# No raw string queries

# Intuitive syntax

# Comfortable interface - one code line per single query

# Built with modern C++14 features (no macros and external scripts)

# CRUD support

Create, read, update and delete support.

## Insert

User user{-1, "Jonh", "Doe", 664416000, std::make\_unique<std::string>("url\_to\_heaven"), 3 };

auto insertedId = storage.insert(user);

cout << "insertedId = " << insertedId << endl; // insertedId = 8

user.id = insertedId;

## Replace

if we need to insert a new user with specified id call storage.replace(user); instead of insert.

## Read

try{

auto user = storage.get<User>(insertedId);

cout << "user = " << user.firstName << " " << user.lastName << endl;

}catch(std::system\_error e) {

cout << e.what() << endl;

}catch(...){

cout << "unknown exeption" << endl;

}

Probably you may not like throwing exceptions. Me too. Exception std::system\_error is thrown because return type in get function is not nullable. You can use alternative version get\_pointer which returns std::unique\_ptr

if(auto user = storage.get\_pointer<User>(insertedId)){

cout << "user = " << user->firstName << " " << user->lastName << endl;

}else{

cout << "no user with id " << insertedId << endl;

}

std::unique\_ptr is used as optional in sqlite\_orm. Of course there is class optional in C++14 just define SQLITE\_ORM\_OPTIONAL\_SUPPORTED

we can extract all objects into std::vector

auto allUsers = storage.get\_all<User>();

cout << "allUsers (" << allUsers.size() << "):" << endl;

for(auto &user : allUsers) {

cout << storage.dump(user) << endl;

}

let's get all users in std::list, not std::vector:

auto allUsersList = storage.get\_all<User, std::list<User>>();

(need push\_back())

get\_all can be too heavy for memory so you can iterate row by row (i.e. object by object):

for(auto &user : storage.iterate<User>()) {

cout << storage.dump(user) << endl;

}

## update

It updates row by id provided in user object and sets all other non primary\_key fields to values stored in the passed user object

user.firstName = "Nicholas";

user.imageUrl = "https://cdn1.iconfinder.com/data/icons/man-icon-set/100/man\_icon-21-512.png"

storage.update(user);

there is a non-CRUD update version update\_all

storage.update\_all(set(c(&User::lastName) = "Hardey",

c(&User::typeId) = 2),

where(c(&User::firstName) == "Tom"));

## delete

storage.remove<User>(insertedId)

CRUD functions get, get\_pointer, remove, update (not insert) work only if your type has a primary key column… else look at examples\date\_time.cpp

# Pure select query support

# Prepared statements support

Prepared statements are strongly typed.

// SELECT doctor\_id

// FROM visits

// WHERE LENGTH(patient\_name) > 8

auto selectStatement = storage.prepare(select(&Visit::doctor\_id, where(length(&Visit::patient\_name) > 8)));

cout << "selectStatement = " << selectStatement.sql() << endl; // prints "SELECT doctor\_id FROM ..."

auto rows = storage.execute(selectStatement); // rows is std::vector<decltype(Visit::doctor\_id)>

// SELECT doctor\_id

// FROM visits

// WHERE LENGTH(patient\_name) > 11

get<0>(selectStatement) = 11;

auto rows2 = storage.execute(selectStatement);

# aggregate functions

// SELECT AVG(id) FROM users

auto averageId = storage.avg(&User::id);

cout << "averageId = " << averageId << endl; // averageId = 4.5

// SELECT AVG(birth\_date) FROM users

auto averageBirthDate = storage.avg(&User::birthDate);

cout << "averageBirthDate = " << averageBirthDate << endl; // averageBirthDate = 6.64416e+08

// SELECT COUNT(\*) FROM users

auto usersCount = storage.count<User>();

cout << "users count = " << usersCount << endl; // users count = 8

// SELECT COUNT(id) FROM users

auto countId = storage.count(&User::id);

cout << "countId = " << countId << endl; // countId = 8

// SELECT COUNT(image\_url) FROM users

auto countImageUrl = storage.count(&User::imageUrl);

cout << "countImageUrl = " << countImageUrl << endl; // countImageUrl = 5

// SELECT GROUP\_CONCAT(id) FROM users

auto concatedUserId = storage.group\_concat(&User::id);

cout << "concatedUserId = " << concatedUserId << endl; // concatedUserId = 1,2,3,4,5,6,7,8

// SELECT GROUP\_CONCAT(id, "---") FROM users

auto concatedUserIdWithDashes = storage.group\_concat(&User::id, "---");

cout << "concatedUserIdWithDashes = " << concatedUserIdWithDashes << endl; // concatedUserIdWithDashes = 1---2---3---4---5---6---7---8

// SELECT MAX(id) FROM users

if(auto maxId = storage.max(&User::id)){

cout << "maxId = " << \*maxId <<endl; // maxId = 12 (maxId is std::unique\_ptr<int>)

}else{

cout << "maxId is null" << endl;

}

// SELECT MAX(first\_name) FROM users

if(auto maxFirstName = storage.max(&User::firstName)){

cout << "maxFirstName = " << \*maxFirstName << endl; // maxFirstName = Jonh (maxFirstName is std::unique\_ptr<std::string>)

}else{

cout << "maxFirstName is null" << endl;

}

// SELECT MIN(id) FROM users

if(auto minId = storage.min(&User::id)){

cout << "minId = " << \*minId << endl; // minId = 1 (minId is std::unique\_ptr<int>)

}else{

cout << "minId is null" << endl;

}

// SELECT MIN(last\_name) FROM users

if(auto minLastName = storage.min(&User::lastName)){

cout << "minLastName = " << \*minLastName << endl; // minLastName = Doe

}else{

cout << "minLastName is null" << endl;

}

// SELECT SUM(id) FROM users

if(auto sumId = storage.sum(&User::id)){ // sumId is std::unique\_ptr<int>

cout << "sumId = " << \*sumId << endl;

}else{

cout << "sumId is null" << endl;

}

// SELECT TOTAL(id) FROM users

auto totalId = storage.total(&User::id);

cout << "totalId = " << totalId << endl; // totalId is double (always)

# UNION, EXCEPT and INTERSECT support

# STL compatible

# Custom types binding support

# BLOB support - maps to std::vector<char> or one can bind your custom type

# FOREIGN KEY support

# Composite key support

# JOIN support

You can perform simple JOIN, CROSS JOIN, INNER JOIN, LEFT JOIN or LEFT OUTER JOIN in your query.

auto crossed = storage.select(columns(&Fondo::id, &Fondo::nombre, &Inversion::fkey\_fondo, &Inversion::num\_participaciones),

cross\_join<Inversion>());

# Transactions support

Explicit calls:

auto secondUser = storage.get<User>(2);

storage.begin\_transaction();

secondUser.typeId = 3;

storage.update(secondUser);

storage.rollback(); // or storage.commit();

secondUser = storage.get<decltype(secondUser)>(secondUser.id);

assert(secondUser.typeId != 3);

implicit call:

storage.transaction([&] () mutable { // mutable keyword allows make non-const function calls

auto secondUser = storage.get<User>(2);

secondUser.typeId = 1;

storage.update(secondUser);

auto gottaRollback = bool(rand() % 2);

if(gottaRollback){ // dummy condition for test

return false; // exits lambda and calls ROLLBACK

}

return true; // exits lambda and calls COMMIT

});

storage.transaction([&] () mutable {

storage.remove\_all<User>(where(c(&User::id) < 100));

auto usersRemoved = storage.changes();

cout << "usersRemoved = " << usersRemoved << endl;

return true;

});

auto commited = storage.transaction([&] () mutable {

auto secondUser = storage.get<User>(2);

secondUser.typeId = 1;

storage.update(secondUser);

auto gottaRollback = bool(rand() % 2);

if(gottaRollback){ // dummy condition for test

return false; // exits lambda and calls ROLLBACK

}

return true; // exits lambda and calls COMMIT

});

if(commited){

cout << "Commited successfully, go on." << endl;

}else{

cerr << "Commit failed, process an error" << endl;

}

try{

auto guard = storage.transaction\_guard(); // calls BEGIN TRANSACTION and returns guard object

user.name = "Paul";

auto notExisting = storage.get<User>(-1); // exception is thrown here, guard calls ROLLBACK in its destructor

guard.commit();

}catch(...){

cerr << "exception" << endl;

}

# Migrations functionality

To some extent by sync\_schema

# Powerful conditions

You also can select objects with custom where conditions with =, !=, >, >=, <, <=, IN, BETWEEN and LIKE

# ORDER BY and LIMIT, OFFSET support

// `SELECT \* FROM users ORDER BY id`

auto orderedUsers = storage.get\_all<User>(order\_by(&User::id));

cout << "orderedUsers count = " << orderedUsers.size() << endl;

for(auto &user : orderedUsers) {

cout << storage.dump(user) << endl;

}

// `SELECT \* FROM users WHERE id < 250 ORDER BY first\_name`

auto orderedUsers2 = storage.get\_all<User>(where(c(&User::id) < 250), order\_by(&User::firstName));

cout << "orderedUsers2 count = " << orderedUsers2.size() << endl;

for(auto &user : orderedUsers2) {

cout << storage.dump(user) << endl;

}

// `SELECT \* FROM users WHERE id > 100 ORDER BY first\_name ASC`

auto orderedUsers3 = storage.get\_all<User>(where(c(&User::id) > 100), order\_by(&User::firstName).asc());

cout << "orderedUsers3 count = " << orderedUsers3.size() << endl;

for(auto &user : orderedUsers3) {

cout << storage.dump(user) << endl;

}

// `SELECT \* FROM users ORDER BY id DESC`

auto orderedUsers4 = storage.get\_all<User>(order\_by(&User::id).desc());

cout << "orderedUsers4 count = " << orderedUsers4.size() << endl;

for(auto &user : orderedUsers4) {

cout << storage.dump(user) << endl;

}

// `SELECT first\_name FROM users ORDER BY ID DESC`

auto orderedFirstNames = storage.select(&User::firstName, order\_by(&User::id).desc());

cout << "orderedFirstNames count = " << orderedFirstNames.size() << endl;

for(auto &firstName : orderedFirstNames) {

cout << "firstName = " << firstName << endl;

}

There are three available versions of LIMIT/OFFSET options:

* LIMIT %limit%
* LIMIT %limit% OFFSET %offset%
* LIMIT %offset%, %limit%

All these versions available with the same interface:

// `SELECT \* FROM users WHERE id > 250 ORDER BY id LIMIT 5`

auto limited5 = storage.get\_all<User>(where(c(&User::id) > 250),

order\_by(&User::id),

limit(5));

cout << "limited5 count = " << limited5.size() << endl;

for(auto &user : limited5) {

cout << storage.dump(user) << endl;

}

// `SELECT \* FROM users WHERE id > 250 ORDER BY id LIMIT 5, 10`

auto limited5comma10 = storage.get\_all<User>(where(c(&User::id) > 250),

order\_by(&User::id),

limit(5, 10));

cout << "limited5comma10 count = " << limited5comma10.size() << endl;

for(auto &user : limited5comma10) {

cout << storage.dump(user) << endl;

}

// `SELECT \* FROM users WHERE id > 250 ORDER BY id LIMIT 5 OFFSET 10`

auto limit5offset10 = storage.get\_all<User>(where(c(&User::id) > 250),

order\_by(&User::id),

limit(5, offset(10)));

cout << "limit5offset10 count = " << limit5offset10.size() << endl;

for(auto &user : limit5offset10) {

cout << storage.dump(user) << endl;

}

# GROUP BY / DISTINCT support

# INDEX support

# Follows single responsibility principle - no need write code inside your data model classes

# Easy integration - single header only lib.

# The only dependency - libsqlite3

# C++ standard code style

# In memory database support - provide :memory: or empty filename

# COLLATE support

# Limits setting/getting support

# User defined functions support