C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\blob.cpp:

struct User {

int id;

std::string name;

std::vector<char> hash; // binary format

};

int main(int, char\*\*) {

using namespace sqlite\_orm;

auto storage = make\_storage("blob.sqlite",

make\_table("users",

make\_column("id", &User::id, primary\_key()),

make\_column("name", &User::name),

make\_column("hash", &User::hash)));

storage.sync\_schema();

storage.remove\_all<User>();

User alex{

0,

"Alex",

{0x10, 0x20, 0x30, 0x40},

};

alex.id = storage.insert(alex);

cout << "users count = " << storage.count<User>() << endl;

cout << "alex = " << storage.dump(storage.get<User>(alex.id)) << endl;

auto hash = storage.get<User>(alex.id).hash;

assert(hash.size() == 4);

assert(hash[0] == 0x10);

assert(hash[1] == 0x20);

assert(hash[2] == 0x30);

assert(hash[3] == 0x40);

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\case.cpp:

{ // without alias

// SELECT ID, NAME, MARKS,

// CASE

// WHEN MARKS >=80 THEN 'A+'

// WHEN MARKS >=70 THEN 'A'

// WHEN MARKS >=60 THEN 'B'

// WHEN MARKS >=50 THEN 'C'

// ELSE 'Sorry!! Failed'

// END

// FROM STUDENT;

auto rows = storage.select(columns(&Student::id,

&Student::name,

&Student::marks,

case\_<std::string>()

.when(greater\_or\_equal(&Student::marks, 80), then("A+"))

.when(greater\_or\_equal(&Student::marks, 70), then("A"))

.when(greater\_or\_equal(&Student::marks, 60), then("B"))

.when(greater\_or\_equal(&Student::marks, 50), then("C"))

.else\_("Sorry!! Failed")

.end()));

for(auto& row: rows) {

cout << std::get<0>(row) << ' ' << std::get<1>(row) << ' ' << std::get<2>(row) << ' ' << std::get<3>(row)

<< endl;

}

cout << endl;

}

////////////////////////////////

struct GradeAlias : alias\_tag {

static const std::string& get() {

static const std::string res = "Grade";

return res;

}

};

// SELECT ID, NAME, MARKS,

// CASE

// WHEN MARKS >=80 THEN 'A+'

// WHEN MARKS >=70 THEN 'A'

// WHEN MARKS >=60 THEN 'B'

// WHEN MARKS >=50 THEN 'C'

// ELSE 'Sorry!! Failed'

// END as 'Grade'

// FROM STUDENT;

auto rows = storage.select(columns(&Student::id,

&Student::name,

&Student::marks,

as<GradeAlias>(case\_<std::string>()

.when(greater\_or\_equal(&Student::marks, 80), then("A+"))

.when(greater\_or\_equal(&Student::marks, 70), then("A"))

.when(greater\_or\_equal(&Student::marks, 60), then("B"))

.when(greater\_or\_equal(&Student::marks, 50), then("C"))

.else\_("Sorry!! Failed")

.end())));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\check.cpp:

struct Contact {

int id = 0;

std::string firstName;

std::string lastName;

std::string email;

std::string phone;

};

struct Product {

int id = 0;

std::string name;

float listPrice = 0;

float discount = 0;

};

auto storage = make\_storage(":memory:",

make\_table("contacts",

make\_column("contact\_id", &Contact::id, primary\_key()),

make\_column("first\_name", &Contact::firstName),

make\_column("last\_name", &Contact::lastName),

make\_column("email", &Contact::email),

make\_column("phone", &Contact::phone),

check(length(&Contact::phone) >= 10)),

make\_table("products",

make\_column("product\_id", &Product::id, primary\_key()),

make\_column("product\_name", &Product::name),

make\_column("list\_price", &Product::listPrice),

make\_column("discount", &Product::discount, default\_value(0)),

check(c(&Product::listPrice) >= &Product::discount and

c(&Product::discount) >= 0 and c(&Product::listPrice) >= 0)));

storage.sync\_schema();

try {

storage.insert(Contact{0, "John", "Doe", {}, "408123456"});

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

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

}

storage.insert(Contact{0, "John", "Doe", {}, "(408)-123-456"});

try {

storage.insert(Product{0, "New Product", 900, 1000});

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

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

}

try {

storage.insert(Product{0, "New XFactor", 1000, -10});

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

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

}

///////////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\collate.cpp:

auto storage = make\_storage(

"collate.sqlite",

make\_table("users",

make\_column("id", &User::id, primary\_key()),

make\_column("name", &User::name),

make\_column("created\_at", &User::createdAt)),

make\_table("foo", make\_column("text", &Foo::text, collate\_nocase()), make\_column("baz", &Foo::baz)));

storage.sync\_schema();

storage.remove\_all<User>();

storage.remove\_all<Foo>();

storage.insert(User{0, "Lil Kim", std::time(nullptr)});

storage.insert(User{0, "lil kim", std::time(nullptr)});

storage.insert(User{0, "Nicki Minaj", std::time(nullptr)});

// SELECT COUNT(\*)

// FROM users

// WHERE name = 'lil kim'

auto preciseLilKimsCount = storage.count<User>(where(is\_equal(&User::name, "lil kim")));

cout << "preciseLilKimsCount = " << preciseLilKimsCount << endl;

// SELECT COUNT(\*) FROM users WHERE name = 'lil kim' COLLATE NOCASE

auto nocaseCount = storage.count<User>(where(is\_equal(&User::name, "lil kim").collate\_nocase()));

cout << "nocaseCount = " << nocaseCount << endl;

// SELECT COUNT(\*) FROM users

cout << "total users count = " << storage.count<User>() << endl;

storage.insert(Foo{"Touch", 10});

storage.insert(Foo{"touch", 20});

cout << "foo count = " << storage.count<Foo>(where(c(&Foo::text) == "touch")) << endl;

// SELECT id

// FROM users

// ORDER BY name COLLATE RTRIM ASC

auto rows = storage.select(&User::id, order\_by(&User::name).collate\_rtrim().asc());

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

//////////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\composite\_key.cpp:

auto storage = make\_storage(

{},

make\_table("users",

make\_column("id", &User::id),

make\_column("first\_name", &User::firstName),

make\_column("last\_name", &User::lastName),

primary\_key(&User::id, &User::firstName)),

make\_table("visits",

make\_column("user\_id", &UserVisit::userId),

make\_column("user\_first\_name", &UserVisit::userFirstName),

make\_column("time", &UserVisit::time),

foreign\_key(&UserVisit::userId, &UserVisit::userFirstName).references(&User::id, &User::firstName)));

storage.sync\_schema();

storage.replace(User{

1,

"Bebe",

"Rexha",

});

auto bebeRexha = storage.get<User>(1, "Bebe");

cout << "bebeRexha = " << storage.dump(bebeRexha) << endl;

auto bebeRexhaMaybe = storage.get\_pointer<User>(1, "Bebe");

try {

// 2 and 'Drake' values will be ignored cause they are primary keys

storage.insert(User{2, "Drake", "Singer"});

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

cout << "exception = " << e.what() << endl;

}

storage.replace(User{2, "The Weeknd", "Singer"});

auto weeknd = storage.get<User>(2, "The Weeknd");

cout << "weeknd = " << storage.dump(weeknd) << endl;

/////////////////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\core\_functions.cpp:

last\_insert\_rowid(), length(), abs(), lower(), upper(), changes(), total\_changes(), storage.changes(), char\_(,,,), ||, trim(), random(), ltrim(), rtrim(), coalesce(), substr(), hex(), quote(), randomblob(), instr(), replace(), round(), Unicode(), typeof\_(), ifnull()

Prepared statements:

Use get<0>(statement) to access or modify question marks

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\cross\_join.cpp

// SELECT rank, suit

// FROM ranks

// CROSS JOIN suits

// ORDER BY suit;

auto cards = storage.select(columns(&Rank::rank, &Suit::suit),

cross\_join<Suit>(),

order\_by(&Suit::suit)); // cards is vector<tuple<string, string>>

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\custom\_aliases.cpp:

// SELECT COMPANY.ID, COMPANY.NAME, COMPANY.AGE, DEPARTMENT.DEPT

// FROM COMPANY, DEPARTMENT

// WHERE COMPANY.ID = DEPARTMENT.EMP\_ID;

auto simpleRows = storage.select(columns(&Employee::id, &Employee::name, &Employee::age, &Department::dept),

where(is\_equal(&Employee::id, &Department::empId)));

///////////////////

// SELECT C.ID, C.NAME, C.AGE, D.DEPT

// FROM COMPANY AS C, DEPARTMENT AS D

// WHERE C.ID = D.EMP\_ID;

using als\_c = alias\_c<Employee>;

using als\_d = alias\_d<Department>;

auto rowsWithTableAliases =

storage.select(columns(alias\_column<als\_c>(&Employee::id),

alias\_column<als\_c>(&Employee::name),

alias\_column<als\_c>(&Employee::age),

alias\_column<als\_d>(&Department::dept)),

where(is\_equal(alias\_column<als\_c>(&Employee::id), alias\_column<als\_d>(&Department::empId))));

///////////////

// SELECT COMPANY.ID as COMPANY\_ID, COMPANY.NAME AS COMPANY\_NAME, COMPANY.AGE, DEPARTMENT.DEPT

// FROM COMPANY, DEPARTMENT

// WHERE COMPANY\_ID = DEPARTMENT.EMP\_ID;

auto rowsWithColumnAliases = storage.select(columns(as<EmployeeIdAlias>(&Employee::id),

as<CompanyNameAlias>(&Employee::name),

&Employee::age,

&Department::dept),

where(is\_equal(get<EmployeeIdAlias>(), &Department::empId)));

////////////////////

// SELECT C.ID AS COMPANY\_ID, C.NAME AS COMPANY\_NAME, C.AGE, D.DEPT

// FROM COMPANY AS C, DEPARTMENT AS D

// WHERE C.ID = D.EMP\_ID;

auto rowsWithBothTableAndColumnAliases =

storage.select(columns(as<EmployeeIdAlias>(alias\_column<als\_c>(&Employee::id)),

as<CompanyNameAlias>(alias\_column<als\_c>(&Employee::name)),

alias\_column<als\_c>(&Employee::age),

alias\_column<als\_d>(&Department::dept)),

where(is\_equal(alias\_column<als\_c>(&Employee::id), alias\_column<als\_d>(&Department::empId))));

//////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\date\_time.cpp:

date(), datetime(), julianday(), time(), strftime()

// SELECT 'NAME' FROM 'COMPANY'

auto pureNames = storage.select(&Employee::name);

// SELECT DISTINCT 'NAME' FROM 'COMPANY'

auto distinctNames = storage.select(distinct(&Employee::name));

// SELECT DISTINCT 'ADDRESS', 'NAME' FROM 'COMPANY'

auto severalColumns = storage.select(distinct(columns(&Employee::address, &Employee::name)));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\distinct.cpp

// SELECT 'NAME' FROM 'COMPANY'

auto pureNames = storage.select(&Employee::name);

// SELECT DISTINCT 'NAME' FROM 'COMPANY'

auto distinctNames = storage.select(distinct(&Employee::name));

// SELECT DISTINCT 'ADDRESS', 'NAME' FROM 'COMPANY'

auto severalColumns = storage.select(distinct(columns(&Employee::address, &Employee::name)));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\enum\_binding.cpp

We can bind any enum to our sqlite db.

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\except\_intersection.cpp:

// SELECT dept\_id

// FROM dept\_master

// EXCEPT

// SELECT dept\_id

// FROM emp\_master

auto rows = storage.select(except(select(&DeptMaster::deptId), select(&EmpMaster::deptId)));

// SELECT dept\_id

// FROM dept\_master

// INTERSECT

// SELECT dept\_id

// FROM emp\_master

auto rows = storage.select(intersect(select(&DeptMaster::deptId), select(&EmpMaster::deptId)));

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

for(auto id: rows) {

cout << id << endl;

}

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\exists.cpp:

// SELECT agent\_code,agent\_name,working\_area,commission

// FROM agents

// WHERE exists

// (SELECT \*

// FROM customer

// WHERE grade=3 AND agents.agent\_code=customer.agent\_code)

// ORDER BY commission;

auto rows = storage.select(columns(&Agent::code, &Agent::name, &Agent::workingArea, &Agent::comission),

where(exists(select(asterisk<Customer>(),

where(is\_equal(&Customer::grade, 3) and

is\_equal(&Agent::code, &Customer::agentCode))))),

order\_by(&Agent::comission));

/////////////////

// SELECT cust\_code, cust\_name, cust\_city, grade

// FROM customer

// WHERE grade=2 AND EXISTS

// (SELECT COUNT(\*)

// FROM customer

// WHERE grade=2

// GROUP BY grade

// HAVING COUNT(\*)>2);

auto rows =

storage.select(columns(&Customer::code, &Customer::name, &Customer::city, &Customer::grade),

where(is\_equal(&Customer::grade, 2) and exists(select(count<Customer>(),

where(is\_equal(&Customer::grade, 2)),

group\_by(&Customer::grade),

having(greater\_than(count(), 2))))));

//////////////////////////

// SELECT agent\_code,ord\_num,ord\_amount,cust\_code

// FROM orders

// WHERE NOT EXISTS

// (SELECT agent\_code

// FROM customer

// WHERE payment\_amt=1400);

auto rows = storage.select(

columns(&Order::agentCode, &Order::num, &Order::amount, &Order::custCode),

where(not exists(select(&Customer::agentCode, where(is\_equal(&Customer::paymentAmt, 1400))))));

///////////////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\foreign\_key.cpp:

// simple case with foreign key to a single column without actions

auto storage = make\_storage("foreign\_key.sqlite",

make\_table("artist",

make\_column("artistid", &Artist::artistId, primary\_key()),

make\_column("artistname", &Artist::artistName)),

make\_table("track",

make\_column("trackid", &Track::trackId, primary\_key()),

make\_column("trackname", &Track::trackName),

make\_column("trackartist", &Track::trackArtist),

foreign\_key(&Track::trackArtist).references(&Artist::artistId)));

try {

// This fails because value inserted into the trackartist column (3)

// does not correspond to row in the artist table.

storage.replace(Track{ 14, "Mr. Bojangles", 3 });

assert(0);

}

catch (const std::system\_error& e) {

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

}

{ // case with ON UPDATE CASCADE

auto storage = make\_storage(

"foreign\_key2.sqlite",

make\_table("artist",

make\_column("artistid", &Artist::artistId, primary\_key()),

make\_column("artistname", &Artist::artistName)),

make\_table("track",

make\_column("trackid", &Track::trackId, primary\_key()),

make\_column("trackname", &Track::trackName),

make\_column("trackartist", &Track::trackArtist),

foreign\_key(&Track::trackArtist).references(&Artist::artistId).on\_update.cascade()));

// Update the artistid column of the artist record for "Dean Martin".

// Normally, this would raise a constraint, as it would orphan the two

// dependent records in the track table. However, the ON UPDATE CASCADE clause

// attached to the foreign key definition causes the update to "cascade"

// to the child table, preventing the foreign key constraint violation.

// UPDATE artist SET artistid = 100 WHERE artistname = 'Dean Martin';

storage.update\_all(set(c(&Artist::artistId) = 100), where(c(&Artist::artistName) == "Dean Martin"));

// Deleting the row from the parent table causes the child key

// value of the dependent row to be set to integer value 0. However, this

// value does not correspond to any row in the parent table. Therefore

// the foreign key constraint is violated and an is exception thrown.

// DELETE FROM artist WHERE artistname = 'Sammy Davis Jr.';

try {

storage.remove\_all<Artist>(where(c(&Artist::artistName) == "Sammy Davis Jr."));

assert(0);

}

// case with ON DELETE SET DEFAULT

auto storage = make\_storage(

"foreign\_key3.sqlite",

make\_table("artist",

make\_column("artistid", &Artist::artistId, primary\_key()),

make\_column("artistname", &Artist::artistName)),

make\_table("track",

make\_column("trackid", &Track::trackId, primary\_key()),

make\_column("trackname", &Track::trackName),

make\_column("trackartist", &Track::trackArtist, default\_value(0)),

foreign\_key(&Track::trackArtist).references(&Artist::artistId).on\_delete.set\_default()));

// This time, the value 0 does correspond to a parent table row. And

// so the DELETE statement does not violate the foreign key constraint

// and no exception is thrown.

// INSERT INTO artist VALUES(0, 'Unknown Artist');

// DELETE FROM artist WHERE artistname = 'Sammy Davis Jr.'

storage.replace(Artist{ 0, "Unknown Artist" });

storage.remove\_all<Artist>(where(c(&Artist::artistName) == "Sammy Davis Jr."));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\generated\_column.cpp**:**

auto storage = make\_storage({},

make\_table("products",

make\_column("id", &Product::id, primary\_key()),

make\_column("name", &Product::name),

make\_column("quantity", &Product::quantity),

make\_column("price", &Product::price),

make\_column("total\_value",

&Product::totalValue,

generated\_always\_as(&Product::price \* c(&Product::quantity)))));

// UPDATE products

// SET quantity = 5 WHERE id = 1;

storage.update\_all(set(c(&Product::quantity) = 5), where(c(&Product::id) == 1));

///////////////

**C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\group\_by.cpp:**

// If you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

// SELECT NAME, SUM(SALARY)

// FROM COMPANY

// GROUP BY NAME;

auto salaryName = storage.select(columns(&Employee::name, sum(&Employee::salary)), group\_by(&Employee::name));

salaryName = storage.select(columns(&Employee::name, sum(&Employee::salary)),

group\_by(&Employee::name),

order\_by(&Employee::name));

salaryName = storage.select(columns(&Employee::name, sum(&Employee::salary)),

group\_by(&Employee::name),

order\_by(&Employee::name).desc());

///////////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\having.cpp:

// SELECT \*

// FROM COMPANY

// GROUP BY name

// HAVING count(name) < 2;

auto rows =

storage.get\_all<Employee>(group\_by(&Employee::name), having(lesser\_than(count(&Employee::name), 2)));

// SELECT \*

// FROM COMPANY

// GROUP BY name

// HAVING count(name) > 2;

auto rows =

storage.get\_all<Employee>(group\_by(&Employee::name), having(greater\_than(count(&Employee::name), 2)));

//////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\in\_memory.cpp:

// transaction also work in memory..

storage.transaction([&] {

storage.insert(RapArtist{-1, "Kanye West"});

return false;

});

//////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\index.cpp:

// beware - put `make\_index` before `make\_table` cause `sync\_schema` is called in reverse order

// otherwise you'll receive an exception

auto storage = make\_storage(

"index.sqlite",

make\_index("idx\_contacts\_name", &Contract::firstName, &Contract::lastName, where(length(&Contract::firstName) > 2)),

make\_unique\_index("idx\_contacts\_email", indexed\_column(&Contract::email).collate("BINARY").desc()),

make\_table("contacts",

make\_column("first\_name", &Contract::firstName),

make\_column("last\_name", &Contract::lastName),

make\_column("email", &Contract::email)));

storage.insert(Contract{

"John",

"Doe",

"john.doe@sqlitetutorial.net",

});

try {

storage.insert(Contract{

"Johny",

"Doe",

"john.doe@sqlitetutorial.net",

});

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

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

}

storage.drop\_index("idx\_contacts\_name");

storage.drop\_index("idx\_contacts\_email");

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\insert.cpp:

struct Employee {

int id;

std::string name;

int age;

std::string address;

double salary;

};

struct DetailedEmployee : public Employee {

std::string birthDate;

};

auto storage = make\_storage("insert.sqlite",

make\_table("COMPANY",

make\_column("ID", &Employee::id, primary\_key()),

make\_column("NAME", &Employee::name),

make\_column("AGE", &Employee::age),

make\_column("ADDRESS", &Employee::address),

make\_column("SALARY", &Employee::salary)));

DetailedEmployee teddy;

teddy.name = "Teddy";

teddy.age = 23;

teddy.address = "Norway";

teddy.salary = 20000.00;

// to insert subclass object as a superclass you have to specify type explicitly

teddy.id = storage.insert<Employee>(teddy);

// transaction is optional. It is used here to optimize sqlite usage - every insert opens

// and closes database. So triple insert will open and close the db three times.

// Transaction openes and closes the db only once.

storage.transaction([&] {

for(auto& employee: otherEmployees) {

storage.insert(employee);

}

return true; // commit

});

// INSERT INTO COMPANY(ID, NAME, AGE, ADDRESS, SALARY)

// VALUES (3, 'Sofia', 26, 'Madrid', 15000.0)

// (4, 'Doja', 26, 'LA', 25000.0)

// ON CONFLICT(ID) DO UPDATE SET NAME = excluded.NAME,

// AGE = excluded.AGE,

// ADDRESS = excluded.ADDRESS,

// SALARY = excluded.SALARY

storage.insert(

into<Employee>(),

columns(&Employee::id, &Employee::name, &Employee::age, &Employee::address, &Employee::salary),

values(std::make\_tuple(3, "Sofia", 26, "Madrid", 15000.0), std::make\_tuple(4, "Doja", 26, "LA", 25000.0)),

on\_conflict(&Employee::id)

.do\_update(set(c(&Employee::name) = excluded(&Employee::name),

c(&Employee::age) = excluded(&Employee::age),

c(&Employee::address) = excluded(&Employee::address),

c(&Employee::salary) = excluded(&Employee::salary))));

///////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\iteration.cpp:

struct MarvelHero {

int id;

std::string name;

std::string abilities;

};

auto storage = make\_storage("iteration.sqlite",

make\_table("marvel",

make\_column("id", &MarvelHero::id, primary\_key()),

make\_column("name", &MarvelHero::name),

make\_column("abilities", &MarvelHero::abilities)));

// one can iterate with custom WHERE conditions..

for(auto& hero: storage.iterate<MarvelHero>(where(c(&MarvelHero::name) == "Thor"))) {

cout << "hero = " << storage.dump(hero) << endl;

}

std::vector<MarvelHero> heroesByAlgorithm;

heroesByAlgorithm.reserve(static\_cast<size\_t>(storage.count<MarvelHero>()));

{

auto view = storage.iterate<MarvelHero>();

std::copy(view.begin(), view.end(), std::back\_inserter(heroesByAlgorithm));

}

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\key\_value.cpp:

struct KeyValue {

std::string key;

std::string value;

};

auto& getStorage() {

using namespace sqlite\_orm;

static auto storage = make\_storage("key\_value\_example.sqlite",

make\_table("key\_value",

make\_column("key", &KeyValue::key, primary\_key()),

make\_column("value", &KeyValue::value)));

return storage;

}

void setValue(const std::string& key, const std::string& value) {

using namespace sqlite\_orm;

KeyValue kv{key, value};

getStorage().replace(kv);

}

std::string getValue(const std::string& key) {

using namespace sqlite\_orm;

if(auto kv = getStorage().get\_pointer<KeyValue>(key)) {

return kv->value;

} else {

return {};

}

}

int storedKeysCount() {

return getStorage().count<KeyValue>();

}

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\left\_and\_inner\_join.cpp:

struct Artist {

std::unique\_ptr<int> artistId;

std::unique\_ptr<std::string> name;

};

struct Album {

std::unique\_ptr<int> albumId;

std::unique\_ptr<std::string> title;

std::unique\_ptr<int> artistId;

};

struct Track {

int trackId;

std::string name;

std::unique\_ptr<int> albumId;

int mediaTypeId;

std::unique\_ptr<int> genreId;

std::unique\_ptr<std::string> composer;

long milliseconds;

std::unique\_ptr<long> bytes;

double unitPrice;

};

inline auto initStorage(const std::string& path) {

using namespace sqlite\_orm;

return make\_storage(path,

make\_table("artists",

make\_column("ArtistId", &Artist::artistId, primary\_key()),

make\_column("Name", &Artist::name)),

make\_table("albums",

make\_column("AlbumId", &Album::albumId, primary\_key()),

make\_column("Title", &Album::title),

make\_column("ArtistId", &Album::artistId)),

make\_table("tracks",

make\_column("TrackId", &Track::trackId, primary\_key()),

make\_column("Name", &Track::name),

make\_column("AlbumId", &Track::albumId),

make\_column("MediaTypeId", &Track::mediaTypeId),

make\_column("GenreId", &Track::genreId),

make\_column("Composer", &Track::composer),

make\_column("Milliseconds", &Track::milliseconds),

make\_column("Bytes", &Track::bytes),

make\_column("UnitPrice", &Track::unitPrice)));

}

// SELECT

// artists.ArtistId,

// albumId

// FROM

// artists

// LEFT JOIN albums ON albums.artistid = artists.artistid

// ORDER BY

// albumid;

auto rows = storage.select(columns(&Artist::artistId, &Album::albumId),

left\_join<Album>(on(c(&Album::artistId) == &Artist::artistId)),

order\_by(&Album::albumId));

////////////////////////

// SELECT

// artists.ArtistId,

// albumId

// FROM

// artists

// LEFT JOIN albums ON albums.artistid = artists.artistid

// WHERE

// albumid IS NULL;

rows = storage.select(columns(&Artist::artistId, &Album::albumId),

left\_join<Album>(on(c(&Album::artistId) == &Artist::artistId)),

where(is\_null(&Album::albumId)));

// SELECT

// trackid,

// name,

// title

// FROM

// tracks

// INNER JOIN albums ON albums.albumid = tracks.albumid;

auto innerJoinRows0 = storage.select(columns(&Track::trackId, &Track::name, &Album::title),

inner\_join<Album>(on(c(&Track::albumId) == &Album::albumId)));

// SELECT

// trackid,

// tracks.name AS Track,

// albums.title AS Album,

// artists.name AS Artist

// FROM

// tracks

// INNER JOIN albums ON albums.albumid = tracks.albumid

// INNER JOIN artists ON artists.artistid = albums.artistid;

auto innerJoinRows2 = storage.select(columns(&Track::trackId, &Track::name, &Album::title, &Artist::name),

inner\_join<Album>(on(c(&Album::albumId) == &Track::albumId)),

inner\_join<Artist>(on(c(&Artist::artistId) == &Album::artistId)));

// SELECT a.doctor\_id,a.doctor\_name,

// c.patient\_name,c.vdate

// FROM doctors a

// LEFT JOIN visits c

// ON a.doctor\_id=c.doctor\_id;

auto rows = storage2.select(columns(&Doctor::id, &Doctor::name, &Visit::patientName, &Visit::vdate),

left\_join<Visit>(on(c(&Doctor::id) == &Visit::doctorId)));

// SELECT doctor\_id,doctor\_name,

// patient\_name,vdate

// FROM doctors

// LEFT JOIN visits

// USING(doctor\_id);

rows = storage2.select(columns(&Doctor::id, &Doctor::name, &Visit::patientName, &Visit::vdate),

left\_join<Visit>(using\_(&Visit::doctorId))); // or using\_(&Doctor::id)

/////////////\

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\multi\_table\_select.cpp:

// SELECT ReqEquip.ReqNumber, ReqEquip.Requestor, ReqDetail.Quantity, ReqDetail.StockNumber

// FROM ReqEquip, ReqDetail

// WHERE ReqEquip.ReqNumber = ReqDetail.ReqNumber;

auto rows = storage.select(

columns(&ReqEquip::reqNumber, &ReqEquip::requestor, &ReqDetail::quantity, &ReqDetail::stockNumber),

where(c(&ReqEquip::reqNumber) == &ReqDetail::reqNumber));

// SELECT ReqEquip.ReqNumber, ReqEquip.Requestor, ReqDetail.Quantity, ReqDetail.StockNumber

// FROM ReqEquip, ReqDetail

// WHERE ReqEquip.ReqNumber = ReqDetail.ReqNumber AND ReqEquip.ReqNumber = 1000;

rows = storage.select(

columns(&ReqEquip::reqNumber, &ReqEquip::requestor, &ReqDetail::quantity, &ReqDetail::stockNumber),

where(c(&ReqEquip::reqNumber) == &ReqDetail::reqNumber and c(&ReqEquip::reqNumber) == 1000));

/////////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\natural\_join.cpp:

// SELECT doctor\_id,doctor\_name,degree,patient\_name,vdate

// FROM doctors

// NATURAL JOIN visits

// WHERE doctors.degree="MD";

auto rows = storage.select(

columns(&Doctor::doctor\_id, &Doctor::doctor\_name, &Doctor::degree, &Visit::patient\_name, &Visit::vdate),

natural\_join<Visit>(),

where(c(&Doctor::degree) == "MD"));

///////////////

// SELECT doctor\_id,doctor\_name,degree,spl\_descrip,patient\_name,vdate

// FROM doctors

// NATURAL JOIN speciality

// NATURAL JOIN visits

// WHERE doctors.degree='MD';

auto rows = storage.select(columns(&Doctor::doctor\_id,

&Doctor::doctor\_name,

&Doctor::degree,

&Speciality::spl\_descrip,

&Visit::patient\_name,

&Visit::vdate),

natural\_join<Speciality>(),

natural\_join<Visit>(),

where(c(&Doctor::degree) == "MD"));

/////////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\nullable\_enum\_binding.cpp:

Interesting!

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\prepared\_statement.cpp:

Ahmmmm

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\private\_class\_members.cpp:

class Player {

int id = 0;

std::string name;

public:

Player() {}

Player(std::string name\_) : name(std::move(name\_)) {}

Player(int id\_, std::string name\_) : id(id\_), name(std::move(name\_)) {}

std::string getName() const {

return this->name;

}

void setName(std::string name) {

this->name = std::move(name);

}

int getId() const {

return this->id;

}

void setId(int id) {

this->id = id;

}

};

auto storage = make\_storage(

"private.sqlite",

make\_table(

"players",

make\_column("id",

&Player::setId, // setter

&Player::getId, // getter

primary\_key()),

make\_column("name",

&Player::getName, // BTW order doesn't matter: setter can be placed before getter or opposite.

&Player::setName)));

auto soloId = storage.insert(Player("Solo"));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\select.cpp:

// select all employees..

auto allEmployees = storage.get\_all<Employee>();

// now let's select id, name and salary..

auto idsNamesSalarys = storage.select(columns(&Employee::id, &Employee::name, &Employee::salary));

auto allEmployeesTuples = storage.select(asterisk<Employee>());

//////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\self\_join.cpp:

// SELECT m.FirstName || ' ' || m.LastName,

// employees.FirstName || ' ' || employees.LastName

// FROM employees

// INNER JOIN employees m

// ON m.ReportsTo = employees.EmployeeId

using als = alias\_m<Employee>;

auto firstNames = storage.select(

columns(c(alias\_column<als>(&Employee::firstName)) || " " || c(alias\_column<als>(&Employee::lastName)),

c(&Employee::firstName) || " " || c(&Employee::lastName)),

inner\_join<als>(on(alias\_column<als>(&Employee::reportsTo) == c(&Employee::employeeId))));

// SELECT emp.FirstName || ' ' || emp.LastName,

// employees.FirstName || ' ' || employees.LastName

// FROM employees

// INNER JOIN employees emp

// ON emp.ReportsTo = employees.EmployeeId

using als = custom\_alias<Employee>;

auto firstNames = storage.select(

columns(c(alias\_column<als>(&Employee::firstName)) || " " || c(alias\_column<als>(&Employee::lastName)),

c(&Employee::firstName) || " " || c(&Employee::lastName)),

inner\_join<als>(on(alias\_column<als>(&Employee::reportsTo) == c(&Employee::employeeId))));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\subentities.cpp:

class Mark {

public:

int value;

int student\_id;

};

class Student {

public:

int id;

std::string name;

int roll\_number;

std::vector<decltype(Mark::value)> marks;

};

auto storage =

make\_storage("subentities.sqlite",

make\_table("students",

make\_column("id", &Student::id, primary\_key()),

make\_column("name", &Student::name),

make\_column("roll\_no", &Student::roll\_number)),

make\_table("marks", make\_column("mark", &Mark::value), make\_column("student\_id", &Mark::student\_id)));

// inserts or updates student and does the same with marks

int addStudent(const Student& student) {

auto studentId = student.id;

if(storage.count<Student>(where(c(&Student::id) == student.id))) {

storage.update(student);

} else {

studentId = storage.insert(student);

}

// insert all marks within a transaction

storage.transaction([&] {

storage.remove\_all<Mark>(where(c(&Mark::student\_id) == studentId));

for(auto& mark: student.marks) {

storage.insert(Mark{mark, studentId});

}

return true;

});

return studentId;

}

/\*\*

\* To get student from db we have to execute two queries:

\* `SELECT \* FROM students WHERE id = ?`

\* `SELECT mark FROM marks WHERE student\_id = ?`

\*/

Student getStudent(int studentId) {

auto res = storage.get<Student>(studentId);

res.marks = storage.select(&Mark::value, where(c(&Mark::student\_id) == studentId));

return res; // must be moved automatically by compiler

}

storage.sync\_schema(); // create tables if they don't exist

Student mike{-1, "Mike", 123, {}}; // create student named `Mike` without marks and without id

mike.marks = {3, 4, 5};

mike.id = addStudent(mike);

mikeId = mike.id;

// also let's create another students with marks..

Student anna{-1, "Anna", 555, {}};

anna.marks.push\_back(6);

anna.marks.push\_back(7);

anna.id = addStudent(anna);

annaId = anna.id;

auto mike = getStudent(mikeId);

cout << "mike = " << storage.dump(mike) << endl;

cout << "mike.marks = ";

for(auto& m: mike.marks) {

cout << m << " ";

}

cout << endl;

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\subquery.cpp:

// SELECT first\_name, last\_name, salary

// FROM employees

// WHERE salary >(

// SELECT salary

// FROM employees

// WHERE first\_name='Alexander');

auto rows = storage.select(

columns(&Employee::firstName, &Employee::lastName, &Employee::salary),

where(greater\_than(&Employee::salary,

select(&Employee::salary, where(is\_equal(&Employee::firstName, "Alexander"))))));

// SELECT employee\_id,first\_name,last\_name,salary

// FROM employees

// WHERE salary > (SELECT AVG(SALARY) FROM employees);

auto rows = storage.select(columns(&Employee::id, &Employee::firstName, &Employee::lastName, &Employee::salary),

where(greater\_than(&Employee::salary, select(avg(&Employee::salary)))));

//////////////

// SELECT first\_name, last\_name, department\_id

// FROM employees

// WHERE department\_id IN

// (SELECT DEPARTMENT\_ID FROM departments

// WHERE location\_id=1700);

auto rows = storage.select(

columns(&Employee::firstName, &Employee::lastName, &Employee::departmentId),

where(in(&Employee::departmentId, select(&Department::id, where(c(&Department::locationId) == 1700)))));

//////////////

// SELECT first\_name, last\_name, department\_id

// FROM employees

// WHERE department\_id NOT IN

// (SELECT DEPARTMENT\_ID FROM departments

// WHERE manager\_id

// BETWEEN 100 AND 200);

auto rows =

storage.select(columns(&Employee::firstName, &Employee::lastName, &Employee::departmentId),

where(not\_in(&Employee::departmentId,

select(&Department::id, where(between(&Department::managerId, 100, 200))))));

///////////////

// SELECT last\_name, salary, department\_id

// FROM employees e

// WHERE salary >(SELECT AVG(salary)

// FROM employees

// WHERE department\_id = e.department\_id);

using als = alias\_e<Employee>;

auto rows = storage.select(

columns(alias\_column<als>(&Employee::lastName),

alias\_column<als>(&Employee::salary),

alias\_column<als>(&Employee::departmentId)),

where(greater\_than(

alias\_column<als>(&Employee::salary),

select(avg(&Employee::salary),

where(is\_equal(&Employee::departmentId, alias\_column<als>(&Employee::departmentId)))))));

////////////////

// SELECT first\_name, last\_name, employee\_id, job\_id

// FROM employees

// WHERE 1 <=

// (SELECT COUNT(\*) FROM Job\_history

// WHERE employee\_id = employees.employee\_id);

auto rows =

storage.select(columns(&Employee::firstName, &Employee::lastName, &Employee::id, &Employee::jobId),

where(lesser\_or\_equal(

1,

select(count<JobHistory>(), where(is\_equal(&Employee::id, &JobHistory::employeeId))))));

////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\synchronous.cpp:

storage.sync\_schema();

storage.pragma.synchronous(0);

storage.remove\_all<Query>();

/////////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\triggers.cpp:

Before Insert:

// CREATE TRIGGER validate\_email\_before\_insert\_leads

// BEFORE INSERT ON leads

// BEGIN

// SELECT

// CASE

// WHEN NEW.email NOT LIKE '%\_@\_\_%.\_\_%' THEN

// RAISE (ABORT,'Invalid email address')

// END;

// END;

make\_trigger("validate\_email\_before\_insert\_leads",

before()

.insert()

.on<Lead>()

.begin(select(case\_<int>()

.when(not like(new\_(&Lead::email), "%\_@\_\_%.\_\_%"),

then(raise\_abort("Invalid email address")))

.end()))

.end())

After update:

// CREATE TRIGGER log\_contact\_after\_update

// AFTER UPDATE ON leads

// WHEN old.phone <> new.phone

// OR old.email <> new.email

// BEGIN

// INSERT INTO lead\_logs (

// old\_id,

// new\_id,

// old\_phone,

// new\_phone,

// old\_email,

// new\_email,

// user\_action,

// created\_at

// )

// VALUES

// (

// old.id,

// new.id,

// old.phone,

// new.phone,

// old.email,

// new.email,

// 'UPDATE',

// DATETIME('NOW')

// ) ;

// END;

make\_trigger("log\_contact\_after\_update",

after()

.update()

.on<Lead>()

.when(is\_not\_equal(old(&Lead::phone), new\_(&Lead::phone)) and

is\_not\_equal(old(&Lead::email), new\_(&Lead::email)))

.begin(insert(into<LeadLog>(),

columns(&LeadLog::oldId,

&LeadLog::newId,

&LeadLog::oldPhone,

&LeadLog::newPhone,

&LeadLog::oldEmail,

&LeadLog::newEmail,

&LeadLog::userAction,

&LeadLog::createdAt),

values(std::make\_tuple(old(&Lead::id),

new\_(&Lead::id),

old(&Lead::phone),

new\_(&Lead::phone),

old(&Lead::email),

new\_(&Lead::email),

"UPDATE",

datetime("NOW")))))

.end())

/////////////////

// Insert a row with an invalid email into the leads table:

//

// INSERT INTO leads (first\_name, last\_name, email, phone)

// VALUES('John', 'Doe', 'jjj', '4089009334');

try {

storage.insert(Lead{0, "John", "Doe", "jjj", "A"}); // "4089009334"

} catch(const std::system\_error &systemError) {

cout << "error: " << systemError.what() << endl;

}

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\union.cpp:

// SELECT EMP\_ID, NAME, DEPT

// FROM COMPANY

// INNER JOIN DEPARTMENT

// ON COMPANY.ID = DEPARTMENT.EMP\_ID

// UNION

// SELECT EMP\_ID, NAME, DEPT

// FROM COMPANY

// LEFT OUTER JOIN DEPARTMENT

// ON COMPANY.ID = DEPARTMENT.EMP\_ID;

auto rows = storage.select(

union\_(select(columns(&Department::employeeId, &Employee::name, &Department::dept),

inner\_join<Department>(on(is\_equal(&Employee::id, &Department::employeeId)))),

select(columns(&Department::employeeId, &Employee::name, &Department::dept),

left\_outer\_join<Department>(on(is\_equal(&Employee::id, &Department::employeeId))))));

/////////////////////

Union all:

// SELECT EMP\_ID, NAME, DEPT

// FROM COMPANY

// INNER JOIN DEPARTMENT

// ON COMPANY.ID = DEPARTMENT.EMP\_ID

// UNION ALL

// SELECT EMP\_ID, NAME, DEPT

// FROM COMPANY

// LEFT OUTER JOIN DEPARTMENT

// ON COMPANY.ID = DEPARTMENT.EMP\_ID;

auto rows = storage.select(

union\_all(select(columns(&Department::employeeId, &Employee::name, &Department::dept),

inner\_join<Department>(on(is\_equal(&Employee::id, &Department::employeeId)))),

select(columns(&Department::employeeId, &Employee::name, &Department::dept),

left\_outer\_join<Department>(on(is\_equal(&Employee::id, &Department::employeeId))))));

/////////

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\unique.cpp:

auto storage = make\_storage("unique.sqlite",

make\_table("unique\_test",

make\_column("id", &Entry::id, autoincrement(), primary\_key()),

make\_column("unique\_text", &Entry::uniqueColumn, unique()),

make\_column("nullable\_text", &Entry::nullableColumn)));

///////////////////////////

auto storage = make\_storage("unique.sqlite",

make\_table("unique\_test",

make\_column("id", &Entry::id, autoincrement(), primary\_key()),

make\_column("unique\_text", &Entry::uniqueColumn, unique()),

make\_column("nullable\_text", &Entry::nullableColumn),

unique(&Entry::id, &Entry::uniqueColumn)));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\update.cpp:

auto employee6 = stor->get<Employee>(6);

employee6.address = "Texas";

stor->update(

employee6); // actually this call updates all non-primary-key columns' values to passed object's fields

// 'UPDATE COMPANY SET ADDRESS = 'Texas', SALARY = 20000.00 WHERE AGE < 30'

using namespace sqlite\_orm;

stor->update\_all(set(c(&Employee::address) = "Texas", c(&Employee::salary) = 20000.00),

where(c(&Employee::age) < 30));

C:\Components\sqlite\_orm\_dev\_10\_feb\_2022\examples\user\_defined\_functions.cpp:

Scalar and aggregate functions