A modern database interface for C++

Erik Smith

WHY DO WE CARE

some data points

- sqlite installed base > # smart phones
- JDBC (java's database API) was introduced in 1998

PREVIOUS PROPOSAL WORK

N3886 - Johann Anhofer - A Proposal to add a Database Access Layer to the Standard Library
N3612 - Thomas Neumann - Desiderata of a C++11 Database Interface
N3458 - Thomas Neumann - Simple Database Integration in C++
N3415 - Bill Seymour - A Database Access Library

THE RELATIONAL MODEL

"rowset"

Table

columns

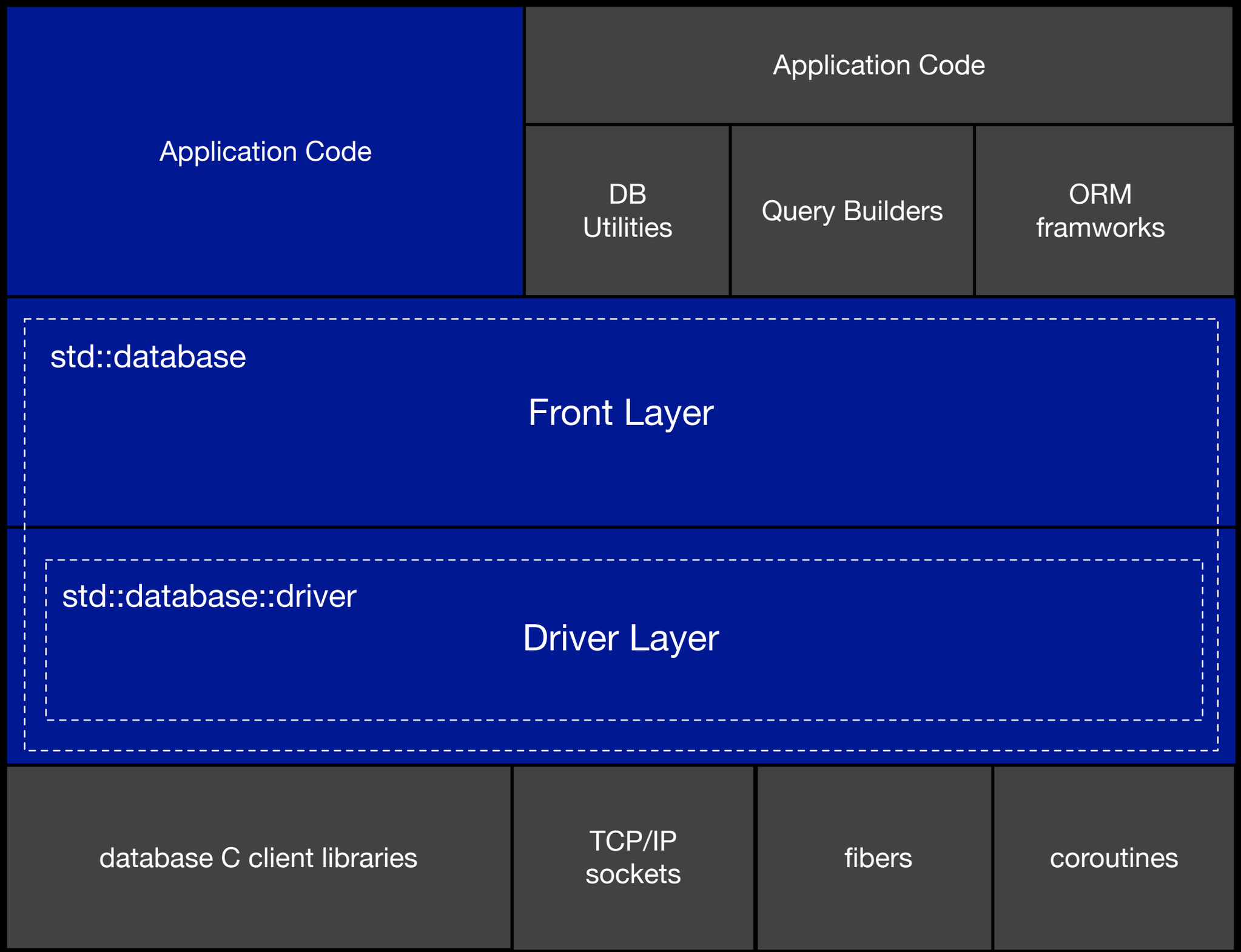
rows

PERSON			
ID INT	LAST_NAME VARCHAR(30)	FIRST_NAME VARCHAR(30)	BIRTHDATE DATE
1	Yokomoto	Akiko	1990-05-03
2	Green	Marjorie	1972-02-06
3	Hoffman	Paul	1995-07-01

SQL query

```
select last_name from person
where birthdate <"1991-01-01"
```

THE CLIENT STACK



EXAMPLE

```
using cppstddb::mysql;
auto db = create_database(“mysql://server/db”);
db
.query("select * from person”)
.rows()
.write(cout);
```

no explicit types

INCLUDE & NAMESPACE ALIAS

details for particular implementation

```
#include <cppstddb/mysql/database.h>

using cppstddb::mysql;
auto db = create_database(“mysql://server/db”);
db
    .query("select * from person”)
    .rows()
    .write(cout);
```

COMPOSABLE

uses a classic method chaining approach

```
class connection {
    auto query(const string& q) {
        ...
        return *this;
};
```

recent Uniform Function Call Syntax (UCFS) proposals (N4474) support this approach

con.query(q) failover to query(con,q);

DIRECT VS POLY

direct: using one specific database type
(no indirection cost)

```
#include <cppstddb/mysql/database.h>
using cppstddb::mysql;
auto db = create_database("mysql://server/db");
```

poly: choose any registered database type at run-time
(more on poly later)

THE TYPES

database	top level context (shared)
connection	connection to server (per thread)
statement	query execution, prepared statements, input binding
columnset	meta-data for columns
column	meta-data for specific column
rowset	accessor for query results
row	row accessor (proxy) type
field	field accessor (proxy), type conversion.

EXAMPLE IN EXPANDED (“CLASSIC”) FORM

```
using namespace cppstddb::mysql;
auto db = create_database(“mysql://server/db");
auto con = db.connection();
auto stmt = con.query("select * from person”);
auto rows = stmt.rows();

for (auto i = rows.begin(); i != rows.end(); ++i) {
    for(int c = 0; c != row.width(); ++c) {
        auto field = row[c];
        cout << “value: ” << field << “\n”;
    }
    cout << “\n”;
}
```

DUAL METHOD STYLES

by type name

```
db
    .connection(uri)
    .statement()
    .rowset()
```

natural style

```
db
    .connect(uri)
    .query(sql)
    .rows()
```

keeping both for now as they both make sense

FLEXIBLE

these are equivalent

db.connection().statement(“select * from t”).execute().rows()
db.connection().query(“select * from t”).execute().rows()
db.connection().query(“select * from t”).rows()
db.query("select * from t”).rows()

hard to get wrong

WHY REFERENCE TYPES?

a non-nested scope example (think MVC)

```
auto get_customers_by_zip(const string& zip) {
    auto db = create_database("mysql://server/db");
    db
        .query("select from person where name=?", zip)
        .rows();
}

auto local_customers = get_customers_by_zip("92122");
```

rowset is safe to escape

rowset holds shared state in std::shard_ptr

DATABASE

```
auto db = create_database;
auto db = create_database("mysql://server/db");          // default URI

auto con = database.connection();                        // uses default URI
auto con = database.connection("mysql://server2/db");    // specific URI
auto con = database.connection("server");                // named source

auto con = database.create_connection();                 // new connection
auto con = database.create_connection("mysql://server2/db");

db.query("insert into person values(1, 'joe');
```

connection: returns same connection per thread
create_connection: new connection for same thread

CONNECTION STRING

URI based string

db.connection(“mysql://server/db?username=app”)

Named source

db.connection(“mysql”)

Custom Resolver →

config file

```
"databases": [
  {
    "name": "mysql",
    "type": "mysql",
    "server": "127.0.0.1",
    "database": "test",
    "username": "",
    "password": ""
  },
```

CONNECTION

```
auto db = connection.database();        // parent accessor

connection.auto_commit(false);      // for transactions
connection.begin();
connection.save();
connection.commit();
connection.rollback();
connection.isolationLevel();
```

TRANSACTIONS & SCOPE EXIT

```
auto con1 = db.connection(“server1").auto_commit(false);
auto con2 = db.connection(“server2").auto_commit(false);

auto goodbye = make_scope_exit([]() {     // proposal N4189
    con1.rollback();
    con2.rollback();
});

con1.begin.query("insert into account(id,amount) values(1,-500000)”);
con2.begin.query("insert into account(id,amount) values(2, 500000)");

con1.commit();
con2.commit();
```

STATEMENT

statement facilitates prepared statements / input binding

```
auto stmt = con
    .query("insert into person(id, name) values(?,?)");

stmt.execute(1,"Doug");
stmt.execute(2,"Cathy");
stmt.execute(3,"Robert");
```

essential for security (SQL injection attacks)

ROWSET

accessors

```
row.width()          // number of columns in row set
row.columns()        // column metadata
row.length()         // number of results (at end or materialized)
```

what about row traversal?

APPROACHES TO ROWSET TRAVERSAL / ACCESS

- next(), done()
- front(), pop_front(), empty()
- iterators?

ITERATORS!

rowset uses InputIterators

forward only, only current result accessible

RANGE BASED FOR

```
auto db = create_database(“mysql://server/db”);
auto rows = db.query("select * from person”).rows()

for (auto row : rows) {
    for (col : row.columns()) {
        cout << “value: “ << row[col] << “\n”;
    }
    cout << “\n”;
}
```

RANGE BASED FOR

rows is an Iterable type with iterator proxies - refine

```
for (auto row : rows)                // works - copy ok!
for (auto&& row : rows)              // also fine
for (auto& row : rows)               // error as expected
for (const auto& row : rows)         // error as expected
for (row : rows)                     // N3853, not today
```

*N3853: Range-Based For-Loops: The Next Generation—Stephan T. Lavavej

ITERATOR OVERHEAD

Are iterators for row traversal bad for performance?

```
class iterator {
...
    bool operator==(const rowset_iterator& rhs) const {
      return
        (rowset_ && !rowset_->done()) ==
        (rhs.rowset_ && !rhs.rowset_->done());
    }
...
};
```

Not that bad for InputIterators, but still have to account for 4 cases at runtime

STD LIBRARY ALGORITHM EXAMPLE 1

```
auto db = create_database();
auto rng = db.query("select name,score from score").rows();

auto key = "carol";
auto i = find_if(rng.begin(), rng.end(), [key](auto row) {
        return row[“name”] == key; });

if (i != rng.end()) {
    cout << “found row: " << *i” << "\n";
}
```

STD LIBRARY ALGORITHM EXAMPLE 2

```
auto db = create_database();
auto rng = db.query("select name,score from score”).rows();

int sum = accumulate(rng.begin(), rng.end(), 0, [](int sum, auto row) {
        return sum + row[“score”].as<int>();});

cout << "sum: " << sum << “\n";  // 150
```

ITERATOR ISSUES

- proxy iterators will be problematic for rowset types beyond InputIterator

RANGES !

Ranges are a proposed addition to the C++ standard library from Eric Niebler.

Ranges address existing issues with iterators, particularly with composability, and bring a whole new class of functionality with range adapters.

RANGE SUPPORT

A rowset is an InputRange

Iterable Concept

begin(s) and end(s) defined, not necessarily same type

- No pure Iterable type exposed in interface

Range Concept

Extends Iterable
ightweight
default constructible, copyable, assignable,destructible

- A rowset is a range
- Element data is accessed through internal shared_ptr

InputRange

iterator type is InputIterator

- Only the current row is accessible.

RANGE OPTIMIZATIONS

standard library (C++98) input iterator comparison:

```
bool operator==(const rowset_iterator& rhs) const {
    return
        (rowset_ && !rowset_->done()) ==
        (rhs.rowset_ && !rhs.rowset_->done());
}
```

comparison optimized for ranges:

```
bool operator==(const sentinel& rhs) const {
    return !rowset_ || rowset_->done();
}
```

RANGES: PASS RANGE INSTEAD OF PAIR

```
auto db = create_database();
auto rng = db.query("select name,score from score").rows();

auto key = "carol";
auto i = find_if(
    rng,
    [key](auto row) {return row[“name”] == key; });

if (i != rng.end()) {
    cout << “found row: " << *i” << "\n";
}
```

Compose the range in the argument to find_if

RANGE ADAPTORS

view::filter like a lazy version of std::copy_if

```
auto db = create_database();
int min_score = 20;
auto rng = view::filter(
        db.query("select name,score from score").rows(),
        [min_score](auto row) {return row["score"] > min_score; });
```

view::filter returns a range that is set up to filter lazily

great for proxy InputRanges like rowset

std::copy_if won't work here

ROW

```
auto field = row[0];                    // by column index

auto field = row[“FIRST_NAME”];  // by column name

auto field = row[“first_name”];      // case insensitive

auto field = row[column];               // by column
```

FIELD

```
cout << field << "\n";          // stream inserter=
field.as<T>()                   // type T
field.as<int>()                 // int
field.as<long>()                // long
field.as<std::string>()         // std::string
field.as<std::string_view>()    // std::string_view

field.variant()                 // as std::variant (nothrow)
field.optional<T>()             // as std::optional<T> (nothrow)
```

FIELD ACCESSORS

```
field.is_null()     // is the value null
field.name()        // name of column
field.type()        // type enumeration
```

ROW FIELD INDEXING

```
auto field = row[“name“];     // easy but less efficient
```

```
auto field = row[1];          // efficient but less readable
```

MORE EFFICIENT ROW FIELD INDEXING

more efficient, readable?

```
auto id_idx = rows.columns()[“id”];
auto name_idx = rows.columns()[“last_name”];
```

```
for(auto row : rows) {
    auto id = row[id_idx];
    auto name = row[name_idx];
}
```

harder to maintain

SINGLE ROW QUERIES WITH into

```
string a;
int b;
date d;
variant d;
db
    .query("select a,b,c,d from table”)
    .into(a,b,c,d);
```

ROW LEVEL into

better living though variadics

```
auto rows =
   .query("select id, name from person")
   .rows();

int id;
string name;
for (auto row : rows) row.into(id, name);
```

INTO RANGE

```
variant rng;
query(sql).into(rng);
query(sql).row().into(rng);
```

need to disambiguate rowset vs row

SERIALIZATION

equivalent syntax to for object serialization

```
person p

db.query(sql).into(p);
rows.into(p)
row.into(p)
row >> p;
```

Important use case for reflection in the standard

HANDLE ACCESSORS

```
auto con = db.connection();
auto mysql = connection.handle();          // typed as MYSQL*
auto mysql = db.connection.handle();       // lifetime fail
```

OUTPUT BINDING

ID	NAME
2	JOE

- A C level buffer interface
- RowSet handles internally

ARRAY OUTPUT BINDING

```
auto rows = con
    .query("select * from t1”)
    .rows();

int sum;
for(auto r : rows)
   sum += r[0].as<int> + r[1].as<int>;
```

1000 row table

A INT	B INT
0	1
1	2
2	3
…	…
999	1000

203 ms

ARRAY OUTPUT BINDING

```
auto rows = con
    .row_array_size(100)
    .query("select * from t1”)
     .rows();

int sum;
for(auto r : rows)
   sum += r[0].as<int> + r[1].as<int>;
```

1000 row table

A INT	B INT
0	1
1	2
2	3
…	…
999	1000

32 ms

ARRAY OUTPUT BINDING

```
auto rs = con
    .row_array_size(500)
    .query("select * from t1”)
     .rows();

int sum;
for(auto r : rows)
   sum += r[0].as<int> + r[1].as<int>;
```

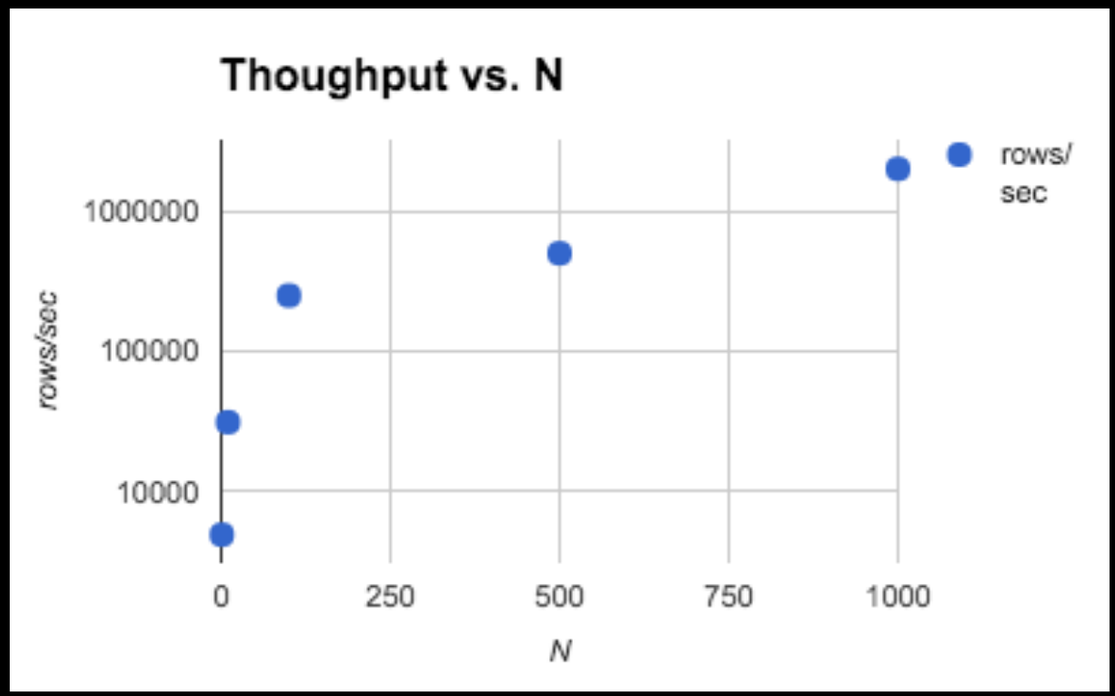
1000 row table

A INT	B INT
0	1
1	2
2	3
…	…
999	1000

2 ms

ARRAY OUTPUT BINDING

```
auto rs = con
    .row_array_size(1000)
    .query("select * from t1”)
     .rows();
```




```
int sum;
for(auto r : rows)
   sum += r[0].as<int> + r[1].as<int>;
```

601 µs 2M rows/sec

400X faster

CONTIGUOUS ARRAY BINDING

some databases can bind column arrays into the same contiguous memory (struct binding / skip parameters)

DETACHED ROWSETS

```
auto rows = detached_rows();
```

- detachable rowset can detach from connection
- rowset is now a RandomAccessRange
- no additional copying
- rowset caching enabler

INPUT BINDING

same as before

```
auto stmt =
    create_database("mysql://server/db")
    .query("insert into table(id, name) values(?,?)");

for(auto d : data) stmt.execute(d.id, d.name);
```

ARRAY INPUT BINDING

```
auto stmt =
    create_database("mysql://server/db")
    .query("insert into table(id, name) values(?,?)")
    .row_array_size(1000);

for(auto d : data) stmt.query(d.id, d.name);
```

huge performance win

TYPE CONVERSION

- Two layers (driver & front end)
- Native driver binding is default

POLICIES

```
struct my_policy {…}
auto db = database<my_policy>;
```

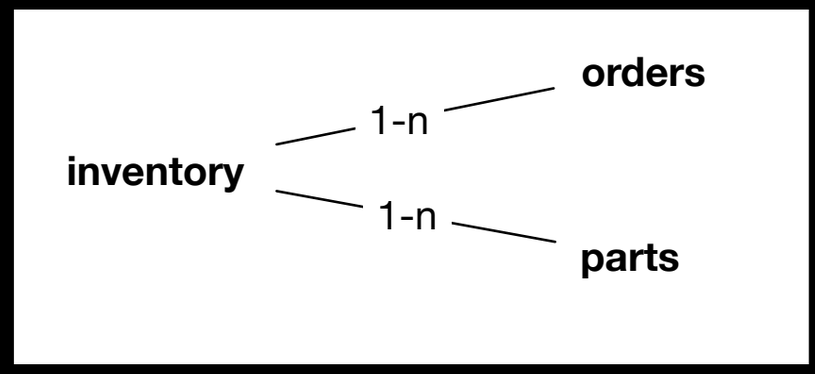
- Custom allocators
- Pluggable connection pool
- assert on release build for cross/illegal type conversions
- Scoped types (no RC)
- Omit handle accessors

UTILITY EXAMPLE: JOIN

```
auto inventory = db.create_connection().query("select id,* inventory").rows();
auto orders = db.create_connection().query("select * from orders order by id").rows();
auto parts = db.create_connection().query("select * from parts order by id").rows();

auto joined_rows = ordered_natural_join(inventory, orders, parts);

for(auto row : joined_rows) {
    int id, order_id, part_id;
    inventory.into(id,...);
    orders.into(order_id,…);
    parts.into(part_id,…);
}
```



overlapping key condition: all tables ordered consistently

takes O(1) space

an approach to the "multiple hierarchies" problem

Use IOT/clustered indexes when tables are big

JOIN ALGORITHMS

```
// ordered join with specific column conditions:
auto j  = ordered_join(inventory.join(orders).on(orders, “part_id”);
```

other more general join possibilities with detached (RandomAccess) rowsets

POLYMORPHIC INTERFACE

Direct Interface

```
auto db = create_database();
auto con = db.connection("mysql://server/db");
```

Poly Interface

```
auto db = cppstddb::poly::create_database();
auto con1 = db.connection("mysql://server/db");
auto con2 = db.connection("sqlite://file.sqlite");
```

poly could conceivably be in the standard library

POLY: ADDING DRIVERS

```
#include <cppstddb/poly/database.h>

using cppstddb::poly;
database.register<cppstddb::sqlite::database>();
database.register<cppstddb::mysql::database>();
database.register<cppstddb::oracle::database>();
database.register<cppstddb::postgres::database>();
```

REFERENCE IMPLEMENTATION TEST SUITE

```
test_all<cppstddb::mysql::database>("mysql");
test_all<cppstddb::sqlite::database>("sqlite");
test_all<cppstddb::oracle::database>("oracle");
...
```

- Templated test framework
- Runs test twice: once direct, ones through poly driver
- Runs carefully in sandbox database

IMPLEMENTATION DETAILS

two layer design

Front End

- Handles reference counting details for all types
- Defines all interface functions
- Consolidates calls to the driver
- Manages state
- Connection pooling

Driver

- Implement driver specific details

DRIVER INTERFACE

```
namespace cppstddb { namespace mysql {

using database =
    std::database::basic_database<
        driver<default_policy>,default_policy>;

auto create_database() {return database<default_policy>();}
```

```
template<class policy> class driver {
    class database {…}
    class connection {…}
    class statement {…}
    class result {…}
}

}}
```

Type name correspondence
between front and
driver layers

DRIVER INTERFACE

```
template<class policy> class driver {
    class database {...};
    class connection {
        connection(database *db, source src, allocator *a) {...}
    };
    class statement {
        statement(connection *con, string sql, allocator *a) {...}
    };
    class result {
        result(ctatement *stmt, allocator *a) {...}
    };
}
```

CHALLENGE: POLY EXECUTE

```
template<typename... Args> auto execute(Args... args) {
    // unpack variants
    // form 2nd variadic template call in driver layer
}
```

#1 Avoid requiring the drivers to handle Variant arguments

#2 Need to transport arguments from one variadic template query call to another at *run time*

Run time ⟶ Compile Time

APPROACH

Front End

stmt.execute("joe", date(2015,2,1), 42)

vector<variant_t>

V<string>	V<int>	V<date>

variadic_dispatch (form unpack_variants call with 3 args)

Driver

unpack_variants(v[0],v[1],v[2])

stmt.execute("joe", Date(2015,2,1), 42)

string date int

Is this a good idea?

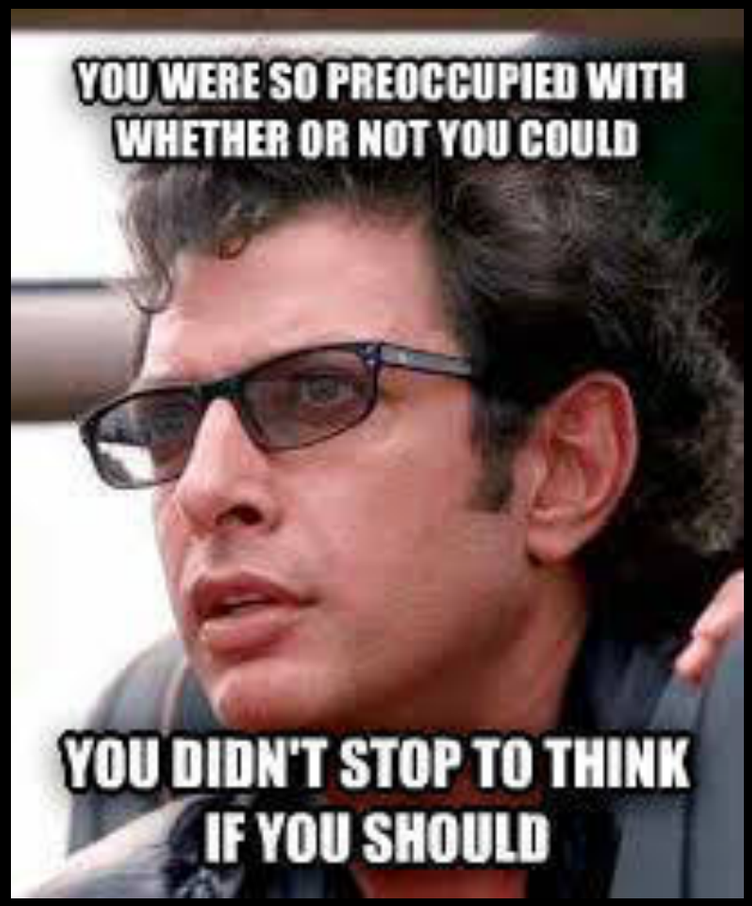
2 minutes to compile

REALITY CHECK

10 types

5 arguments

P(10,5) = 30240 function calls



Alternative:

auto execute(std::vector<variant_t>& args); // an additional call

OR

execute must support variants

not a big issue

NON-BLOCKING INTERFACE

Can be implemented for underlying database clients that that support NBIO

“We're Gonna Need A Bigger Database”

REFERENCE IMPLEMENTATION ROADMAP

current support (limited depth)

phase II (soon)

phase III


QUESTIONS

github: https://github.com/cruisercoder/cppstddb

twitter: @cruisercoder