SQL mit C++

...

Schnittstellen im Vergleich

Volker Aßmann (volker.assmann@gmail.com)

SQL++ ??!!

- Entwickler vs. Datenbank
  - “Impedance Mismatch”
  - Doch lieber einfach Dateien?
  - NoSQL?
- SQL Datenbanken sind
  - Konsistent und transaktionssicher (ACID)
  - Stark typisiert
  - Schnell (relativ)
- Keine SQL Standard Schnittstelle wie JDBC, LINQ ... in C++
  - *Wie spreche ich mit meiner Datenbank ?*

SQL++ ??!!

- Hier geht es nicht um
  - Gutes / schönes / schnelles SQL
  - Spezifische Datenbanken (Beispiele mit SQLite und PostgreSQL)
  - nur am Rande: Performance, Sicherheit
- Auswahl der richtigen API für C++ Datenbankzugriff
  - Projekt: von Sybase ASE zu generischer DB API
  - Bisher: Embedded SQL
  - Evaluation verschiedener Optionen

Auswahl einer API

- Welche Datenbanken werden unterztützt?
- Welche SQL / C++ Features werden unterstützt?
  - Anpassung an Datenbankdialekte? Formatkonvertierungen?
  - Objektorientierung?
  - Multi-threading, Verschlüsselung, Authentifizierung, Performance?
- Wie komplex ist die API?
- Abhängigkeiten? Lizenzen? z.B. Qt, sqlapi etc.

Inhalt

- Beispiel: Notizbuch
- DB spezifische (C) APIs
  - SQLite3
  - libpq (PostgreSQL)
- Einfache Abstraktion: QtSql
- Objekte vs. Records mit Wt::dbo
- SQL DSL per TMP (sqlpp11)

Beispielanwendung: Notizbuch

- Notizbücher: (id, title)
- Notizen: (id, title, content, notebook_id, last_change, reminder)
- Tags: (id, title)
    - “tags_nm”: (tag_id, note_id)

Demo !

Code: https://github.com/volka/talks/

C APIs der Datenbanken

- Spezifisch für DB
- Low Level API - alle anderen APIs bauen auf diese auf!
  - d.h. ist es hilfreich die APIs zu kennen
- für kleine Aufgaben ausreichend, z.B. SQLite als File IO Ersatz
- Zugriff auf “Spezialfeatures” der Datenbanken
- SCHNELL!

Genereller Ablauf

- connect()
- execute(query) -- oder -- prepare(query)
  - für Prepare: bind(parameter) ; execute(prepared)
- Auf DB: plan(query), execute(plan), send_result()
- fetch_all() -- oder -- fetch_next() -> interner Result Puffer
- while (has_next()) bind_result(column#, type, target)
- free(result / query); disconnect()

SQLite3

http://www.sqlite.org

SQLite3

- Embedded DBMS - “Datenbank in einer Datei”
  - mit “:memory:” auch in-memory DB
  - Portabel und fast überall verfügbar
- API
  - “plain C” Bibliothek, enthält komplettes DBMS
  - simples Typsystem (NULL, INTEGER, REAL, TEXT, BLOB)

SQLite3 Minimal

```
sqlite3* conn;
auto result = sqlite3_open(“/tmp/notebook.db”, &conn);
if (result != SQLITE_OK) throw runtime_exception(“open failed”);

sqlite3_stmt *stmt;
string create = “SELECT * FROM notebooks”;
result = sqlite3_prepare_v2(conn, create.c_str(), (int)create.size(), &stmt, nullptr);
if (result != SQLITE_OK) throw runtime_exception(“prepare failed”);

result = sqlite3_step(stmt);
if (result != SQLITE_OK && result != SQLITE_ROW && result != SQLITE_DONE)
      throw runtime_exception(“binding result failed”);
int id = sqlite3_column_int(stmt, 0); // nicht result, value, bind !
char* title = sqlite3_column_text(stmt, 1);

sqlite3_finalize(stmt); // or sqlite3_reset to reuse
sqlite3_close_v2(conn);
```

SQLite3 - Abkürzungen

- sqlite3_exec(conn, sql, callback, self, errmsg) - für einfache “one shot” Queries
  - Callback: int (*callback)(void* self, int columns, char** result_text_ptrs, char** col_names)
- prepare / step / finalize ruft Ergebnisse einzeln ab
- sqlite3_get_table() lädt stattdessen das gesamte Resultset (Achtung: “legacy”)
  - d.h ist bekannt wie viele Results es gibt
  - sqlite3_free_table(result) zum freigeben

SQLite3 - Hinweise

- kein fester Datums / Zeit Typ
  - speichern in TEXT (ISO8601 - YYYY-MM-DD HH:MM:SS.SSS)
  - INT (Unix Timestamp) oder
  - REAL (“Julian day numbers, days sincs noon 24.11.4714 BC !”)
- Escaping: per “printf” Funktionen
  - sqlite3_mprintf, sqlite3_vmprintf (“malloc”), brauchen sqlite3_free
  - sqlite3_snprintf, sqlite3_vsnprintf (vorallokierter Buffer)
  - zusätzliche Formate “%q”, “%Q”, “%w”, “%z” für SQL Escaping

SQLite3

DEMO !

RAII Wrapper,

Einfache Query,

Result Parsing

PostgreSQL: libpq

http://www.postgresql.org

PostgreSQL

- PostgreSQL: “mächtigstes” Open Source DBMS
- Objekt-relationale Datenbank
- Sehr Standardkonform, oder angelehnt an Oracle (besonders PL/PGSQL)
- XML / JSON Erweiterungen
- libpq: “C” API implementiert Client-Server Kommunikation
  - neben JDBC einzige “native” Protokollimplementierung
- Sehr gute Doku !

PostgreSQL: libpq

- PGconn* / PGresult* als zentrale “Objekte”, kein globaler State
- Connect: `PGconn* conn = PQconnectdb(conn_info);`
- Einfache Queries: `PGresult* res = PQexec(“SELECT 1;”);`
- Ausgabe:

```
cout << PQntuples(res) << “ ” << PQnfields(res);

cout << PQgetvalue(res, 0, 0);
```

- Aufräumen: `PQclear(res); PQfinish(conn);`

PostgreSQL: PGconn State

- Achtung: Connections haben “state”
- Aktive Transaktionen / Isolation Level
  - Achtung: keine “nested transactions”
- Prepared Statements / Cursor
- allgemein “Portals” als Sicht in Abfrageresultate
- D.h. Connection per Thread oder vorsichtiges “herumreichen” der Connections

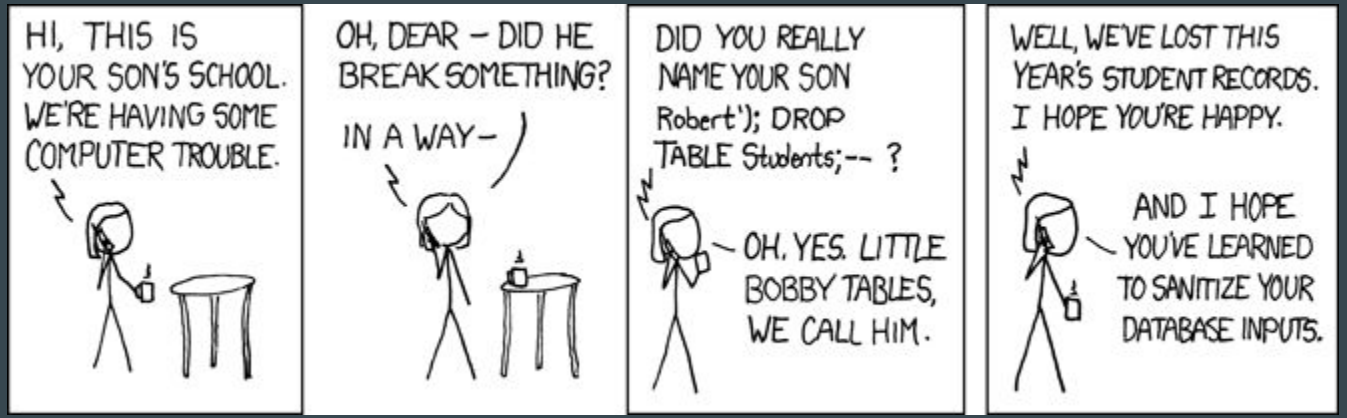
PostgreSQL: Connection Handling

DEMO !

PostgreSQL: Query Parameter

- Einfache Abfrage:

  PQexec(“SELECT * FROM notes WHERE (title like ‘“ + title + “’)”);



https://xkcd.com/327/

PostgreSQL: Query Parameter

- Strings escapen:

  char* PQescapeLiteral(PGconn *conn, const char *str, size_t length);

  size_t PQescapeStringConn(PGconn *conn, char *to, const char *from,
  size_t length, int *error);

  char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);

- Warum conn? Encoding!

PostgreSQL: Prepared Statements

- Aufteilen von “Query Plan” und “Execute”,
- Für “batch” statements, häufig aufgerufene INSERT / UPDATE / DELETE / SELECT mit Parametern
- PQprepare / PQexecPrepared
  - Parameter hier mit “$1”
- DEMO: “newNote()”

PostgreSQL: Asynchrones Interface

- PQexec & Co. warten bis das Resultat vollständig geladen ist
- Asynchrone alternativen:

```
PQsendQuery(); PQsendQueryParams();

PQsendPrepare(); PQsendQueryPrepared();

PQgetResult();
```

PostgreSQL: Performance “tweaks”

- Binary Interface
  - PQexecParams, PQexecPerpared → Text oder Binärformat für Parameter / Resultate
  - Lokale Datenbank: Socket Interface statt TCP (geringere Latenz, höherer Durchsatz, ~ 30%)
- (!!) Aber: größte Performancegewinne durch gutes Schema / Queries (→ Explain)
  - Daten in DB halten / nur Abfragen was man braucht! z.B. durch Stored Procedures / Views
- siehe auch
  - pqxx - C++ Wrapper um libpq
  - libpqtypes - Binary Interace und Typkonvertierung (z.B. Timestamps, Floats usw.)
  - Tools: pgAdmin3, phpPgAdmin, psql

Qt Sql

http://www.qt.io

QtSql: einfache DB Abstraktion

- Eigene “Konnektoren” für viele Datenbanken
  - Low level Query / Result API
  - High level “Table Model” API spezifisch für Qt Widgets
- “Qt-ified” - QStrings, QVariants ... überall
- High Level Interface: QSqlTableModel / QSqlQueryModel

QtSql: unterstützte Datenbanken

QDB2	IBM DB2
QIBASE	Borland InterBase Driver
QMYSQL	MySQL Driver
QOCI	Oracle Call Interface Driver
QODBC	ODBC Driver (includes Microsoft SQL Server)
QPSQL	PostgreSQL Driver
QSQLITE	SQLite version 3 or above
QSQLITE2	SQLite version 2
QTDS	Sybase Adaptive Server

QtSql: Connection Handling

- QSqlDatabase::addDatabase()
  - Mit "type" String der QSqlDriver
  - Default Connection oder "benannt"

```
PGconn *conn = PQconnectdb("dbname=postgres user=volker");
QPSQLDriver *drv =  new QPSQLDriver(conn);
QSqlDatabase db = QSqlDatabase::addDatabase(drv);

// oder besser:

QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL");
db.setHostName("localhost");
db.setDatabaseName("postgres");
db.setUserName("volker");
db.setPassword("*****");
bool ok = db.open();
```

QtSql: Query Interface

- QSqlQuery()
- Direkter Aufruf: `QSqlQuery q("SELECT * FROM notes");`
    - wirklich direkt ausgeführt, Resultat per:

```
if(q.isActive()) {
    while (q.next()) {
        if (q.isValid()) cout << q.value(0).toInt();
} }
```

QtSql: Query Interface - Prepared Statements

- Prepared Statements: `prepare("INSERT INTO notebooks VALUES (?, ?)")`
  - Parameter: "?" oder ":name" - `bindValue(":title", "Einkaufsliste");`
  - Parameter Typ: `QVariant` - QString, Int, Float, QDateTime usw.
  - Ausführung: `exec(); execBatch()`

```
QSqlQuery q;
q.prepare("insert into notebooks values (?, ?)");

QVariantList ints << 1 << 2 << 3 << 4;
q.addBindValue(ints);

QVariantList titles << "Code" << "Arbeit" << "Kochen" << QVariant(QVariant::String);  // <-- NULL VALUE !
q.addBindValue(titles);

if (!q.execBatch())
   qDebug() << q.lastError();
```

QtSql: Fehlerbehandlung

- `class QSqlError`
  - `text(), driverText(), databaseText() -> Errortext der Schichten`
  - `isValid(), nativeErrorCode(), operator==, ErrorType`
- `QSqlDatabase::lastError()` - auf Connection Ebene
- `QSqlQuery::lastError()` - bezogen auf die Query
- Achtung: keine Exceptions !
  - `prepare() / exec() / execBatch()` liefern bool Resultate
  - `isValid() / isActive()`

QtSql: Transaktionen

- `bool QSqlDatabase::transaction()`
  - Startet Transaktion, kein RAII Objekt !
- `QSqlDatabase::commit(), QSqlDatabase::rollback()`
  - Achtung: einige DBs mögen keine “active” Querys - `Query::finish() / Query::clear()`
- Transaktionen unterstützt?
- `QSqlDababase::database()::driver()::hasFeature(QSqlDriver::Transactions)`

QtSql: High Level - QSqlTableModel / QSqlQueryModel

```
void initializeModel(QSqlQueryModel *model)
{
   model->setQuery("select * from person");
   model->setHeaderData(0, Qt::Horizontal, QObject::tr("ID"));
}
QTableView* createView(QSqlQueryModel *model, const QString &title = "")
{
   QTableView *view = new QTableView;
   view->setModel(model);
   view->setWindowTitle(title);
   view->show();
   return view;
}
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   if (!createConnection())    return 1;
   QSqlTableModel model;
   initializeModel(&model);
   QTableView *view1 = createView(&model,
       QObject::tr("Table Model (View 1)"));
   view1->show();
   return app.exec();
}
```

http://doc.qt.io/qt-5/sql-presenting.html

Wt::Dbo

http://www.webtoolkit.eu/wt

Wt::Dbo: Objekt-Relationales Mapping

- Wt = “Web Toolkit” - Wt::Dbo als DB Schnittstelle
  - GPL oder kommerzielle Lizenz!
- Template basiertes Mapping von Klassen auf Objekte
- ORMs vermeiden “impedance Mismatch”, vs. Performance / Generalität
- In der Regel keine direkten SQL Queries, Arbeit mit Objekten
  - aber SQL möglich falls nötig
- Funktionen zum generieren des Schemas

Wt::Dbo: Objekt Mapping Klassen

- Problem: Objekte im Speicher vs. Records in DB
  - Synchronisation per cache / dirty marker !
- Mapping : zentral per Klasse in template Methode “persist”

```
template <class Action> void persist(Action &a) {

// dbo::field(a, id_, “id”); // not used here - Wt::Dbo generates id / version itself !
dbo::field(a, title_, “title”);
dbo::belongsTo(a, notebook, "notebook_");
dbo::hasMany(a, tags, dbo::ManyToMany, “tags_nm”); // same on Tags side
}

dbo::ptr<Notebook> notebook;
dbo::collection<dbo::ptr<Tag>> tags;
```

Wt::Dbo: Objektverwaltung

- `Wt::Dbo::ptr< C >` - Shared Pointer zum Verwalten von DB Objekten
  - `get()` ermglicht “const” Zugriff - “persistente” Objekte (synchron mit DB)
  - `modify()` für Schreibzugriffe - transiente Objekte (geändert oder neu)
  - Aktionen immer in Transaktion, per default `flush()` transienter Objekte bei `commit()`
  - `setFlushMode()` für manuelles flushen

```
dbo::Transaction transaction(session);

auto nb = make_unique<Notebook>();
nb->title("Code");

dbo::ptr<Notebook> nbPtr = session.add(nb.release());

transaction.commit();     // WARNING: ~Transaction() will always rollback() unless explicitly commit()’ed !!!
```

Wt::Dbo: Objektverwaltung

- `Wt::Dbo::Collection< C >` - Collection Klasse meist von `ptr<C>` Objekten
  - Als Query Resultat - Read Only (nur ein begin() erlaubt)
  - Als Many Seite einer ManyToOne Relation - insert() / erase()
  - STL Interface (Iteratoren, begin() / end())
  - Wenn Query Result, dann ist nur eine Iteration erlaubt ! Ggf. Kopie in STL Container

```
void get_notes(dbo::ptr<Notebook>>& notebook)
{
typedef std::vector<dbo::ptr<Note> > Notes;
Notes notes(notebook->notes.begin(), notebook->notes.end()); // copy into STL container, query freed

for (const auto& n: notes) {
std::cerr << "Note: " << n->title() << std::endl;
   }
}
```

Wt::Dbo: Connection Handling

- `Wt::Dbo::Session` - “besitzt” Objekt Mapping und Connections
- Connections durch “Backends”
  - `Wt::Dbo::Backend::Postgres conn(“dbname=postgres user=volker”)`
  - `Wt::Dbo::Backend::Sqlite3 conn(“:memory:”)`
  - `session.setConnection(conn)`
- Mappings durch “mapClass”
  - `session.mapClass<Note>(“notes”);`
  - `session.createTables(); // throw Wt::Dbo::Exception`

Wt::Dbo: Suchen / Listen

- `Session::query<T>()`
  - `session.query<int>("select count(*) from notes");`
  - `session.query<ptr<Note>>("select * from notes")`
    `.where("title = ?").bind("C++");`
- `Session::find<T>()`
  - `ptr<Note> note = session.find<ptr<Note>>().where("title = ?").bind("C++");`
  - `collection<ptr<Note>> res = session.find<Note>().order_by("title");`
- Achtung: Query Result Objekte mit ptr/collection Konvertierung, kein auto!

Wt::Dbo: Fehlerbehandlung

- `Wt::Dbo::Exception`
  - NoUniqueResultException
  - ObjectNotFoundException
  - StaleObjectException - Objekt wurde nebenläufig geändert
- `what()` → Error Message
- `code()` → Backend spezifischer Code (z.B. SQLSTATE)

Wt::Dbo: Spezialisierung für eigene Typen

- Spezialisierung von “field” für eigene “Value” Typen in der DB
  - `field()` → kann Felder in mehreren Columns (der gleichen Tabelle) speichern
  - `Wt::Dbo::dbo_traits` z.B. für eigene ID / Versions Felder
  - Eigenes lese / schreibe Verhalten: `Wt::Dbo::sql_value_traits` spezialisieren !

```
struct Coordinate {
  int x, y;
};
namespace Wt {
  namespace Dbo {
     template <class Action>
     void field(Action& action, Coordinate& coordinate, const std::string& name, int size = -1)
     {
       field(action, coordinate.x, name + "_x");
       field(action, coordinate.y, name + "_y");
     }
  } // namespace Dbo
} // namespace Wt
```

SQLPP11

https://github.com/rbock/sqlpp11

SQLPP11

- “Domain Specific Language” ähnlich LINQ in C#
- Basiert auf generierter C++ Repräsentation des DB Schemas
- Implementiert mit sehr viel TMP
  - Mapping der SQL Datentypen
  - Expression templates z.B. für “WHERE” Klauseln
- Datenbank spezifische “Connectoren” implementieren Dialekte
  - Codegenerierung per “serialize()” kann je nach DB überschrieben werden
  - Connectoren: aktuell PostgreSQL, MySQL, SQLite, STL Container, (Sybase ASE)

SQLPP11

- Datenbank spezifische “Connectoren” implementieren Dialekte
  - Codegenerierung per “serialize()” kann je nach DB überschrieben werden
- Connectoren:
  - PostgreSQL, MySQL, SQLite3
  - STL Container
  - (Sybase ASE)
- Connectoren sind werden gelinkt, SQLPP “Header only”

SQLPP11 - Beispiel Query

```
sqlpp::sqlite3::connection_config config;
config.path_to_database = ":memory:";
config.flags = SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE;
config.debug = true;

sqlpp::sqlite3::connection db(config);

NotesTable notes;

for (const auto& row : db(select(notes.title, notes.content, notes.notebook)
                          .from(notes)
                          .where(notes.notebook > 2 and notes.title.like("%C++%"))))
{
   if (row.content.is_null())
       std::cerr << "content is null" << std::endl;
   else
      std::string content = row.content; // implicit string conversion for varchar fields
   int notebook = row.notebook;          // implicit int conversion
}
```

SQLPP11 - Abfragen

- `select(all_of()).from(join()).where().group_by().having().order_by().limit()...;`
- `insert_into(notes).set(notes.title = “foo”, notes.content = “bar”, ...);`
- `update(notes).set(notes.title = “baz”).where(notes.id == 1);`
- `remove_from(notes).where(notes.id == 1);`
- Für nicht “ausdrückbare” Queries: `db.execute(“CREATE TABLE ...”);`

Demo !

- Doku: https://github.com/rbock/sqlpp11/wiki

SQLPP11 - Schemadefinition

- Mapping von Klassen / Feldern per TMP
- viel syntaktischer Overhead!

Demo!

SQLPP11 - Schemadefinition mit Boost PPGen

- Makros zur Feld / Tabellendefinition
- Keine Schemaverwaltung

```
SQLPP_DECLARE_TABLE(
    (notes),
    (id          , int            , SQLPP_PRIMARY_KEY)
    (title       , varchar(255)   , SQLPP_NOT_NULL   )
    (content     , varchar(255)   , SQLPP_NULL       )
    (notebook    , int            , SQLPP_NOT_NULL    )
    (reminder    , timestamp      , SQLPP_NULL       )
    (last_change , timestamp      , SQLPP_NOT_NULL   )
)
```

SQLPP11 - Schemaverwaltung per Python Generator

- Python SQL Alchemy Modul zur Schemaverwaltung
  - Verwaltet Schema und bietet DB Abstraktion, unterstützt Migration
  - Skript zur Generierung von Headern basierend auf Python Metadaten (aktuell intern)

```
from sqlalchemy import *

engine = create_engine('sqlite:///:memory:')
metadata = MetaData()

notes = Table('notes', metadata,
   Column('id', Integer, primary_key=True),
   Column('title', String(255), nullable=False),
[...]
   Column('reminder', DateTime, server_default=text('NOW()'))
)
metadata.create_all(engine)
# sqlpp_gen.gen_source(“db_schema/”, “notes::schema”, metadata)
```

Überblick

- C DB APIs:
  - für "kleine" Projekte oder Spezialaufgaben
- QtSql
  - Gute einfache all-round Schnittstelle für GUI Anwendungen oder Tools
- Wt::Dbo
  - wenn man ein ORM möchte gute API, alternative ODB, QxORM ?
  - elegantes Design durch Templates, keine getrennte Schemadefinition oder Code Generierung
- SQLPP11
  - "Experimentell", aber sehr flexibler und schlanker Ansatz
  - Zur Reduziernug der Compilezeiten und Schutz anderer Entwickler vor TMP gut kapseln !

Alternativen

- ODBC (UnixODBC)
  - zu komplex, langsam, nur wenn es nicht anders geht !
- SOCI / POCO SQL
  - Ähnlich QtSql, einfaches SQL Interface, kein Sybase Connector (außer ODBC)
- ODB
  - ORM, ähnlich zu Wt::Dbo - GPL/Kommerziell, Achtung: Mapping per GCC Plugin!
- SQLAPI++
  - Kommerziell, aber viele unterstützte DBs, rel. Low Level Interface

Vielen Dank !

Fragen ???

Beispielcode: https://github.com/volka/talks

http://geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html