in 1992. Obviously the integration in C++ is quite poor, the code is very verbose and error handling is cumbersome. But ODBC itself is well supported by vendors.

```
SQLHandle st;
SQLRETURN res=SQLAllocHandle(SQL_HANDLE_STMT,db,&st);
if ((res!=SQL_SUCCESS)&&(res!=SQL_SUCCESS_WITH_INFO))
    return false;
res=SQLPrepare(st,"select name from users where id=?",34);
if ((res!=SQL_SUCCESS)&&(res!=SQL_SUCCESS_WITH_INFO)) {
    SQLFreeHandle(SQL_HANDLE_STMT,st);
    return false;
}
SQLINTEGER val=1234;
SQLLen len=sizeof(val);
res=SQLBindParameter(st,1,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_INTEGER,len,0,&val,len,&len);
if ((res!=SQL_SUCCESS)&&(res!=SQL_SUCCESS_WITH_INFO)) {
    SQLFreeHandle(SQL_HANDLE_STMT,st);
    return false;
}
res=SQLExecute(st);
if ((res!=SQL_SUCCESS)&&(res!=SQL_SUCCESS_WITH_INFO)) {
    SQLFreeHandle(SQL_HANDLE_STMT,st);
    return false;
```

```
}
while (true) {
    res=SQLFetch(st);
    if (res==SQL_NO_DATA)
        break;
    if ((res!=SQL_SUCCESS)&&(res!=SQL_SUCCESS_WITH_INFO)) {
        SQLFreeHandle(SQL_HANDLE_STMT,st);
        return false;
    }
    char buffer[1024]; SQLLen len;
    res=SQLGetData(st,1,SQL_C_CHAR,buffer,sizeof(buffer)-1,&len);
    if ((res!=SQL_SUCCESS)&&(res!=SQL_SUCCESS_WITH_INFO)) {
        SQLFreeHandle(SQL_HANDLE_STMT,st);
        return false;
    }
    buffer[len]=0;
    cout << buffer << endl;
}
SQLFreeHandle(SQL_HANDLE_STMT,st);
```

JDBC

`http://www.oracle.com/technetwork/java/javase/jdbc/index.html`

Limitations: 1.1, 1.3, 2.4, 3.1, 3.3

JDBC is a Java binding, and as such not so relevant for a C++ library. But JDBC is extremely popular, and therefore included here. Interestingly, JDBC is one of the few database interfaces to support arbitrary data types.

```
PreparedStatement st;
ResultSet rs;
try {
    st=db.prepareStatement("select name from users where id=?");
    st.setInt(1,1234);
    rs=st.executeQuery();
    while (rs.next())
        System.out.println(rs.getString(1));
} finally {
    if (rs!=null) try { rs.close(); } catch (SQLException ignore) {}
    if (st!=null) try { st.close(); } catch (SQLException ignore) {}
}
```

libpqxx

`http://pqxx.org/development/libpqxx/`

Limitations: 1.1, 1.3, 2.1, 2.4, 3.3

Libpqxx is a C++ binding for PostgreSQL. While database specific, and thus not generic, it offers a reasonable C++98 interface.

```
db.prepare("myquery","select name from user where id=$1");
pqxx::prepare::invocation st=trans.prepared("myquery");
pqxx::result result=st(1234).exec();
for (pqxx::result::const_iterator iter=result.begin(), limit=result.end(); iter!=limit;++iter) {
    pqxx::tuple tuple=*iter;
    cout << tuple[0].to<std::string>() << endl;
}
```

C interface for SQLite

`http://www.sqlite.org/capi3ref.html`

Limitations: 1.1, 1.3, 2.1, 2.4, 3.1, 3.3

SQLite offer a C/C++ binding that can be used for database access. Being C-based, it is quite cumbersome to use compared to C++ bindings.

```
sqlite3_stmt* st;
if (sqlite3_prepare_v2(db,"select name from users where id=?",33,&st,nullptr)!=SQLITE_OK)
    return false;
if (sqlite3_bind_int(st,1,1234)!=SQLITE_OK) {
    sqlite3_finalize (st);
    return false;
}
while (true) {
    int res=sqlite3_step(st);
    if (res==SQLITE_DONE) break;
    if (res!=SQLITE_ROW) {
        sqlite3_finalize (st);
        return false;
    }
    cout << sqlite3_column_text(st,0) << endl;
}
sqlite3_finalize (st);
```

Poco::Data Library

`http://pocoproject.org/docs-1.5.0/category-POCO_Data_Library-index.html`

Limitations: 1.3, 2.1, 2.2, 2.4, 3.2, 3.3

The poco project offers a database library with a clean C++98 interface. It is easy to use, but its SQL-92 support is incomplete regarding data types and the separation between compilation and execution is weak.

```
// Query compilation, all variables must already exist
std::string name; int value;
Statement stmt = ( db << "select name from users where id=?", use(value), into(name) );
// Qury execution
value=1234;
stmt.execute()
RecordSet result(stmt);
```

```
while (result.moveNext())
    cout << name << endl;
```

SOCI

`http://soci.sourceforge.net/`

Limitations: 1.3, 2.4, 3.2, 3.3

SOCI offers a very nice and powerful C++ interface for database access. It makes use of unusual constructs (comma operators, delayed side effects by destructors) to work around C++98 limitations. It is very generic and explicitly aims at supporting aribtrary data types. Like poco, separation between compilation and execution is weak.

```
// Query compilation, all variables must already existing
std::string name; int value;
statement st = (db.prepare << "select name from users where id=:val",use(value),into(name));
// Query execution
value=1234;
st.execute();
while (st.fetch())
    cout << name << endl;
```

OTL

`http://otl.sourceforge.net/home.htm` Limitations: 1.3, 2.1, 2.4, 3.2, 3.3

The OTL library offers a rich C++ library for database access. Although geared towards Oracle support, it can be configured for other databases using preprocessor defines. It is simple to use, but the heavy reliance on preprocessor settings for configuration seems unfortunate.

```
otl_stream st(50," select name from users where id=:f");
st << 1234;
while (!st.eof()) {
    std::string name;
    st >> name;
    cout << name << endl;
}
```

5.1 DTL

`http://dtemplatelib.sourceforge.net/index.htm`

Limitations: 1.1, 1.2, 1.3, 2.1, 2.2, 2.4, 3.2, 3.3

The Database Template Library (DTL) offers STL-style view over database tables, allowing for simple access. It aims for shielding the user from raw SQL, but its native

SQL support is quite limited. This makes usage awkward for more complex application scenarios.

```
// Earlier declarations
struct Row {
    std::string name;
};
template<> class dtl::DefaultBCA<Row> {
public:
    void operator()(BoundIOs &cols, Row &rowbuf) {
        cols["name"]==rowbuf.name;
    }
};
struct Param {
    int value;
};
class BPAParam {
public:
    void operator()(BoundIOs &boundIOs, Param &paramObj) {
        boindIOs[0] << paramObj.int;
    }
};

// Usage
DBView<Row,Param> view("USERS",dtl::DefaultBCA<Row>(),"where id=?",BPAParam());
DBView<Row>::select_iterator read_it = view.begin():
read_it.Params().value=1234;
for ( ; read_it != view.end();  ++read_it)
    cout << (*read_it).name << endl;
```

libodbc++

`http://libodbcxx.sourceforge.net/`

Limitations: 1.1, 1.3, 2.1, 2.4, 3.3

libodbc++ offers a JDBC-style database binding for C++. Like its Java equivalent it is somewhat verbose, but at least error handling is simpler in C++.

```
std::auto_ptr<PreparedStatement> st=db->prepareStatement("select name from users where
      id=?");
st->setInt(1,1234);
std::auto_ptr<ResultSet> rs=st->executeQuery();
while (rs->next())
    cout << rs->getString(0) << endl;
```

QtSQL

`http://qt-project.org/doc/qt-4.8/qtsql.html`

Limitations: 1.1, 1.3, 2.1, 2.4, 3.1, 3.3

The well-known qt library offers an SQL module for database access.

```
SqlQuery st(db);
if (!st.prepare("select  name from users where id=?"))
    return false;
st.addBindValue(1234);
if (!st.exec())
    return false;
while (st.next())
     cout << query.value(0).toString().toStdString() << endl;
```

N3458

`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3458.pdf`

Limitations: Currently only a prototype

The N3458 proposal was made to address all the requirements here. However, currently only a prototype for an experimental database back-end and a partial ODBC back-end exist.

```
prepared_query<int> query=db.prepare_query("select name from users where id=?");
std::string  name;
for (auto row:query(1234).into(name))
    cout << name << endl;
```

Acknowledgements: Thanks to Jens Maurer for his help in preparing this document and many insightful comments.