Tests\select\_constraints\_tests.cpp

# TEST\_CASE("select constraints")

## SELECT \* FROM ‘EMPLOYEE’;

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

auto allEmployees = storage.select(object<Employee>());

## SELECT "COMPANY"."NAME" FROM 'COMPANY'

storage.select(&Employee::name);

Employee is mapped to COMPANY table: (is allowed!!)

auto storage = make\_storage({},

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)));

## SELECT DISTINCT("COMPANY"."NAME") FROM 'COMPANY'

storage.select(distinct(&Employee::name));

storage.select(distinct(&Employee::name), from<Employee>());

## SELECT ?

SECTION("as\_optional") {

SECTION("prepared statements bindings")

auto statement = storage.prepare(select(5));

auto statement = storage.prepare(select(as\_optional(5)));

auto rows = storage.execute(statement); // SELECT 5

get<0>(statement) = 10; // pass value 10 to parameter

rows = storage.execute(statement);

## SELECT ("COMPANY"."NAME") FROM 'COMPANY'

SECTION("just names") {

auto rows = storage.select(as\_optional(&Employee::name));

rows[0] std::optional<string>

// JDH

constexpr bool same = std::is\_same<decltype(rows), std::vector<std::optional<std::string>>>::value;

if( rows[0] ) {

auto val = \*rows[0];

std::ignore = val;

}

## SELECT "Author"."id", "Author"."name", "Book"."id", "Book"."title", "Book"."author\_id" FROM 'Author' LEFT JOIN 'Book' ON ("Author"."id" = "Book"."author\_id")

auto rows = storage2.select(columns(&Author::id, &Author::name, &Book::id, &Book::title, &Book::authorId),

left\_join<Book>(on(c(&Author::id) == &Book::authorId)));

using Rows = std::vector<

std::tuple<int, std::string, std::optional<int>, std::optional<std::string>, std::optional<int>>>;

rows = storage2.select(columns(&Author::id,

&Author::name,

as\_optional(&Book::id),

as\_optional(&Book::title),

as\_optional(&Book::authorId)),

left\_join<Book>(on(c(&Author::id) == &Book::authorId)));

## SELECT "users"."id" FROM 'users', 'visits' WHERE (EXISTS (SELECT "visits"."id" FROM 'users', 'visits' WHERE ((("visits"."time" = ?) AND ("visits"."userId" = "users"."id")))))

auto rows = storage.select(

&User::id,

where(exists(select(&Visit::id, where(c(&Visit::time) == 200000 and eq(&Visit::userId, &User::id))))));

# TEST\_CASE("Exists")

## SELECT "users"."id" FROM 'users' JOIN 'visits' 'a' ON ('a'."userId" = "users"."id") WHERE (EXISTS (SELECT "visits"."id" FROM 'visits', 'users' WHERE (((("visits"."time" = ?) AND ("visits"."userId" = "users"."id")) AND ('a'."id" = "visits"."id")))))

auto rows = storage.select(

&User::id,

from<User>(), join<als\_v\_o>(on(is\_equal(alias\_column<als\_v\_o>(&Visit::userId), &User::id))),

where(exists(select(&Visit::id,

from<Visit, User>(),

where(c(&Visit::time) == 200000 and

eq(&Visit::userId, &User::id) and

eq(alias\_column<als\_v\_o>(&Visit::id), &Visit::id))))));

int

## SELECT "users"."id", 'a'."id", 'a'."time" FROM 'users' JOIN 'visits' 'a' ON ('a'."userId" =

## "users"."id") WHERE (EXISTS (SELECT "visits"."id" FROM 'visits', 'users' WHERE (((("visits"."time" = ?) AND ("visits"."userId" = "users"."id")) AND ('a'."id" = "visits"."id")))))

auto rows = storage.select(

columns(&User::id,

alias\_column<als\_v\_o>(&Visit::id),

alias\_column<als\_v\_o>(&Visit::time)),

from<User>(), join<als\_v\_o>(on(is\_equal(alias\_column<als\_v\_o>(&Visit::userId), &User::id))),

where(exists(select(&Visit::id,

from<Visit, User>(),

where(c(&Visit::time) == 200000 and

eq(&Visit::userId, &User::id) and

eq(alias\_column<als\_v\_o>(&Visit::id), &Visit::id))))));

std::tuple<int>

## SELECT 'b'."id", 'a'."id", 'a'."time" FROM 'users' 'b' JOIN 'visits' 'a' ON ('a'."userId" = 'b'."id") WHERE (EXISTS (SELECT "visits"."id" FROM 'visits', 'users' WHERE (((("visits"."time" = ?) AND ("visits"."userId" = "users"."id")) AND ('a'."id" = "visits"."id")))))

auto rows = storage.select(

columns(alias\_column<als\_u\_o>(&User::id),

alias\_column<als\_v\_o>(&Visit::id),

alias\_column<als\_v\_o>(&Visit::time)),

from<als\_u\_o>(), join<als\_v\_o>(on(is\_equal(alias\_column<als\_v\_o>(&Visit::userId), alias\_column<als\_u\_o>(&User::id)))),

where(exists(select(&Visit::id,

from<Visit, User>(),

where(c(&Visit::time) == 200000 and

eq(&Visit::userId, &User::id) and

eq(alias\_column<als\_v\_o>(&Visit::id), &Visit::id))))));

## SELECT 'b'."id", 'a'."id", 'a'."time" FROM 'users' 'b' JOIN 'visits' 'a' ON ('a'."userId" = 'b'."id") WHERE (EXISTS (SELECT 'c'."id" FROM 'visits' 'c', 'users' 'z' WHERE (((('c'."time" = ?) AND ('c'."userId" = 'z'."id")) AND ('a'."id" = 'c'."id")))))

auto rows = storage.select(

columns(alias\_column<als\_u\_o>(&User::id),

alias\_column<als\_v\_o>(&Visit::id),

alias\_column<als\_v\_o>(&Visit::time)),

from<als\_u\_o>(), join<als\_v\_o>(on(is\_equal(alias\_column<als\_v\_o>(&Visit::userId ), alias\_column<als\_u\_o>(&User::id)))),

where(exists(select(alias\_column<als\_v\_i>(&Visit::id),

from<als\_v\_i, als\_u\_i>(),

where(c(alias\_column<als\_v\_i>(&Visit::time)) == 200000 and

eq(alias\_column<als\_v\_i>(&Visit::userId), alias\_column<als\_u\_i>(&User::id)) and

eq(alias\_column<als\_v\_o>(&Visit::id), alias\_column<als\_v\_i>(&Visit::id)))))));

# TEST\_CASE(“Case”):

## SELECT CASE "users"."country" WHEN ? THEN ? ELSE ? END FROM 'users' ORDER BY "users"."last\_name" , "users"."first\_name"

auto rows = storage.select(

columns(case\_<std::string>(&User::country).when("USA", then("Dosmetic")).else\_("Foreign").end()),

multi\_order\_by(order\_by(&User::lastName), order\_by(&User::firstName)));

## SELECT CASE "users"."country" WHEN ? THEN ? ELSE ? END AS Grade FROM 'users' ORDER BY "users"."last\_name" , "users"."first\_name"

struct GradeAlias : alias\_tag {

static const std::string& get() {

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

return res;

}

};

rows = storage.select(

columns(as<GradeAlias>(

case\_<std::string>(&User::country).when("USA", then("Dosmetic")).else\_("Foreign").end())),

multi\_order\_by(order\_by(&User::lastName), order\_by(&User::firstName)));

## SELECT CASE WHEN ("tracks"."milliseconds" < ?) THEN ? WHEN (("tracks"."milliseconds" >= ?) AND ("tracks"."milliseconds" < ?)) THEN ? ELSE ? END FROM 'tracks' ORDER BY "tracks"."name"

auto rows = storage.select(

case\_<std::string>()

.when(c(&Track::milliseconds) < 60000, then("short"))

.when(c(&Track::milliseconds) >= 60000 and c(&Track::milliseconds) < 300000, then("medium"))

.else\_("long")

.end(),

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

## SELECT CASE WHEN ("tracks"."milliseconds" < ?) THEN ? WHEN (("tracks"."milliseconds" >= ?) AND ("tracks"."milliseconds" < ?)) THEN ? ELSE ? END AS Grade FROM 'tracks' ORDER BY "tracks"."name"

rows = storage.select(

as<GradeAlias>(

case\_<std::string>()

.when(c(&Track::milliseconds) < 60000, then("short"))

.when(c(&Track::milliseconds) >= 60000 and c(&Track::milliseconds) < 300000, then("medium"))

.else\_("long")

.end()),

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

# TEST\_CASE("Where")

## SELECT "users"."id", "users"."age", "users"."name" FROM 'users' WHERE (?)

auto users2 = storage.get\_all<User>(where(true));

REQUIRE(users2.size() == 2);

## SELECT "users"."id", "users"."age", "users"."name" FROM 'users' WHERE (((? OR ("users"."id" = ?)) AND (? OR ("users"."age" = ?))))

auto users6 = storage.get\_all<User>(where((false or c(&User::id) == 4) and (false or c(&User::age) == 18)));

REQUIRE(users6.empty());

# TEST\_CASE("collate")

## SELECT "users"."id", "users"."first\_name" FROM 'users' WHERE ("users"."first\_name" = ? COLLATE NOCASE)

User user1{1, "HELLO"};

User user2{2, "Hello"};

User user3{3, "HEllo"};

storage.replace(user1);

storage.replace(user2);

storage.replace(user3);

auto rows = storage.get\_all<User>(where(is\_equal(&User::firstName, "hello").collate\_nocase()));

std::vector<User> expected = {user1, user2, user3};

REQUIRE(rows == expected);

# TEST\_CASE("Dynamic order by")

## SELECT "users"."id", "users"."first\_name", "users"."last\_name", "users"."register\_time" FROM 'users' ORDER BY "first\_name" ASC, "id" DESC

auto orderBy = dynamic\_order\_by(storage);

std::vector<decltype(User::id)> expectedIds;

SECTION("firstName asc + id desc") {

orderBy.push\_back(order\_by(&User::firstName).asc());

orderBy.push\_back(order\_by(&User::id).desc());

expectedIds = {

3,

1,

2,

4,

};

}

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

REQUIRE(rows.size() == 4);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\arithmetic\_operators.cpp

# TEST\_CASE("statement\_serializator arithmetic operators")

internal::serializator\_context\_base context;

std::string value;

decltype(value) expected;

SECTION("add") {

SECTION("func") {

value = serialize(add(3, 5), context);

}

SECTION("operator") {

value = serialize(c(3) + 5, context);

}

expected = "(3 + 5)";

}

Arithmetic functions:

* sub
* add
* mul
* div
* mod

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\arithmetic\_operators.cpp

# TEST\_CASE("Arithmetic operators")

## SELECT ("objects"."name\_len" + ?) FROM 'objects'

auto rows = storage.select(c(&Object::nameLen) + 1000);

REQUIRE(int(rows[0]) == name.length() + 1000);

auto rows = storage.select(columns(c(&Object::nameLen) + 1000));

REQUIRE(int(std::get<0>(rows[0])) == name.length() + 1000);

## SELECT ("objects"."name" || ?) FROM 'objects'

auto rows = storage.select(c(&Object::name) || suffix);

REQUIRE(row == name + suffix);

## SELECT ("objects"."name" || ?) FROM 'objects'

auto rows = storage.select(columns(conc(&Object::name, suffix)));

REQUIRE(std::get<0>(row) == name + suffix);

## SELECT ("objects"."name" || ?), ("objects"."name" || ?), ("objects"."name" || ?), ("objects"."name" || ?), ("objects"."name\_len" + "objects"."number"), ("objects"."name\_len" + "objects"."number"), ("objects"."name\_len" + "objects"."number"), ("objects"."name\_len" + "objects"."number"), ("objects"."name\_len" + ?), ("objects"."name\_len" - "objects"."number"), ("objects"."name\_len" - "objects"."number"), ("objects"."name\_len" - "objects"."number"), ("objects"."name\_len" - "objects"."number"), ("objects"."name\_len" - ?), ("objects"."name\_len" \* "objects"."number"), ("objects"."name\_len" \* "objects"."number"), ("objects"."name\_len" \* "objects"."number"), ("objects"."name\_len" \* "objects"."number"), ("objects"."name\_len" \* ?), ("objects"."name\_len" / "objects"."number"), ("objects"."name\_len" / "objects"."number"), ("objects"."name\_len" / "objects"."number"), ("objects"."name\_len" / "objects"."number"), ("objects"."name\_len" / ?) FROM 'objects'

auto rows = storage.select(columns(conc(&Object::name, suffix),

c(&Object::name) || suffix,

&Object::name || c(suffix),

c(&Object::name) || c(suffix),

add(&Object::nameLen, &Object::number),

c(&Object::nameLen) + &Object::number,

&Object::nameLen + c(&Object::number),

c(&Object::nameLen) + c(&Object::number),

c(&Object::nameLen) + 1000,

sub(&Object::nameLen, &Object::number),

c(&Object::nameLen) - &Object::number,

&Object::nameLen - c(&Object::number),

c(&Object::nameLen) - c(&Object::number),

c(&Object::nameLen) - 1000,

mul(&Object::nameLen, &Object::number),

c(&Object::nameLen) \* &Object::number,

&Object::nameLen \* c(&Object::number),

c(&Object::nameLen) \* c(&Object::number),

c(&Object::nameLen) \* 1000,

div(&Object::nameLen, &Object::number),

c(&Object::nameLen) / &Object::number,

&Object::nameLen / c(&Object::number),

c(&Object::nameLen) / c(&Object::number),

c(&Object::nameLen) / 2));

REQUIRE(std::get<1>(rows[0]) == std::get<0>(row[0]));

///

## SELECT ("objects"."name\_len" % "objects"."number"), ("objects"."name\_len" % "objects"."number"), ("objects"."name\_len" % "objects"."number"), ("objects"."name\_len" % "objects"."number"), ("objects"."name\_len" % ?) FROM 'objects'

auto rows = storage.select(columns(mod(&Object::nameLen, &Object::number),

c(&Object::nameLen) % &Object::number,

&Object::nameLen % c(&Object::number),

c(&Object::nameLen) % c(&Object::number),

c(&Object::nameLen) % 5));

std::tuple<double,double,double,double,double>

REQUIRE(std::get<1>(row) == std::get<0>(row));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\backup\_tests.cpp:

# BACKUP

storage.backup\_to(backupFilename);

storage.backup\_to(storage2);

storage2.backup\_from(backupFilename);

storage2.backup\_from(storage);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\binary\_operators.cpp:

# TEST\_CASE("binary operators")

## is\_equal, eq, ==

## SELECT ("users"."id" = ?) FROM 'users'

Std::vector<bool>

rows = storage.select(is\_equal(&User::id, 1));

rows = storage.select(eq(&User::id, 1));

rows = storage.select(c(&User::id) == 1);

rows = storage.select(&User::id == c(1));

rows = storage.select(column<User>(&User::id) == 1);

## is\_not\_equal, NE, !=

Idem is\_equal …

## greater\_than, gt, >

Idem is\_equal

## greater\_or\_equal, ge, >=

Idem is\_equal

## lesser\_than, lt, <

Idem is\_equal

## lesser\_or\_equal. Le, <=

Idem is\_equal

## And, or

rows = storage.select(greater\_than(&User::id, 1) and lesser\_than(&User::id, 3));

rows = storage.select(lesser\_than(&User::id, 2) or greater\_than(&User::id, 2));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\bitwise.cpp:

# TEST\_CASE("bitwise operators")

auto rows = storage.select(bitwise\_or(60, 13));

REQUIRE(rows == std::vector<int>{61});

auto rows = storage.select(bitwise\_and(60, 13));

REQUIRE(rows == std::vector<int>{12});

auto rows = storage.select(bitwise\_shift\_left(60, 2));

REQUIRE(rows == std::vector<int>{240});

auto rows = storage.select(bitwise\_shift\_right(60, 2));

REQUIRE(rows == std::vector<int>{15});

auto rows = storage.select(bitwise\_not(60));

REQUIRE(rows == std::vector<int>{-61});

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\cast.cpp:

# TEST\_CASE("Cast")

auto rows = storage.select(columns(cast<int>(&Student::scoreFloat), cast<int>(&Student::scoreString)));

std::tuple<int,int>

auto rows = storage.select(cast<std::string>(5));

std::string

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\constraints\check.cpp:

# TEST\_CASE("check")

auto storage = make\_storage({},

make\_table("BOOK",

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

make\_column("Book\_name", &Book::name),

make\_column("Pub\_name", &Book::pubName),

make\_column("PRICE", &Book::price, check(c(&Book::price) > 0))));

auto storage = make\_storage({},

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),

check(like(&Contact::email, "a%") && true)); // like throws exception if && true is missing at sync\_schema!

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\collate.cpp:

# TEST\_CASE("statement\_serializator collate")

auto col = collate\_nocase();

auto col = collate\_binary();

auto col = collate\_rtrim();

auto storage = make\_storage("",

make\_table("objects",

make\_column("name", &Object::name, collate\_nocase()),

make\_column("name\_len", &Object::nameLen, generated\_always\_as(length(&Object::name)+offset))[[1]](#footnote-1),

make\_column("number", &Object::number)));

storage.sync\_schema();

auto rows = storage.get\_all<Object>(where(c(&Object::name) == "zombie"));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\constraints\composite\_key.cpp:

# TEST\_CASE("Composite key")

struct Record {

int year = 0;

int month = 0;

int amount = 0;

};

auto recordsTableName = "records";

auto storage = make\_storage({},

make\_table(recordsTableName,

make\_column("year", &Record::year),

make\_column("month", &Record::month),

make\_column("amount", &Record::amount),

primary\_key(&Record::year, &Record::month)));

storage.sync\_schema();

storage.replace(Record{1, 2, 3});

REQUIRE(storage.count<Record>() == 1);

auto record = storage.get<Record>(1, 2);

REQUIRE(record.year == 1);

REQUIRE(record.month == 2);

REQUIRE(record.amount == 3);

auto recordPointer = storage.get\_pointer<Record>(1, 2);

REQUIRE(recordPointer);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\core\_functions.cpp:

# TEST\_CASE("statement\_serializator core functions")[[2]](#footnote-2)

length, abs, lower, upper, total\_changes, changes, trim, ltrim, rtrim, hex, quote, randomblob, instr, replace, round, char\_, random, coalesce, date, time, datetime, julianday, strftime, zeroblob, substr, soundex

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\built\_in\_functions\_tests\core\_functions\_tests.cpp:[[3]](#footnote-3)

# TEST\_CASE("substr")

# TEST\_CASE("replace func")

storage.update\_all(set(c(&Contact::phone) = replace(&Contact::phone, "410", "+1-410")));

# TEST\_CASE("coalesce")

storage.select(coalesce<std::optional<double>>(&Foo::field, nullptr));

storage.select(coalesce<std::optional<double>>(&Foo::field, std::nullopt));

# TEST\_CASE(“IFNULL”)

auto rows = storage.select(columns(&Customer::firstName,

&Customer::lastName,

ifnull<std::string>(&Customer::fax, "Call:" || c(&Customer::phone))),

order\_by(&Customer::firstName));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\built\_in\_functions\_tests\datetime\_function\_tests.cpp:

# TEST\_CASE("time")

auto rows = storage.select(time("12:00", "localtime"));

auto rows = storage.select(time("12:00", "utc"));

# TEST\_CASE("julianday")

auto rows = storage.select(julianday(&Test::text));

auto rows = storage.select(datetime("now"));

auto rows = storage.select(date("now", "start of month", "+1 month", "-1 day"));

auto rows = storage.select(strftime("%Y %m %d", "now"));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\constraints\default.cpp - use in make\_column:

auto emailColumn = make\_column("Email", &User::email, default\_value("example@email.com"));

auto emailDefault = emailColumn.default\_value();

auto storage = make\_storage(

{},

make\_table("induction",

make\_column("timestamp", &Induction::time, default\_value(datetime("now", "localtime")))));

auto storage =

make\_storage({},

make\_table("contacts",

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

make\_column("first\_name", &Contact::firstName, default\_value<std::string>("")),

make\_column("last\_name", &Contact::lastName, default\_value<std::string>("")),

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

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\unique\_cases\delete\_with\_two\_fields.cpp: IN clause:

# SECTION("static condition"):

storage.remove\_all<Device>(where(in(std::make\_tuple(&Device::serialNumber, &Device::deviceId),

values(std::make\_tuple("abc", "123"), std::make\_tuple("def", "456")))));

## DELETE FROM 'devices' WHERE (("devices"."serial\_number", "devices"."device\_id") IN (VALUES (?, ?), (?, ?)))

Such tuples as in values() can be obtained from an appropriate select(columns(…)) as in:

auto rows = storage.select(columns(&Device::deviceId, &Device::serialNumber));

auto val1 = rows[0];

auto val2 = rows[1];

static\_assert(std::is\_same<decltype(val1), std::tuple<std::string, std::string>>::value);

static\_assert(std::is\_same<decltype(val2), std::tuple<std::string, std::string>>::value);

storage.remove\_all<Device>(where(in(std::make\_tuple(&Device::serialNumber, &Device::deviceId),

values(val1,val2))));

auto chs = storage.select(total\_changes()); chs[0] == 2

# SECTION("dynamic condition")

storage.remove\_all<Device>(

where(in(std::make\_tuple(&Device::serialNumber, &Device::deviceId),

values(std::vector<std::tuple<std::string, std::string>>{std::make\_tuple("abc", "123"),std::make\_tuple("def", "456")}))));

such vectors as in values() can be obtained from an appropriate select(columns(…)) as in:

auto rows = storage.select(columns(&Device::deviceId, &Device::serialNumber));

storage.remove\_all<Device>(

where(in(std::make\_tuple(&Device::serialNumber, &Device::deviceId),

values(rows))));

auto chs = storage.select(total\_changes()); // chs[0] == 2

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\ast\upsert\_clause.cpp:

## CREATE TABLE phonebook2(

## name TEXT PRIMARY KEY,

## phonenumber TEXT,

## validDate DATE

## );

## INSERT INTO phonebook2(name, phonenumber, validDate)

## VALUES('Alice', '704-555-1212', '2018-05-08')

## ON CONFLICT(name) DO UPDATE SET

## phonenumber = excluded.phonenumber,

## validDate = excluded.validDate

## WHERE excluded.validDate > phonebook2.validDate;

struct phonebook{

std::string name;

std::string phonenumber;

std::string validDate;

};

auto phonebookTable = make\_table("phonebook",

make\_column("name", &phonebook::name, primary\_key()),

make\_column("phonenumber", &phonebook::phonenumber, default\_value("")),

make\_column("validDate", &phonebook::validDate));

auto storage = make\_storage("upsert\_complete.sqlite", phonebookTable);

storage.sync\_schema();

storage.insert(into<phonebook>(), columns(&phonebook::name, &phonebook::phonenumber, &phonebook::validDate), values(std::make\_tuple("Alice", "506-555-1212", datetime("now"))),

on\_conflict(&phonebook::name).do\_update(set(assign(&phonebook::phonenumber, excluded(&phonebook::phonenumber)),assign(&phonebook::validDate, excluded(&phonebook::validDate))), where(excluded(&phonebook::validDate) > c(&phonebook::validDate))));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\explicit\_columns.cpp:

# Inheritance between persistent types:

struct Object {

int id;

};

struct User : Object {

std::string name;

User(decltype(id) id\_, decltype(name) name\_) : Object{id\_}, name(std::move(name\_)) {}

};

struct Token : Object {

std::string token;

int usedId;

Token(decltype(id) id\_, decltype(token) token\_, decltype(usedId) usedId\_) :

Object{id\_}, token(std::move(token\_)), usedId(usedId\_) {}

};

auto storage = make\_storage(

"column\_pointer.sqlite",

make\_table<User>("users", make\_column("id", &User::id, primary\_key()), make\_column("name", &User::name)),

make\_table<Token>("tokens",

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

make\_column("token", &Token::token),

make\_column("user\_id", &Token::usedId),

foreign\_key(&Token::usedId).references(column<User>(&User::id))));

# Explicit columns:

auto w = where(is\_equal(&User::name, "Bruno"));

{

auto rows = storage.select(column<User>(&User::id), w);

REQUIRE(rows.size() == 1);

REQUIRE(rows.front() == brunoId);

}

{

auto rows2 = storage.select(columns(column<User>(&User::id)), w);

REQUIRE(rows2.size() == 1);

REQUIRE(std::get<0>(rows2.front()) == brunoId);

auto rows3 = storage.select(columns(column<User>(&Object::id)), w);

REQUIRE(rows3 == rows2);

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\constraints\foreign\_key.cpp:

# Foreign keys

auto table1 = make\_table("test\_1",

make\_column("id", &test1::id, primary\_key()),

make\_column("val1", &test1::val1),

make\_column("val2", &test1::val2));

auto table2 = make\_table("test\_2",

make\_column("id", &test2::id, primary\_key()),

make\_column("fk\_id", &test2::fk\_id),

make\_column("val1", &test2::val1),

make\_column("val2", &test2::val2),

foreign\_key(&test2::fk\_id).references(&test1::id));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\foreign\_key.cpp:

# TEST\_CASE("statement\_serializator foreign key")

auto fk = foreign\_key(&Visit::userId).references(&User::id);

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_update.no\_action();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_update.restrict\_();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_update.set\_null();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_update.set\_default();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_update.cascade();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_delete.no\_action();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_delete.restrict\_();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_delete.set\_null();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_delete.set\_default();

auto fk = foreign\_key(&Visit::userId).references(&User::id).on\_delete.cascade();

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

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\get\_all.cpp

# TEST\_CASE("Prepared get all")

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

auto sql\_str = storage.dump(statement);

auto rows = storage.execute(statement);

auto statement = storage.prepare(get\_all<User>(where(lesser\_than(&User::id, 3))));

auto id = 3;

auto statement = storage.prepare(get\_all<User>(where(lesser\_than(&User::id, std::ref(id)))));

auto statement = storage.prepare(

get\_all<User>(where(lesser\_or\_equal(&User::id, 1) and is\_equal(&User::name, "Team BS")), limit(10))); auto statement = storage.prepare(

get\_all<User>(where(lesser\_or\_equal(&User::id, std::ref(id)) and is\_equal(&User::name, std::ref(name)))));

auto statement = storage.prepare(

get\_all<User>(where(lesser\_or\_equal(&User::id, 2) and (like(&User::name, "T%") or glob(&User::name, "\*S"))),

limit(20.0f)));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\get\_all\_custom\_containers.cpp:

tester.testContainer<std::vector<User>>(storage.get\_all<User>());

tester.testContainer<std::vector<User>>(storage.get\_all<User, std::vector<User>>());

tester.testContainer<std::deque<User>>(storage.get\_all<User, std::deque<User>>());

tester.testContainer<std::list<User>>(storage.get\_all<User, std::list<User>>());

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\get\_all\_optional.cpp

auto statement = storage.prepare(get\_all\_optional<User>());

auto statement = storage.prepare(get\_all\_optional<User>(where(lesser\_than(&User::id, 3))));

auto id = 3;

auto statement = storage.prepare(get\_all\_optional<User>(where(lesser\_than(&User::id, std::ref(id)))));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\get\_all\_pointer.cpp:

# TEST\_CASE("Prepared get all pointer")

auto statement = storage.prepare(get\_all\_pointer<User>());

std::unique\_ptr<User>

auto statement = storage.prepare(get\_all\_pointer<User>(where(lesser\_than(&User::id, 3))));

auto id = 3;

auto statement = storage.prepare(get\_all\_pointer<User>(where(lesser\_than(&User::id, std::ref(id)))));

auto rows = storage.execute(statement);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\unique\_cases\get\_all\_with\_two\_tables.cpp:

SECTION("straight insert") {

std::vector<Pattern> patterns;

patterns.push\_back({"n"});

patterns.push\_back({"w"});

storage.begin\_transaction();

storage.insert\_range(patterns.begin(), patterns.end());

}

SECTION("pointers insert") {

std::vector<std::unique\_ptr<Pattern>> patterns;

patterns.push\_back(std::make\_unique<Pattern>(Pattern{"n"}));

patterns.push\_back(std::make\_unique<Pattern>(Pattern{"w"}));

storage.begin\_transaction();

storage.insert\_range<Pattern>(patterns.begin(),

patterns.end(),

[](const std::unique\_ptr<Pattern>& pointer) -> const Pattern& {

return \*pointer;

});

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\get\_optional.cpp:

# TEST\_CASE("Prepared get optional")

auto statement = storage.prepare(get\_optional<User>(1));

auto user = storage.execute(statement);

REQUIRE(user.has\_value());

REQUIRE(\*user == User{1, "Team BS"});

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\get\_pointer.cpp

# TEST\_CASE("Prepared get pointer")

auto statement = storage.prepare(get\_pointer<User>(1));

REQUIRE(get<0>(statement) == 1);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\glob.cpp

# TEST\_CASE("Glob")

auto employees = storage.get\_all<Employee>(where(glob(&Employee::salary, "1\*")));

auto employees = storage.get\_all<Employee>(where(glob(&Employee::firstName, "Shwet?")));

auto employees = storage.get\_all<Employee>(where(glob(&Employee::lastName, "[A-J]\*")));

auto employees = storage.get\_all<Employee>(where(glob(&Employee::lastName, "[^A-J]\*")));

auto rows = storage.select(glob(&Employee::firstName, "S\*"));

auto rows = storage.select(glob(distinct(&Employee::firstName), "S\*"));

auto rows = storage.select(columns(not glob(&Employee::firstName, "S\*")));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\in.cpp

# TEST\_CASE("In")

rows = storage.get\_all<User>(where(in(&User::id, {1, 2, 3})));

rows = storage.get\_all<User>(where(c(&User::id).in(1, 2, 3)));

auto statement = storage.prepare(get\_all<User>(where(c(&User::id).in(1, 2, 3))));

std::vector<int> inArgument;

inArgument.push\_back(1);

inArgument.push\_back(2);

inArgument.push\_back(3);

rows = storage.get\_all<User>(where(in(&User::id, inArgument)));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\schema\index.cpp:

# TEST\_CASE("statement\_serializator index")

auto index = make\_index("id\_index", &User::id);

value = internal::serialize(index, context);

expected = "CREATE INDEX IF NOT EXISTS 'id\_index' ON 'users' (\"id\")";

auto index = make\_index("idx\_users\_id", indexed\_column(&User::id).desc());

value = internal::serialize(index, context);

expected = "CREATE INDEX IF NOT EXISTS 'idx\_users\_id' ON 'users' (\"id\" DESC)";

auto index = make\_index("idx\_users\_id", indexed\_column(&User::id).asc());

auto index = make\_index("idx\_users\_id",indexed\_column(&User::id).collate("compare"));

auto index = make\_index("my\_index", indexed\_column(&User::id).collate("compare").asc());

// compound column:

// Juan Dent added:

TEST\_CASE("Compound index") {

struct User {

int id = 0;

std::string name;

};

auto table = make\_table("users", make\_column("id", &User::id), make\_column("name", &User::name));

using storage\_impl\_t = internal::storage\_impl<decltype(table)>;

auto storageImpl = storage\_impl\_t{table};

using context\_t = internal::serializator\_context<storage\_impl\_t>;

context\_t context{storageImpl};

std::string value;

decltype(value) expected;

auto index = make\_index("idx\_users\_id\_and\_name", indexed\_column(&User::id).asc(), indexed\_column(&User::name).asc());

value = internal::serialize(index, context);

expected = "CREATE INDEX IF NOT EXISTS 'idx\_users\_id\_and\_name' ON 'users' (\"id\" ASC, \"name\" ASC)";

REQUIRE(value == expected);

auto storage = make\_storage("compound\_index.sqlite", index, table);

storage.sync\_schema();

storage.insert(User{ 1,"juan" });

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\index\_tests.cpp:

auto storage = make\_storage({}, make\_index("name\_index", indexed\_column(&User::name).collate("binary")), table);

storage.sync\_schema();

auto storage =

make\_storage({}, make\_index("name\_index", indexed\_column(&User::name).collate("nocase").desc()), table);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\insert.cpp:

auto statement = storage.prepare(

insert(into<User>(), columns(&User::id, &User::name), values(std::make\_tuple(1, "Ellie"))));

storage.execute(statement);

storage.insert(into<User>(), columns(&User::id, &User::name), values(std::make\_tuple(1, "Ellie")));

storage.insert(into<User>(),

columns(&User::id, &User::name), (std::make\_tuple(1, "Ellie"), std::make\_tuple(5, "Calvin")));

storage.insert(into<Artist>(), default\_values());

storage.insert(into<ArtistBackup>(), select(columns(&Artist::id, &Artist::name)));

User user{0, "Stromae"};

auto statement = storage.prepare(insert(std::ref(user)));

auto insertedId = storage.execute(statement);

user.name = “Lehman”;

insertedId = storage.execute(statement);

get<0>(statement).name = "Sia";

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\insert\_explicit.cpp:

auto statement = storage.prepare(insert(user, columns(&User::name)));

* INSERT INTO 'users' ("name") VALUES (?)

auto statement = storage.prepare(insert(visit, columns(&Visit::id, &Visit::userId)));

* INSERT INTO 'visits' ("id", "user\_id") VALUES (?, ?)

storage.insert(visit, columns(&Visit::userId));

* INSERT INTO 'visits' ("user\_id") VALUES (?)

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\insert\_range.cpp:

SECTION("container of pointers") {

std::vector<std::unique\_ptr<User>> usersPointers;

usersPointers.reserve(users.size());

std::transform(users.begin(), users.end(), std::back\_inserter(usersPointers), [](const User &user) {

return std::make\_unique<User>(user);

});

auto statement = storage.prepare(insert\_range<User>(usersPointers.begin(),

usersPointers.end(),

[](const std::unique\_ptr<User> &pointer) -> const User & {return \*pointer; }));

* INSERT INTO 'users' ("name") VALUES (?)

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\insert\_replace.cpp:

User user{5, "Gambit"};

auto statement = replace(user);

* REPLACE INTO 'users' ("id", "name") VALUES (?, ?)

auto statement = insert(user); // no PK defined in User!! Insert ==== replace

* INSERT INTO 'users' ("id", "name") VALUES (?, ?)

auto statement = insert(into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "The Weeknd")));

* INSERT INTO 'users' ("id", "name") VALUES (?, ?)

auto statement = insert(or\_abort(),

into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "The Weeknd")));

* INSERT OR ABORT INTO users ("id", "name") VALUES (1, 'The Weeknd')

auto statement = insert(or\_fail(),

into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "The Weeknd")));

* INSERT OR FAIL INTO users ("id", "name") VALUES (1, 'The Weeknd')

auto statement = insert(or\_ignore(),

into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "The Weeknd")));

* INSERT OR IGNORE INTO users ("id", "name") VALUES (1, 'The Weeknd')

auto statement = insert(or\_replace(),

into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "The Weeknd")));

* INSERT OR REPLACE INTO users ("id", "name") VALUES (1, 'The Weeknd')

auto statement = insert(or\_rollback(),

into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "The Weeknd")));

* INSERT OR ROLLBACK INTO users ("id", "name") VALUES (1, 'The Weeknd')

auto statement =

insert(or\_abort()[[4]](#footnote-4),

into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "The Weeknd"), std::make\_tuple(4, "Jonas Blue")));

* INSERT OR ABORT INTO users ("id", “name") VALUES (1, 'The Weeknd'), (4, 'Jonas Blue')

auto statement = insert(or\_abort()[[5]](#footnote-5), into<User>(), default\_values());

* INSERT OR ABORT INTO users DEFAULT VALUES

auto statement = insert(or\_abort()[[6]](#footnote-6), into<User>(), select(columns(&UserBackup::id, &UserBackup::name)));

* INSERT OR ABORT INTO users SELECT "users\_backup"."id", "users\_backup"."name" FROM 'users\_backup'

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\is\_null.cpp:

struct User {

int id = 0;

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

};

auto storage = make\_storage(

"",

make\_table("users", make\_column("id", &User::id, primary\_key()), make\_column("name", &User::name)));

storage.sync\_schema();

storage.replace(User{1, std::make\_unique<std::string>("Sheldon")});

storage.replace(User{2});

storage.replace(User{3, std::make\_unique<std::string>("Leonard")});

REQUIRE(storage.count<User>() == 3);

REQUIRE(storage.count<User>(where(is\_null(&User::name))) == 1);

REQUIRE(storage.count<User>(where(is\_not\_null(&User::name))) == 2);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\unique\_cases\join\_iterator\_ctor\_compilation\_error.cpp: NOTE: limit

auto statement = storage.prepare(select(columns(&Tag::text, count(&Tag::text)),

group\_by(&Tag::text),

order\_by(count(&Tag::text)).desc(),

limit(offs, lim)));

* SELECT "tags"."text", (COUNT("tags"."text")) FROM 'tags' GROUP BY "tags"."text" ORDER BY (COUNT("tags"."text")) DESC LIMIT ?, ?

auto statement = storage.prepare(select(columns(&Tag::text, count(&Tag::text)),

group\_by(&Tag::text),

order\_by(count(&Tag::text)).desc(),

limit(lim, offset(offs))));

* SELECT "tags"."text", (COUNT("tags"."text")) FROM 'tags' GROUP BY "tags"."text" ORDER BY (COUNT("tags"."text")) DESC LIMIT ? OFFSET ?

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\operators\like.cpp:

storage.insert(User{0, "Sia"});

storage.insert(User{0, "Stark"});

storage.insert(User{0, "Index"});

auto whereCondition = where(like(&User::name, "S%"));

{

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

REQUIRE(users.size() == 2);

}

* SELECT "users"."id", "users"."name" FROM 'users' WHERE ("users"."name" LIKE ?)

auto rows = storage.select(like("ototo", "ot\_to"));

REQUIRE(rows.size() == 1);

REQUIRE(rows.front() == true);

* SELECT ? LIKE ?

auto rows = storage.select(not like("ototo", "ot\_to"));

REQUIRE(rows.size() == 1);

REQUIRE(rows.front() == false);

* SELECT NOT (? LIKE ? )

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\logical\_operators.cpp:

stringValue = internal::serialize(c(0) and 0, context); // operator

stringValue = internal::serialize(0 and c(0), context); // operator

stringValue = internal::serialize(and\_(0, 0), context); // function

stringValue = internal::serialize(c(0).and\_(0), context); // member function

* expected = "(0 AND 0)";

stringValue = internal::serialize(c(&User::id) == 5 and c(&User::name) == "Ariana", context); // operator

stringValue = internal::serialize(is\_equal(&User::id, 5) and is\_equal(&User::name, "Ariana"), context); // functions

* expected = "(("id" = 5) AND ("name" = 'Ariana'))";

auto inValue = c(&User::id).in(1, 2, 3); // static in

auto inValue = in(&User::id, {1, 2, 3}); // dynamic in (can use vector)

* expected = ""id" IN (1, 2, 3)";

auto inValue = c(&User::id).not\_in(1, 2, 3); // static not in

auto inValue = not\_in(&User::id, {1, 2, 3}); // dynamic not in (can use vector)

* expected = ""id" NOT IN (1, 2, 3)";

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\built\_in\_functions\_tests\math\_functions.cpp:

[Built-In Mathematical SQL Functions (sqlite.org)](https://www.sqlite.org/lang_mathfunc.html)

#ifdef SQLITE\_ENABLE\_MATH\_FUNCTIONS // must define this!!

Defines these functions: (100% of built-in scalar SQL math functions available in SQLITE!)

acos(), acosh(), asin(), asinh(), atan(), atan2(), atanh(), ceil(), ceiling(), cos(), cosh(), degrees(), exp(), floor(), ln(), log(), log(,), log2(), mod\_f(,), pi(), pow(,), power(,), radians(), sin(), sinh(), sqrt(), tan(), tanh(), trunc().

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\schema\new\_old.cpp:

Para usar en triggers:

auto expression = new\_(&User::id);

value = serialize(expression, context);

expected = "NEW.\"id\"";

auto expression = old(&User::id);

value = serialize(expression, context);

expected = "OLD.\"id\"";

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\pragma\_tests.cpp:

auto jm = stor->pragma.journal\_mode();

stor->pragma.journal\_mode(journal\_mode::MEMORY);

stor->pragma.journal\_mode(journal\_mode::WAL);

stor->pragma.journal\_mode(journal\_mode::OFF);

stor->pragma.journal\_mode(journal\_mode::PERSIST);

stor->pragma.journal\_mode(journal\_mode::TRUNCATE);

storage.pragma.synchronous(value);

REQUIRE(storage.pragma.synchronous() == value);

auto version = storage.pragma.user\_version();

storage.pragma.user\_version(version + 1);

REQUIRE(storage.pragma.user\_version() == version + 1);

storage.pragma.auto\_vacuum(0);

REQUIRE(storage.pragma.auto\_vacuum() == 0);

auto value = storage.pragma.busy\_timeout();

REQUIRE(value == 0);

storage.pragma.busy\_timeout(10);

value = storage.pragma.busy\_timeout();

REQUIRE(value == 10);

REQUIRE(storage.pragma.integrity\_check() == std::vector<std::string>{"ok"});

REQUIRE(storage.pragma.integrity\_check(5) == std::vector<std::string>{"ok"});

REQUIRE(storage.pragma.integrity\_check(tablename) == std::vector<std::string>{"ok"});

storage.pragma.application\_id(3);

REQUIRE(storage.pragma.application\_id() == 3);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\unique\_cases\prepare\_get\_all\_with\_case.cpp:

const std::string name = "Iggy";

Empty before WHEN clause

auto statement0 = storage.prepare(get\_all<UserProfile>(where(is\_equal(

&UserProfile::firstName,

case\_<std::string>().when((length(name) > 0), then(name)).else\_(&UserProfile::firstName).end()))));

auto t0 = storage.execute(statement0);

* SELECT "user\_profiles"."id", "user\_profiles"."first\_name" FROM 'user\_profiles' WHERE (("user\_profiles"."first\_name" = CASE WHEN ((LENGTH(?)) > ?) THEN ? ELSE "user\_profiles"."first\_name" END))

Not empty before WHEN clause

auto statement3 = storage.prepare(get\_all<UserProfile>(where(is\_equal(

&UserProfile::id,

case\_<int>(c(&UserProfile::id) \* 2).when(2, then(1)).when(4, then(2)).else\_(3).end()))));

auto r2 = storage.execute(statement3);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\column\_constraints\primary\_key.cpp:

SECTION("single column pk") {

auto pk = primary\_key(&User::id);

value = serialize(pk, context);

expected = "PRIMARY KEY(id)";

}

SECTION("double column pk") {

auto pk = primary\_key(&User::id, &User::name);

value = serialize(pk, context);

expected = "PRIMARY KEY(id, name)";

}

SECTION("empty pk asc") {

auto pk = primary\_key().asc();

value = serialize(pk, context);

expected = "PRIMARY KEY ASC";

}

SECTION("empty pk desc") {

auto pk = primary\_key().desc();

value = serialize(pk, context);

expected = "PRIMARY KEY DESC";

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\private\_getters\_tests.cpp:

auto storage = make\_storage(

{},

make\_table("a", make\_column("role", &A::getId, &A::setId, primary\_key()), make\_column("name", &A::name)));

storage.sync\_schema();

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\schema\raise.cpp: (for use inside triggers)

SECTION("ignore") {

auto expression = raise\_ignore();

value = serialize(expression, context);

expected = "RAISE(IGNORE)";

}

SECTION("rollback") {

auto expression = raise\_rollback("no rap");

value = serialize(expression, context);

expected = "RAISE(ROLLBACK, 'no rap')";

}

SECTION("abort") {

auto expression = raise\_abort("no rap");

value = serialize(expression, context);

expected = "RAISE(ABORT, 'no rap')";

}

SECTION("fail") {

auto expression = raise\_fail("no rap");

value = serialize(expression, context);

expected = "RAISE(FAIL, 'no rap')";

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\remove.cpp:

auto id = 1;

auto statement = storage.prepare(remove<User>(std::ref(id)));

REQUIRE(get<0>(statement) == id);

REQUIRE(&get<0>(statement) == &id);

storage.execute(statement);

auto statement = storage.prepare(remove<User>(1));

REQUIRE(get<0>(statement) == 1);

storage.execute(statement);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\remove\_all.cpp:

auto statement = storage.prepare(remove\_all<User>());

storage.execute(statement);

REQUIRE(storage.count<User>() == 0);

auto statement = storage.prepare(remove\_all<User>(where(is\_equal(&User::id, 2))));

REQUIRE(get<0>(statement) == 2);

storage.execute(statement);

auto id = 2;

auto statement = storage.prepare(remove\_all<User>(where(is\_equal(&User::id, std::ref(id)))));

REQUIRE(get<0>(statement) == 2);

storage.execute(statement);

std::string name = "Shy'm";

auto id = 10;

auto statement = storage.prepare(remove\_all<User>(

where(is\_equal(&User::name, std::ref(name)) and lesser\_than(&User::id, std::ref(id)))));

REQUIRE(get<0>(statement) == "Shy'm");

REQUIRE(get<1>(statement) == 10);

storage.execute(statement);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\replace.cpp:

storage.replace(into<User>(), columns(&User::id, &User::name), values(std::make\_tuple(1, "Ellie")));

storage.replace(into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "Ellie"), std::make\_tuple(5, "Calvin")));

storage.replace(into<User>(),

columns(&User::id, &User::name),

values(std::make\_tuple(1, "Ellie"), std::make\_tuple(5, "Calvin")));

storage.replace(into<User>(), default\_values());

storage.replace(into<User>(), select(columns(5, "Carma")));

* REPLACE INTO users SELECT ?, ?

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\replace\_range.cpp:

users.push\_back(user);

users.push\_back(user2);

auto statement = storage.prepare(replace\_range(users.begin(), users.end()));

auto lambda = [](const std::unique\_ptr<User>& pointer) -> const User& {

return \*pointer;

};

userPointers.push\_back(std::make\_unique<User>(user));

userPointers.push\_back(std::make\_unique<User>(user2));

auto statement = storage.prepare(replace\_range<User>(userPointers.begin(), userPointers.end(), lambda));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\simple\_query.cpp:

auto twoAgain = storage.select(add(1, 1));

auto math = storage.select(columns(sqlite\_orm::div(10, 5), mul(2, 4)));

auto twoRows = storage.select(columns(1, 2));

* SELECT 1, 2

auto twoRowsUnion = storage.select(union\_all(select(columns(1, 2)), select(columns(3, 4))));

REQUIRE(twoRowsUnion.size() == 2);

REQUIRE(twoRowsUnion[0] == std::make\_tuple(1, 2));

REQUIRE(twoRowsUnion[1] == std::make\_tuple(3, 4));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\storage\_non\_crud\_tests.cpp:

rows = storage.select(&User::id, from<User>());

using als = alias\_u<User>;

rows = storage.select(alias\_column<als>(&User::id), from<als>());

rows = storage.select(&User::id, from<User>(), where(is\_equal(&User::name, "Bebe Rexha")));

using als = alias\_u<User>;

rows = storage.select(alias\_column<als>(&User::id), from<als>(),

where(is\_equal(alias\_column<als>(&User::name), "Bebe Rexha")));

struct User {

int id = 0;

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

};

storage.update\_all(set(assign(&User::name, nullptr)));

storage.update\_all(set(assign(&User::name, nullptr)), where(is\_equal(&User::id, 1)));

auto firstRow = storage.get\_no\_throw<Word>(firstId);

auto secondRow = storage.get\_pointer<Word>(secondId);

auto cols = columns(&Word::id, &Word::currentWord, &Word::beforeWord, &Word::afterWord, &Word::occurances);

auto rawTuples = storage.select(cols, where(eq(&Word::id, firstId)));

auto ordr = order\_by(&Word::id);

auto idsOnly = storage.select(&Word::id, ordr);

storage.update\_all(set(assign(&Word::currentWord, "ototo")), where(is\_equal(&Word::id, firstId)));

storage.replace\_range<Object>(vector.begin(),

vector.end(),

[](const std::unique\_ptr<Object> &pointer) -> const Object & {

return \*pointer;

});

std::vector<std::unique\_ptr<Object>> emptyVector;

storage.replace\_range<Object>(emptyVector.begin(),

emptyVector.end(),

[](const std::unique\_ptr<Object> &pointer) -> const Object & {

return \*pointer;

});

REQUIRE(storage.count<Object>() == 2);

storage.remove\_all<Object>(where(c(&Object::id) == 1));

REQUIRE(storage.count<Object>() == 1);

auto id = storage.insert(user, columns(&User::name, &User::age));

REQUIRE(storage.get<User>(id).email == "dummy@email.com");

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\storage\_tests.cpp:

TEST\_CASE("busy handler") {

auto storage = make\_storage({});

storage.busy\_handler([](int /\*timesCount\*/) {

return 0;

});

}

TEST\_CASE("drop table") {

storage.drop\_table(usersTableName);

REQUIRE(!storage.table\_exists(usersTableName));

REQUIRE(storage.table\_exists(visitsTableName));

}

storage.rename\_table<User>(userNewTableName);

storage.rename\_table(usersTableName, userNewTableName);

// Need to sync\_schema before inserting, updating or reading from User!!

storage.has\_dependent\_rows(user5)

\*storage.column\_name(&User::id) == "id"

Sqlite\_orm.h:

/\*\*

\* This is a cute function used to replace migration up/down functionality.

\* It performs check storage schema with actual db schema and:

\* \* if there are excess tables exist in db they are ignored (not dropped)

\* \* every table from storage is compared with it's db analog and

\* \* if table doesn't exist it is being created

\* \* if table exists its colums are being compared with table\_info from db and

\* \* if there are columns in db that do not exist in storage (excess) table will be dropped and

\* recreated

\* \* if there are columns in storage that do not exist in db they will be added using `ALTER TABLE

\* ... ADD COLUMN ...' command

\* \* if there is any column existing in both db and storage but differs by any of

\* properties/constraints (pk, notnull, dflt\_value) table will be dropped and recreated. Be aware that

\* `sync\_schema` doesn't guarantee that data will not be dropped. It guarantees only that it will make db

\* schema the same as you specified in `make\_storage` function call. A good point is that if you have no db

\* file at all it will be created and all tables also will be created with exact tables and columns you

\* specified in `make\_storage`, `make\_table` and `make\_column` calls. The best practice is to call this

\* function right after storage creation.

\* @param preserve affects function's behaviour in case it is needed to remove a column. If it is `false`

\* so table will be dropped if there is column to remove if SQLite version is < 3.35.0 and rmeove column if SQLite version >= 3.35.0,

\* if `true` - table is being copied into another table, dropped and copied table is renamed with source table name.

\* Warning: sync\_schema doesn't check foreign keys cause it is unable to do so in sqlite3. If you know how to get foreign key info please

\* submit an issue https://github.com/fnc12/sqlite\_orm/issues

\* @return std::map with std::string key equal table name and `sync\_schema\_result` as value.

\* `sync\_schema\_result` is a enum value that stores table state after syncing a schema. `sync\_schema\_result`

\* can be printed out on std::ostream with `operator<<`.

\*/

std::map<std::string, sync\_schema\_result> sync\_schema(bool preserve = false);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\static\_tests\storage\_traits.cpp:

auto column1 = make\_column("id", &Table::id);

auto column2 = make\_column("a", &Table::a);

auto column3 = make\_column("b", &Table::b);

auto column4 = make\_column("c", &Table::c);

auto uniqueC = sqlite\_orm::unique(&Table::a, &Table::b, &Table::c);

auto table = make\_table("table", column1, column2, column3, column4, uniqueC);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\table\_tests.cpp:

REQUIRE(\*table.find\_column\_name(&Contact::id) == "contact\_id");

REQUIRE(\*table.find\_column\_name(&Contact::setId) == "contact\_id");

auto compositeKeyColumnsNames = table.composite\_key\_columns\_names();

auto visitCallsCount = 0;

visitTable.for\_each\_foreign\_key([&visitCallsCount](auto& foreignKey) {

++visitCallsCount;

REQUIRE(foreignKey == foreign\_key(&Visit::location).references(&Location::id));

});

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\tests.cpp:

auto sqlLength = storage.limit.sql\_length();

auto newSqlLength = sqlLength - 10;

storage.limit.sql\_length(newSqlLength);

sqlLength = storage.limit.sql\_length();

REQUIRE(sqlLength == newSqlLength);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\tests2.cpp:

storage.remove\_all<Object>();

auto id1 = storage.insert(Object{0, "Skillet"});

REQUIRE(storage.count<Object>() == 1);

storage.remove<Object>(id1);

REQUIRE(storage.count<Object>() == 0);

storage.replace(Object{1, "Skillet"});

assert(storage.count<Object>() == 1);

storage.remove<Object>(1, "Skillet");

REQUIRE(storage.count<Object>() == 0);

auto storage =

make\_storage({},

make\_table("products",

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

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

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

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

make\_column("net\_price",

&Product::netPrice, generated\_always\_as(c(&Product::price) \* (1 - c(&Product::discount)) \* (1 + c(&Product::tax))))));

std::vector<std::unique\_ptr<Object>> pointers;

pointers.reserve(initList.size());

std::transform(initList.begin(), initList.end(), std::back\_inserter(pointers), [](const Object &object) {

return std::make\_unique<Object>(Object{object});

});

storage.insert\_range<Object>(pointers.begin(),

pointers.end(),

[](const std::unique\_ptr<Object> &pointer) -> const Object & {

return \*pointer;

});

// User defined functions: SCAL

struct HasPrefixFunction {

bool operator()(const std::string &str, const std::string &prefix) {

return str.compare(0, prefix.size(), prefix) == 0;

}

static std::string name() {

return "HAS\_PREFIX";

}

};

storage.create\_scalar\_function<HasPrefixFunction>();

auto rows = storage.select(func<HasPrefixFunction>("one", "o"));

storage.delete\_scalar\_function<HasPrefixFunction>();

// User defined functions: AGGREGATE

struct MeanFunction {

double total = 0;

int count = 0;

void step(double value) {

total += value;

++count;

}

double fin() const {

return total / count;

}

static std::string name() {

return "MEAN";

}

};

storage.create\_aggregate\_function<MeanFunction>();

auto rows = storage.select(func<MeanFunction>(&User::id));

storage.delete\_aggregate\_function<MeanFunction>();

storage.update\_all(set(c(&Data::mContentLang1) = data.mContentLang1,

c(&Data::mContentLang2) = data.mContentLang2,

c(&Data::mContentLang3) = data.mContentLang3,

c(&Data::mContentLang4) = data.mContentLang4,

c(&Data::mContentLang5) = data.mContentLang5),

where(c(&Data::mDefault) == data.mDefault));

// wide strings

struct Alphabet {

int id;

std::wstring letters;

};

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

make\_table("alphabets",

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

make\_column("letters", &Alphabet::letters)));

auto vms = storage.select(columns(&Data::mId, &Data::mLangId), where(like(&Data::mName, CurrCity)));

storage.update(selena);

* UPDATE 'COMPANY' SET "NAME" = ?, "AGE" = ?, "ADDRESS" = ?, "SALARY" = ? WHERE "INDEX" = ?

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\tests4.cpp:

// unique ptr in update:

struct User {

int id = 0;

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

};

auto storage = make\_storage(

{},

make\_table("users", make\_column("id", &User::id, primary\_key()), make\_column("name", &User::name)));

storage.sync\_schema();

storage.insert(User{});

storage.insert(User{});

storage.insert(User{});

{

storage.update\_all(set(assign(&User::name, std::make\_unique<std::string>("Nick"))));

* UPDATE 'users' SET "name" = ?

for(auto& row : rows) {

REQUIRE(\*row.name == "Nick");

}

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

REQUIRE(storage.count<User>(where(is\_null(&User::name))) == 0);

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

storage.update\_all(set(assign(&User::name, move(ptr))));

REQUIRE(storage.count<User>(where(is\_not\_null(&User::name))) == 0);

// optional in update

struct User {

int id = 0;

std::optional<int> carYear; // will be empty if user takes the bus.

};

auto storage = make\_storage(

{},

make\_table("users", make\_column("id", &User::id, primary\_key()), make\_column("car\_year", &User::carYear)));

storage.sync\_schema();

storage.insert(User{});

storage.insert(User{});

storage.insert(User{});

storage.insert(User{0, 2006});

REQUIRE(storage.count<User>(where(is\_not\_null(&User::carYear))) == 1);

{

storage.update\_all(set(assign(&User::carYear, std::optional<int>{})));

REQUIRE(storage.count<User>(where(is\_not\_null(&User::carYear))) == 0);

}

storage.update\_all(set(assign(&User::carYear, nullptr)));

REQUIRE(storage.count<User>(where(is\_not\_null(&User::carYear))) == 0);

storage.update\_all(set(assign(&User::carYear, std::nullopt)));

REQUIRE(storage.count<User>(where(is\_not\_null(&User::carYear))) == 0);

// joins

storage.replace(will);

storage.replace(smith);

storage.replace(nicole);

// 3 Users

id = 1;

storage.replace(Visit{id++, will.id, 10});

storage.replace(Visit{id++, will.id, 20});

storage.replace(Visit{id++, will.id, 30});

storage.replace(Visit{id++, smith.id, 25});

storage.replace(Visit{id++, smith.id, 35});

// 5 visits

auto rows = storage.get\_all<User>(left\_join<Visit>(on(is\_equal(&Visit::userId, 2))));

// filters Visit table to those rows where Visit::userId == 2 ==> which are 2 rows X 3 users ==> 6 rows

REQUIRE(rows.size() == 6);

auto rows = storage.get\_all<User>(left\_join<Visit>(on(is\_equal(&Visit::userId, &User::id))));

// there is no Visit with userId == 3, so Nicole appears only once, Will appears 3 times, Smith 2 times

REQUIRE(rows.size() == 6);

auto rows = storage.select(columns(&User::id, &User::name, &Visit::id, &Visit::userId),

join<Visit>(on(is\_equal(&Visit::userId, &User::id))));

REQUIRE(rows.size() == 5);

auto rows = storage.select(columns(&User::id, &User::name, &Visit::id, &Visit::userId), natural\_join<Visit>());

REQUIRE(rows.size() == 3);

auto rows = storage.select(columns(&User::id, &User::name, &Visit::id, &Visit::userId), left\_outer\_join<Visit>(on(is\_equal(&Visit::userId, &User::id))));

REQUIRE(rows.size() == 6);

auto rows = storage.select(columns(&User::id, &User::name, &Visit::id, &Visit::userId),

join<Visit>(on(is\_equal(&Visit::userId, 2))));

// filters Visit table to those rows where Visit::userId == 2 ==> which are 2 rows X 3 users ==> 6 rows

REQUIRE(rows.size() == 6);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\tests5.cpp:

// iterate blob

struct Test {

int64\_t id;

std::vector<char> key;

};

struct TestComparator {

bool operator()(const Test& lhs, const Test& rhs) const {

return lhs.id == rhs.id && lhs.key == rhs.key;

}

};

auto db =

make\_storage("",

make\_table("Test", make\_column("key", &Test::key), make\_column("id", &Test::id, primary\_key())));

db.sync\_schema(true);

std::vector<char> key(255);

iota(key.begin(), key.end(), 0);

Test v{5, key};

db.replace(v);

TestComparator testComparator;

for(auto& obj: db.iterate<Test>()) {

REQUIRE(testComparator(obj, v));

} // test that view\_t and iterator\_t compile

// different getters and setters

auto filename = "different.sqlite";

auto storage0 = make\_storage(

filename,

make\_table("users", make\_column("id", &User::id, primary\_key()), make\_column("name", &User::name)));

auto storage1 = make\_storage(filename,

make\_table("users",

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

make\_column("name", &User::setNameByConstRef, &User::getNameByVal)));

auto storage2 =

make\_storage(filename,

make\_table("users",

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

make\_column("name", &User::getConstNameByRefConst, &User::setNameByRef)));

storage0.sync\_schema();

storage0.remove\_all<User>();

REQUIRE(storage0.count<User>() == 0);

REQUIRE(storage1.count<User>() == 0);

REQUIRE(storage2.count<User>() == 0);

storage0.replace(User{1, "Da buzz"});

REQUIRE(storage0.count<User>() == 1);

REQUIRE(storage1.count<User>() == 1);

REQUIRE(storage2.count<User>() == 1);

auto ids = storage0.select(&User::id);

REQUIRE(ids.size() == 1);

REQUIRE(ids.front() == 1);

auto ids2 = storage1.select(&User::getIdByValConst);

REQUIRE(ids == ids2);

auto ids3 = storage1.select(&User::setIdByVal);

REQUIRE(ids3 == ids2);

auto ids4 = storage2.select(&User::getConstIdByRefConst);

REQUIRE(ids4 == ids3);

auto ids5 = storage2.select(&User::setIdByRef);

REQUIRE(ids5 == ids4);

auto ids2 = storage1.select(&User::getIdByValConst, where(is\_equal(&User::setNameByConstRef, "Da buzz")));

REQUIRE(ids == ids2);

auto ids3 = storage1.select(&User::setIdByVal, where(is\_equal(&User::getNameByVal, "Da buzz")));

REQUIRE(ids3 == ids2);

// Dump

struct User {

int id = 0;

std::optional<int> carYear; // will be empty if user takes the bus.

};

auto storage = make\_storage(

{},

make\_table("users", make\_column("id", &User::id, primary\_key()), make\_column("car\_year", &User::carYear)));

storage.sync\_schema();

auto userId\_1 = storage.insert(User{0, {}});

auto userId\_2 = storage.insert(User{0, 2006});

std::ignore = userId\_2;

REQUIRE(storage.count<User>(where(is\_not\_null(&User::carYear))) == 1);

auto rows = storage.select(&User::carYear, where(is\_equal(&User::id, userId\_1)));

REQUIRE(rows.size() == 1);

REQUIRE(!rows.front().has\_value());

const std::string dumpUser1 = storage.dump(allUsers[0]);

REQUIRE(dumpUser1 == std::string{"{ id : '1', car\_year : 'null' }"});

auto storage = make\_storage("",

make\_table("A",

make\_column("address", &A::getAddress, &A::setAddress),

make\_column("type", &A::getType, &A::setType),

make\_column("idx", &A::getIndex, &A::setIndex),

make\_column("value", &A::getValue, &A::setValue),

primary\_key(&A::getAddress, &A::getType, &A::getIndex)));

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\transaction\_tests.cpp:

struct Object {

int id = 0;

std::string name;

};

auto filename = "transaction\_test.sqlite";

::remove(filename);

auto storage = make\_storage(

"test\_transaction\_guard.sqlite",

make\_table("objects", make\_column("id", &Object::id, primary\_key()), make\_column("name", &Object::name)));

REQUIRE(!storage.is\_opened());

storage.sync\_schema();

REQUIRE(!storage.is\_opened());

storage.transaction([&] {

storage.insert(Object{0, "Jack"});

REQUIRE(storage.is\_opened());

return true;

});

REQUIRE(!storage.is\_opened());

// transaction rollback

SECTION("insert, call make a storage to call an exception and check that rollback was fired") {

auto countBefore = storage.count<Object>();

try {

storage.transaction([&] {

storage.insert(Object{0, "John"});

storage.get<Object>(-1);

REQUIRE(false);

return true;

});

} catch(...) {

auto countNow = storage.count<Object>();

REQUIRE(countBefore == countNow);

}

}

// transaction\_guard

SECTION("insert, call make a storage to call an exception and check that rollback was fired") {

auto countBefore = storage.count<Object>();

try {

auto guard = storage.transaction\_guard();

storage.insert(Object{0, "John"});

storage.get<Object>(-1);

REQUIRE(false);

} catch(...) {

auto countNow = storage.count<Object>();

REQUIRE(countBefore == countNow);

}

}

SECTION("commit explicitly and check that after exception data was saved") {

auto countBefore = storage.count<Object>();

try {

auto guard = storage.transaction\_guard();

storage.insert(Object{0, "John"});

guard.commit();

storage.get<Object>(-1);

REQUIRE(false);

} catch(...) {

auto countNow = storage.count<Object>();

REQUIRE(countNow == countBefore + 1);

}

}

SECTION("rollback explicitly") {

auto countBefore = storage.count<Object>();

try {

auto guard = storage.transaction\_guard();

storage.insert(Object{0, "Michael"});

guard.rollback();

storage.get<Object>(-1);

REQUIRE(false);

} catch(...) {

auto countNow = storage.count<Object>();

REQUIRE(countNow == countBefore);

}

}

SECTION("commit on exception") {

auto countBefore = storage.count<Object>();

try {

auto guard = storage.transaction\_guard();

guard.commit\_on\_destroy = true;

storage.insert(Object{0, "Michael"});

storage.get<Object>(-1);

REQUIRE(false);

} catch(...) {

auto countNow = storage.count<Object>();

REQUIRE(countNow == countBefore + 1);

}

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\statement\_serializator\_tests\schema\trigger.cpp:

using internal::serialize;

struct Lead {

int id = 0;

std::string firstName;

std::string lastName;

std::string email;

std::string phone;

};

auto table = make\_table("leads",

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

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

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

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

make\_column("phone", &Lead::phone));

using storage\_impl\_t = internal::storage\_impl<decltype(table)>;

auto storageImpl = storage\_impl\_t{table};

using context\_t = internal::serializator\_context<storage\_impl\_t>;

context\_t context{storageImpl};

std::string value;

decltype(value) expected;

SECTION("without for each row") {

auto expression = make\_trigger("validate\_email\_before\_insert\_leads",

before()

.insert()

.on<Lead>()

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

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

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

.end()))

.end());

value = serialize(expression, context);

expected =

"CREATE TRIGGER IF NOT EXISTS 'validate\_email\_before\_insert\_leads' BEFORE INSERT ON 'leads' BEGIN SELECT "

"CASE WHEN NOT (NEW.\"email\" LIKE '%\_@\_\_%.\_\_%' ) THEN RAISE(ABORT, 'Invalid email address') END; END";

}

REQUIRE(value == expected);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\trigger\_tests.cpp:

make\_trigger(

"trigger\_insert",

after()

.update\_of(&TestInsert::x)

.on<TestInsert>()

.begin(insert(test\_insert),

insert(into<TestInsert>(),

columns(&TestInsert::id, &TestInsert::text, &TestInsert::x, &TestInsert::y),

values(std::make\_tuple(123, "HelloTrigger", 12, 13))),

insert(test\_insert, columns(&TestInsert::id, &TestInsert::text, &TestInsert::x, &TestInsert::y)),

replace(TestInsert{8, "replace", 3, 4}),

replace(test\_insert))

.end())

make\_trigger("trigger\_update",

after()

.insert()

.on<TestUpdate>()

.begin(update(TestUpdate{test\_update.id, "update", test\_update.x, test\_update.y}),

update\_all(set(c(&TestUpdate::x) = 42), where(c(&TestUpdate::text) == "update")))

.end())

make\_trigger(

"trigger\_delete",

after()

.insert()

.on<TestDelete>()

.begin(

// select(columns(&TestDelete::id), where(greater\_than(&TestDelete::x, select(avg(&TestDelete::x))))), // **TODO near "(": syntax error: SQL logic error (expression is surronded by parenthesis and SQL returns an error for that)**

remove<TestDelete>(test\_delete.id),

remove\_all<TestDelete>(where(c(&TestDelete::text) != "test")))

.end())

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\tuple\_helper\_tests.cpp:

auto lambda = [&types](const auto &item) {

auto& id = typeid(item);

types.push\_back(typeid(item));

};

SECTION("char, long") {

std::tuple<char, long> tuple;

expected.push\_back(typeid(char));

expected.push\_back(typeid(long));

iterate\_tuple(tuple, lambda);

}

SECTION("char, long") {

iterate\_tuple<std::tuple<char, long>>(lambda);

expected.push\_back(typeid(char));

expected.push\_back(typeid(long));

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\constraints\unique.cpp:

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, unique()))

storage.insert(Contact{0, "John", "Doe", "john.doe@gmail.com"});

try {

storage.insert(Contact{0, "Johnny", "Doe", "john.doe@gmail.com"});

REQUIRE(false);

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

//..

} catch(...) {

REQUIRE(false);

}

make\_table("shapes",

make\_column("shape\_id", &Shape::id, primary\_key()),

make\_column("background\_color", &Shape::backgroundColor),

make\_column("foreground\_color", &Shape::foregroundColor),

sqlite\_orm::unique(&Shape::backgroundColor, &Shape::foregroundColor))

storage.insert(Shape{0, "red", "green"});

storage.insert(Shape{0, "red", "blue"});

try {

storage.insert(Shape{0, "red", "green"});

REQUIRE(false);

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

//..

} catch(...) {

REQUIRE(false);

}

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\update.cpp:

User user{2, "Stromae"};

auto statement = storage.prepare(update(std::ref(user)));

storage.execute(statement);

REQUIRE(storage.get<User>(2) == user);

user.name = "Sia";

storage.execute(statement);

REQUIRE(storage.get<User>(2) == user);

auto statement = storage.prepare(update(user));

storage.execute(statement);

get<0>(statement).name = "Sia";

storage.execute(statement);

REQUIRE(storage.count<User>(where(is\_equal(&User::name, "Sia"))) == 1);

get<0>(statement) = {user.id, "Paris"};

storage.execute(statement);

REQUIRE(storage.count<User>(where(is\_equal(&User::name, "Paris"))) == 1);

C:\Components\sqlite\_orm\_dev\_23\_jan\_2022\tests\prepared\_statement\_tests\update\_all.cpp:

auto statement = storage.prepare(update\_all(set(assign(&User::name, conc(&User::name, "\_")))));

1. [Generated Columns (sqlite.org)](https://www.sqlite.org/gencol.html) [↑](#footnote-ref-1)
2. [Built-In Scalar SQL Functions (sqlite.org)](https://www.sqlite.org/lang_corefunc.html) [↑](#footnote-ref-2)
3. See this file for detailed examples of the use of some of the core functions! [↑](#footnote-ref-3)
4. Or the other conflict resolution algorithms [↑](#footnote-ref-4)
5. idem [↑](#footnote-ref-5)
6. idem [↑](#footnote-ref-6)