A GUIDE TO SQLITE_ORM FOR SQL AND C++ USERS

By Juan Dent © 2024 Version 4.1

Table of Contents

C++ for writing data intensive applications

C++ is a large language with a very expressive and rich syntax. Perhaps its most salient characteristic is its ability to control the level of abstraction enabling application programming to be done in terms of the problem domain’s concepts. An extension to this capability is the ability to define Domain Specific Languages (DSL) like SQL in terms of “ordinary” C++ code. This makes C++ a very compelling language for writing data intensive applications.

Properties of a DSL:

- It is a language, that defines: [CPPTMP,216]
    - An alphabet (set of symbols)
    - Well defined rules stating how the alphabet may be used to build well-formed compositions
    - A well-defined subset of all well-formed compositions that are assigned specific meanings
- It is domain specific not general-purpose
    - Examples include regular expressions, UML, Morse code
    - By this restriction, we gain significantly higher level of abstraction and expressiveness because
        - Specialized alphabet and notations allow pattern-matching that matches our mental model

        - Enabling writing code in terms close to the abstractions of the problem domain is the characteristic property and motivation behind all DSLs
        - We use the language's notation to write down a statement of the problem itself and the language's semantics take care of generating a solution
        - The most successful DSLs are often ***declarative*** languages providing us with notations to describe what rather than how
            - The how can be seen as a consequence of the what
- In a sense, DSL is an enhancement to object-oriented programming in which the development is done in terms of the problem domain conceptual model
    - We are just taking an extra step towards enriched notational support

This document refers to a SQL DSL called ***SQLITE_ORM***, which provides direct support for writing SQL in C++. This is indeed a worthy capability and one that allows for the clear concise creation of data intensive applications. This library is not only a SQL DSL but a sort of object relational tool (ORM[1]) in that it provides means to associate data structures in C++ with relational tables in sqlite3.
One can truly raise C++ abstraction level by thinking in a combination of imperative C++ enhanced with compile time metaprogramming and SQL. The synergy is indeed attractive and powerful.

ORM: What is it?

Object relational mapping maps the data types and entities between an object-oriented language like C++ to a relational type system as SQL. ORM is a set of technologies that allows developers to access RDBMS data from an object-oriented programming language.
Challenges:

1. The type systems are different – we must decide how we are going to map each primitive type in C++ to an equivalent type in SQL and vice versa.
    a. The main difference is that the RDBMS stores scalar values, whereas C++ can contain complex objects within. Some of these properties could be lists of other objects that are stored in another table…there is a discrepancy at the fundamental level.
    b. This is called impedance mismatch
2. Object containment: objects in C++ can contain other objects and collections of objects to any depth, while SQL requires each entity to be normalized. An example is an Invoice with its InvoiceLines which can be programed as an aggregate object in C++ but must be represented as normalized tables in SQL – one for Invoices and one for InvoiceLines. The containment is represented in the relational databases by the presence of a foreign key from InvoiceLines into Invoices. The containment in C++ may be a vector of InvoiceLines inside an Invoice object. Sqlite_orm provides a straightforward support for a transformation between them
3. Inheritance: IS-A relations between classes in C++ must be some how represented in one or more tables in the relational database. The general schemes according to recognized author Martin Fowler[2] are:
    a. Store all objects in an inheritance tree in the same table even though not all columns will be used in every row; additionally, each row may have a label that identifies the type to which it is related.

[1] Object Relational Mapper: ORM
[2] Patterns of Enterprise Application Architecture, 2003, Addison Wesley Signature Series

            i. Queries and modifications to the objects' data will be fast, but storage size will likely be wasted
    b. Store objects from each class into a separate table. Queries made at some level in the tree, must perform queries in the tables for that level and below so it will have an impact in the performance which will probably require UNIONs. Insert or update objects will go directly into the corresponding table having cero penalty in speed and wasted storage.
    c. Then there are other cases like disjoint types. Take Person and its subclasses Man and Woman: they are mutually exclusive, so Person is an abstraction which combines objects from the Man and Woman tables using a UNION.
4. SQL is a language specifically created to handle structured data following normalization of entities and it excels in this. C++ is, on the other hand, a general-purpose programming language with support for low level access as well as the ability to alter the abstraction level. This makes it a very suitable language in which to embed a Domain Specific Language (DSL) like SQL, which makes C++ persistent!

Advantages and disadvantages of ORMs in general

Some products in the industry have gone haywire crazy implementing ORM features giving them a bad reputation. Let's consider several points regarding the realities of ORMs:

1. ORMs can bring a lot of additional complexity to the table; it takes time to manage these tools, there is a learning curve
2. It can affect performance
    a. In most languages greater abstractions bring with them greater overheads
    b. C++ is a very notable exception thanks to template metaprogramming
    c. In C++ we don't pay for what we don't use
3. ORMs are not able to solve the problems completely
    a. The abstractions leak making you need to dive into SQL and try to solve some of the problems manually
    b. Of course, they leak, there is an impedance mismatch (the models are not 100% congruent or isomorphic)
4. As an extreme, one can either accept the shortcomings stated above or avoid using ORMs, falling to SQL and abandoning the object-oriented paradigm completely.
5. You still need to know how a database works and you still need to learn how to code in SQL. **An ORM is a supplement for your SQL skills, not a replacement for them**.
6. Some ORMs support object with subobjects in a way that when the outer container is loaded the inner subobjects are also loaded… this tends to bring into memory more data than may be necessary. For instance, loading an invoice, one might only need to know how many invoice lines it contains but is completely indifferent to the invoice lines themselves. This is information that is known at query time but if we use an invoice.load() it might bring all lines as well. When reading, the data is better served by using a Data Transfer Object whose structure mimics the exact data desired. In read mode, ORM can be thus inefficient. A DTO is better suited by these reads and the object model is not relevant and the ORM may be used in a query mode using specific structures or "views" for loading only the required data.

7. When writing, we know what invoice lines were modified and if the invoice itself was modified, so we can be more selective upon persisting the changes. Thus, in write mode, the object model may be very efficient and effective. ORMs excel in this mode but not in read mode except when using DTOs.
8. These 2 considerations bring us to the conclusion that object models are efficient for writing while DTOs are the better choice for reading. In other words, the ORM is more useful when writing complex object models than when reading specific data that needs to be displayed, for example.
9. Even though we have seen that ORMs do not (cannot) cover the 100% of the use cases regarding data persistence, they are still worth it, specially if we consider the differences between reading and writing modes. Complex objects can indeed benefit from the data support offered by the ORM.
10. Learning an ORM pays off for three reasons.
    a. You don't have to abandon the object model of your application
    b. The write mode is very efficient because we know what has changed while the read mode is very efficient because it uses direct DTOs to gather only the data of interest – and only loads complex compound objects when truly needed
    c. It reduces the time needed to write complex queries by allowing us to save and load structures directly
11. It's amazing how much more productive you become when you master an ORM.
12. When that ORM is created in template C++ programming, the abstractions do not hinder performance

Features of sqlite_orm:

1. Able to match C++ language with sqlite type system
    a. User defined types may be bound by implementing certain hooks in the architecture
2. Able to generate database schema from C++
    a. Makes C++ structs/classes into table representation
3. Support for SQL as a DSL[3], no need to invent a proprietary query language when a standard is available
4. Object persistence and mapping achieved through simple functions: make_storage, make_table, make_column, make_trigger, etc.
5. Support for SQL triggers, checks and unique constraints
6. Support for SQL primary and foreign keys
7. Support for table and column aliases
8. Database schema defined at compile time via metaprogramming
    a. Reduces overhead
9. Direct object CRUD[4] support
10. Support for containment and aggregation via FK-PK

[3] Does not invent its own proprietary language instead follows the industry standard SQL

[4] Create, retrieve, update and delete at the object level

11. Support for inheritance in C++: programmer can use the various schemes on top of the native 1:1 entity-table support out of the box
12. SQL like syntax
    a. Raw SQL: Not supported
    b. DB Creation and changes using only C++
    c. General SQL query support
    d. Prepared statements support
    e. Set operations support
        i. UNION, EXCEPT and INTERSECT
    f. FK support:
        i. On_update or on_delete
            1. Cascade()
            2. Restrict_()
            3. Set_null()
            4. Set_default()
            5. No_action()
    g. JOIN support
        i. NATURAL JOIN: provided
        ii. INNER JOIN: provided
        iii. LEFT OUTER JOIN: provided
        iv. RIGHT OUTER JOIN: easily implemented
        v. FULL OUTER JOIN: easily implemented
    h. Full WHERE support
        i. Relational operators
        ii. IN, NOT IN, BETWEEN, LIKE, GLOB, EXISTS
    i. Full GROUP BY/HAVING support
    j. Full DISTINCT support
    k. Full ORDER_BY and MULTI_ORDER_BY and limit, offset support
    l. Subquery support: with some limitations
    m. Full Index support
    n. Full triggers support
    o. Constraints support
        i. Check
        ii. Unique
        iii. Default values
    p. Full READ, INSERT, UPDATE, REMOVE support at the object level and at the column level

    q. Find by id
    r. Load all objects of a mapped type that support an arbitrarily complex condition
    s. Create “virtual objects” from JOINED and/or UNIONED tables
        i. Goes beyond normalized tables
    t. INSERT from SELECT
13. Supports C++ 14,17 and 20 features
    a. Code Style is standard
    b. STL container support
    c. STL iterator support
    d. No macros or external scripts: everything is C++
    e. User defined scalar and aggregate functions may be used in queries
    f. Type deduction is used everywhere
    g. Templates are predominant so code is extremely efficient
    h. Static assertions test the code at compile time as much as possible
    i. Expressive: synergy between metaprogramming and SQL high level code
14. Database concepts support:
    a. Transactions
    b. BLOB support
    c. Migrations functionality: on PLUS Version only
    d. Collate support at the mapping definition and the query level
    e. Database Limits setting/getting
15. Easy integration: Single header only
16. Only one dependency:
    a. SQLite3
17. Fast and storage friendly (small footprint)
18. In memory database support
19. Persistent structs have no persistent code in them nor do they need to inherit from any class to become persistent
    a. Plain old C++ objects work immediately
    b. The functionality is not intrusive
20. Persist nested objects: requires some implementation[5]
21. Multiple database support: Not available
    a. Very specific support for SQLite3
    b. The advantage is this allows a very close fit, no unnecessary abstractions

[5] See examples/subentities.cpp

    c. We don’t pay for what we don’t use
22. Backups support
23. Table dropping and renaming
24. Dump query expressions into SQL string to verify what is being sent to the database engine
25. Views: in TODO list
26. WITH support and Common Table Expressions including recursion: in TODO list
27. SQL expressions can be made up of sub expressions, so one can define the WHERE clause somewhere and the column-list somewhere else and then combine the pieces
28. Inheritance support
    a. Class table inheritance pattern: supported
    b. Polymorphic collections using container of unique_ptr<>s: supported[6]
    c. Arbitrary queries on inheritance trees: supported
29. No reverse engineering from existing Db supported
30. Useful to see SQL commands being executed in debug runs: TO BE INVESTIGATED

Working level

SQLITE_ORM allows us to interact with the persistent objects in two fundamentally different styles:

- By columns (pure SQL)
- By objects (SQL mapped to structures)

It is this support for dealing with persistency at the object level that explains the ORM suffix of the library name. Basically, each normalized table in the database may be represented in an application as a struct or class in a 1-to-1 relationship. This allows us to work at a high level of abstraction. We refer to these instances as “persistent atoms” to indicate their normalization and their unbreakable nature.

On the other hand, we have access to all the power and expression of SQL by allowing us to define queries for reading or writing in terms of columns just like you would if working in a relational client, but by virtue of this library being a Domain Specific Language (DSL)[7], it allows us to use C++ code to write SQL. This document is dedicated to every user of the library and can be thought of a dictionary of sorts between SQL and SQLITE_ORM.

The object queries deal with collections of instances of the persistent table associated with the object type. The column queries accessing persistent tables, deal at the column level of the corresponding object types.

[6] As long as at least one virtual method at the root, usually the destructor
[7] See Chapter 10 of C++ Template Metaprogramming, by David Abrahams and Aleksey Gurtovoy

Mapping types to tables – making types persistent

In order to work with persistent types we need to map them to tables and columns with a call to make_storage(…) like in this example for type User:

```
struct User {
	int id = 0;
	std::string name;
};
auto storage = make_storage(
  {dbFileName},
	make_table("users",
		make_column("id", &User::id, primary_key()),
		make_column("name", &User::name)));
storage.sync_schema();    // synchronizes memory schema (called storage) with database schema
```

This creates non-nullable columns by default. To make one column nullable, say name, we must declare name to have one of these types:

- std::optional<std::string>
- std::unique_ptr<std::string>>
- std::shared_ptr<std::string>>

Another point to have in mind is that the fields may have any bindable type which includes all fundamental C++ data types, std::string and std::vector. Other types can be used but you must provide some code to make them bindable (an example is std::chrono::sys_days). In particular enumerations can be bound quite easily, for instance.

Simple query

Simple calculation:

```
// Select 10/5;
auto rows = storage.select(10/5);
```
This statement produces std::vector<int>.

```
// Select 10/5, 2*4;
auto rows = storage.select(columns(10/5, 2*4));
```
This statement produces std::vector<std::tuple<int, int>>

General SELECT Syntax:

SELECT DISTINCT column_list
FROM table_list
JOIN table ON join_condition
WHERE row_filter
ORDER BY column
LIMIT count OFFSET offset
GROUP BY column
HAVING group_filter;

One table select:

```
// SELECT name, id FROM User;
auto rows = storage.select(columns(&User::name, &User::id));
auto rows = storage.select(columns(&User::name, &User::id), from<User>());
```

These statements are equivalent and they produce a std::vector<std::tuple<std::string, int>>[8]. When the from clause is omitted there is an algorithm that detects all types present in a statement and adds all of them to the from clause. This works immediately when only one type is involved but sometimes we need to add joins to the other tables in which case it is best to use the explicit from<>().

```
// SELECT * FROM User;
auto rows = storage.select(asterisk<User>());// get all columns from User
```

Produces std::vector<std::tuple<std::string,int>>

```
auto objects = storage.get_all<User>();      // get all persistent instances of the User type
```

Produces std::vector<User>

[8] Assuming the definition of columns as not nullable; otherwise it would be std::vector<std::tuple<std::optional<std::string>,int> if we use std::optional to create the name field

When dealing with large resultsets

We don't have to load whole result set into memory! We can iterate the collections!

```
for(auto& employee: storage.iterate<Employee>()) {
      cout << storage.dump(employee) << endl;
}

for(auto& hero: storage.iterate<MarvelHero>(where(length(&MarvelHero::name) < 6))) {
      cout << "hero = " << storage.dump(hero) << endl;
}

auto view = storage.iterate(select(columns(&Doctor::id, &Doctor::name, &Visit::patientName, &Visit::vdate),
                                       left_join<Visit>(on(c(&Doctor::id) == &Visit::doctorId))));
for(auto&& row : view) {
    std::cout << get<0>(row) << "\t";
    std::cout << get<1>(row) << "\t";
    std::cout << get<2>(row) << "\t";
    std::cout << get<3>(row) << std::endl;
}
```

Combining selected columns into a struct instead of a tuple

Normal select creates a collection of tuples which might not be convenient. We can therefore create an adhoc structure using sqlite_orm's struct_ like so:

```
struct Artist {
     int id;
     std::string name;
};

struct Album {
     int id;
     int artist_id;
     std::string name;
};

struct Z {
     decltype(Album::name) album_name;
     decltype(Artist::name) artist_name;
};
// define SQL expression for ad-hoc construction of Z (in place of selected columns)
constexpr auto z_struct = struct_<Z>(&Album::name, &Artist::name);

cout << "albums.name, artists.name\n";

for(auto& row: storage.select(z_struct, join<Album>(on(c(&Album::artist_id) == &Artist::id)))) {
    cout << row.album_name << '\t' << row.artist_name << '\n';
}
```

Row is an instance of Z and we select its columns by name instead of by position, as it would have been if row was a tuple. See the example project called select that ships with sqlite_orm, in the function named_adhoc_structs().

Sorting rows

Order by

General syntax:

```
// SELECT select_list  FROM  table  ORDER BY  column_1 ASC, column_2 DESC;
```

Simple order by:

```
// SELECT "User"."first_name", "User"."last_name" FROM 'User' ORDER BY "User"."last_name" COLLATE NOCASE DESC

auto rows = storage.select(columns(&User::first_name, &User::last_name),
                          order_by(&User::last_name).desc().collate_nocase());

// SELECT "User"."id", "User"."first_name", "User"."last_name" FROM 'User'  ORDER BY "User"."last_name" DESC
auto objects = storage.get_all<User>(order_by(&User::last_name).desc());
```

If desc() is omitted the ordering is ascending by default and if we omit the collation, binary is the default.

Compound order by:

```
auto rows = storage.select(columns(&User::name, &User::age), multi_order_by(
                   order_by(&User::name).asc(),
                   order_by(&User::age).desc()));
```

```
auto objects = storage.get_all<User>(multi_order_by(
                    order_by(&User::name).asc(),
                    order_by(&User::age).desc()));
```

Dynamic Order by

Sometimes the exact arguments that determine ordering is not known until at runtime, which is why we have this alternative:

```
auto orderBy = dynamic_order_by(storage);
orderBy.push_back(order_by(&User::firstName).asc());
orderBy.push_back(order_by(&User::id).desc());
auto rows = storage.get_all<User>(orderBy);
```

Ordering by a function

```
// SELECT ename, job from EMP order by substring(job, len(job)-1,2)
auto rows = storage.select(columns(&Employee::m_ename, &Employee::m_job),
      order_by(substr(&Employee::m_job, length(&Employee::m_job) - 1, 2)));
```

Dealing with NULLs when sorting[9]

```
// SELECT "Emp"."ename", "Emp"."salary", "Emp"."comm" FROM 'Emp' ORDER BY CASE WHEN "Emp"."comm" IS NULL THEN 0 ELSE
// 1 END DESC
auto rows = storage.select(
      columns(&Employee::m_ename, &Employee::m_salary, &Employee::m_commission),
                   order_by(case_<int>()
                   .when(is_null(&Employee::m_commission), then(0))
                   .else_(1)
                   .end()).desc());
```

This can of course be simplified like this below but using case_ is more powerful (e.g. when you have more than 2 values):

```
auto rows = storage.select(columns(&Employee::m_ename, &Employee::m_salary, &Employee::m_commission),
                          order_by(is_null(&Employee::m_commission)).asc());
```

[9] See CASE in this document

Sorting on a data dependent key

```
auto rows = storage.select(columns(&Employee::m_ename, &Employee::m_salary, &Employee::m_commission),
                    order_by(case_<double>()
                    .when(is_equal(&Employee::m_job, "SalesMan"), then(&Employee::m_commission))
                    .else_(&Employee::m_salary)
                    .end()).desc());
```

Filtering Data

Select distinct

SELECT DISTINCT select_list FROM table;

```
// SELECT DISTINT(name) FROM EMP[10]
auto names = storage.select(distinct(&Employee::name));
```

result is of type std::vector<std::optional<std::string>>

```
auto names = storage.select(distinct(columns(&Employee::name)));
```

result is of type std::vector<std::tuple<std::optional<std::string>>>

Where Clause

SELECT   column_list FROM table WHERE search_condition;

Search condition can be formed from these clauses and their composition with and/or:

- WHERE column_1 = 100;
- WHERE column_2 IN (1,2,3);
- WHERE column_3 LIKE 'An%';
- WHERE column_4 BETWEEN 10 AND 20;
- WHERE expression1 Op expression2
  - Op can be any comparison operator:
    - =   (== in C++)

[10] Assume Employee::name is nullable

- !=, <>   (!= in C++)
- <
- >
- <=
- >=

```
//  SELECT COMPANY.ID, COMPANY.NAME, COMPANY.AGE, DEPARTMENT.DEPT
//  FROM COMPANY, DEPARTMENT
//  WHERE COMPANY.ID = DEPARTMENT.EMP_ID;
auto rows = storage.select(columns(&Employee::id, &Employee::name, &Employee::age, &Department::dept),
                          where(is_equal(&Employee::id, &Department::empId)));

auto rows = storage.select(columns(&Employee::id, &Employee::name, &Employee::age,&Department::dept),
                          where(c(&Employee::id) == &Department::empId));
```

composite where clause: clause1 [and|or] clause2 … and can also be && , or can also be ||.

```
auto rows = storage.select(columns(&Employee::id, &Employee::name, &Employee::age, &Department::dept),
                    where(c(&Employee::id) == &Department::empId) and c(&Employee::age) < 20);

auto objects = storage.get_all<User>(where(lesser_or_equal(&User::id, 2)
                    and (like(&User::name, "T%") or glob(&User::name, "*S")))
```

the where clause can be also be used in UPDATE and DELETE statements.

Limit

Constrain the number of rows returned (limit) by a query optionally indicating how many rows to skip (offset).

```
// SELECT column_list FROM table LIMIT row_count;
auto rows = storage.select(columns(&Employee::id, &Employee::name, &Employee::age, &Department::dept),
                          where(c(&Employee::id) == &Department::empId),
                          limit(4));

auto objects = storage.get_all<Employee>(limit(4));

// SELECT column_list FROM table LIMIT row_count OFFSET offset;
auto rows = storage.select(columns(&Employee::id, &Employee::name, &Employee::age, &Department::dept),
                          where(c(&Employee::id) == &Department::empId),
                          limit(4, offset(3)));

auto objects = storage.get_all<Employee>(limit(4, offset(3)));
```

```
// SELECT column_list FROM table LIMIT offset, row_count;

auto rows = storage.select(columns(&Employee::id, &Employee::name, &Employee::age, &Department::dept),
                          where(c(&Employee::id) == &Department::empId),
                          limit(3, 4));

auto objects = storage.get_all<Employee>(limit(3, 4));

Using limit with order by:

// get the 2 employees with the second and third higher salary
auto rows = storage.select(columns(&Employee::name, &Employee::salary), order_by(&Employee::salary).desc(),
                          limit(2, offset(1)));

auto objects = storage.get_all<Employee>(order_by(&Employee::salary).desc(), limit(2, offset(1)));
```

Between Operator

Logical operator that tests whether a value is inside a range of values including the boundaries.
NOTE: BETWEEN can be used in the WHERE clause of the SELECT, DELETE, UPDATE and REPLACE statements.

```
// Syntax:
// test_expression BETWEEN low_expression AND high_expression

// SELECT DEPARTMENT_ID FROM departments
// WHERE manager_id
// BETWEEN 100 AND 200
auto rows = storage.select(&Department::id, where(between(&Department::managerId, 100, 200)));

// SELECT DEPARTMENT_ID FROM departments
// WHERE DEPARTMENT_NAME
// BETWEEN “D” AND “F”
auto rows = storage.select(&Department::id, where(between(&Department::dept, "D", "F")));

auto objects = storage.get_all<Department>(where(between(&Department::dept, "D", "F")));
```

In

Whether a value matches any value in a list or subquery, syntax being:

expression [NOT] IN (value_list|subquery);

```
//  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 IN (10,20,30)
std::vector<int> ids{ 10,20,30 };
auto rows = storage.select(columns(&Employee::firstName, &Employee::departmentId),
                           where(in(&Employee::departmentId, ids)));

auto objects = storage.get_all<Employee>(where(in(&Employee::departmentId, {10,20,30} )));

//  SELECT first_name, last_name, department_id
//  FROM employees
//  WHERE department_id NOT IN (10,20,30)
std::vector<int> ids{ 10,20,30 };
auto rows = storage.select(columns(&Employee::firstName, &Employee::departmentId),
                           where(not_in(&Employee::departmentId, ids)));

auto objects = storage.get_all<Employee>(where(not_in(&Employee::departmentId, { 10,20,30 })));
```

Like

Matches a pattern using 2 wildcards: % and _.
% matches 0 or more characters while _ matches any character. For characters in the ASCII range, the comparison is case insensitive; otherwise it is case sensitive.

SELECT   column_list FROM table_name WHERE column_1 LIKE pattern;

```
auto whereCondition = where(like(&User::name, "S%"));
auto users = storage.get_all<User>(whereCondition);

auto rows = storage.select(&User::id, whereCondition);
auto rows = storage.select(like("ototo", "ot_to"));
auto rows = storage.select(like(&User::name, "%S%a"));
auto rows = storage.select(like(&User::name, "^%a").escape("^"));
```

Glob

Similar to the like operator but using UNIX wildcards like so:

- The asterisk (*) matches any number of characters (pattern Man* matches strings that start with Man)
- The question mark (?) matches exactly one character (pattern Man? matches strings that start with Man followed by any character)
- The list wildcard [] matches one character from the list inside the brackets. For instance [abc] matches either an a, a b or a c.
- The list wildcard can use ranges as in [a-zA-Z0-9]
- By using ^, we can match any character except those in the list ([^0-9] matches any non-numeric character).

```
auto rows = storage.select(columns(&Employee::lastName), where(glob(&Employee::lastName, "[^A-J]*")));
auto employees = storage.get_all<Employee>(where(glob(&Employee::lastName, "[^A-J]*")));
```

IS NULL

```
//  SELECT
//      artists.ArtistId,
//      albumId
//  FROM
//      artists
//  LEFT JOIN albums ON albums.artistid = artists.artistid
//  WHERE
//      albumid IS NULL;
auto rows = storage.select(columns(&Artist::artistId, &Album::albumId),
                    left_join<Album>(on(c(&Album::artistId) == &Artist::artistId)),
                    where(is_null(&Album::albumId)));
```

Dealing with NULL values in columns

```
// Transforming null values into real values
// SELECT COALESCE(comm,0), comm FROM EMP
auto rows = storage.select(columns(coalesce<double>(&Employee::m_commission, 0), &Employee::m_commission));
```

Joining tables

Table expressions are divided into join and nonjoin table expressions:

Table-expressions := join-table-expression | nonjoin-table-expression
Join-table-expression := table-reference CROSS JOIN table-reference
| table-reference [NATURAL] [join-type] JOIN table-reference [ON conditional-expression] | USING (column-commalist)]
nonjoin-table-exp
ression := table-reference UNION table-reference | table-reference UNION ALL table-reference

SQLite join

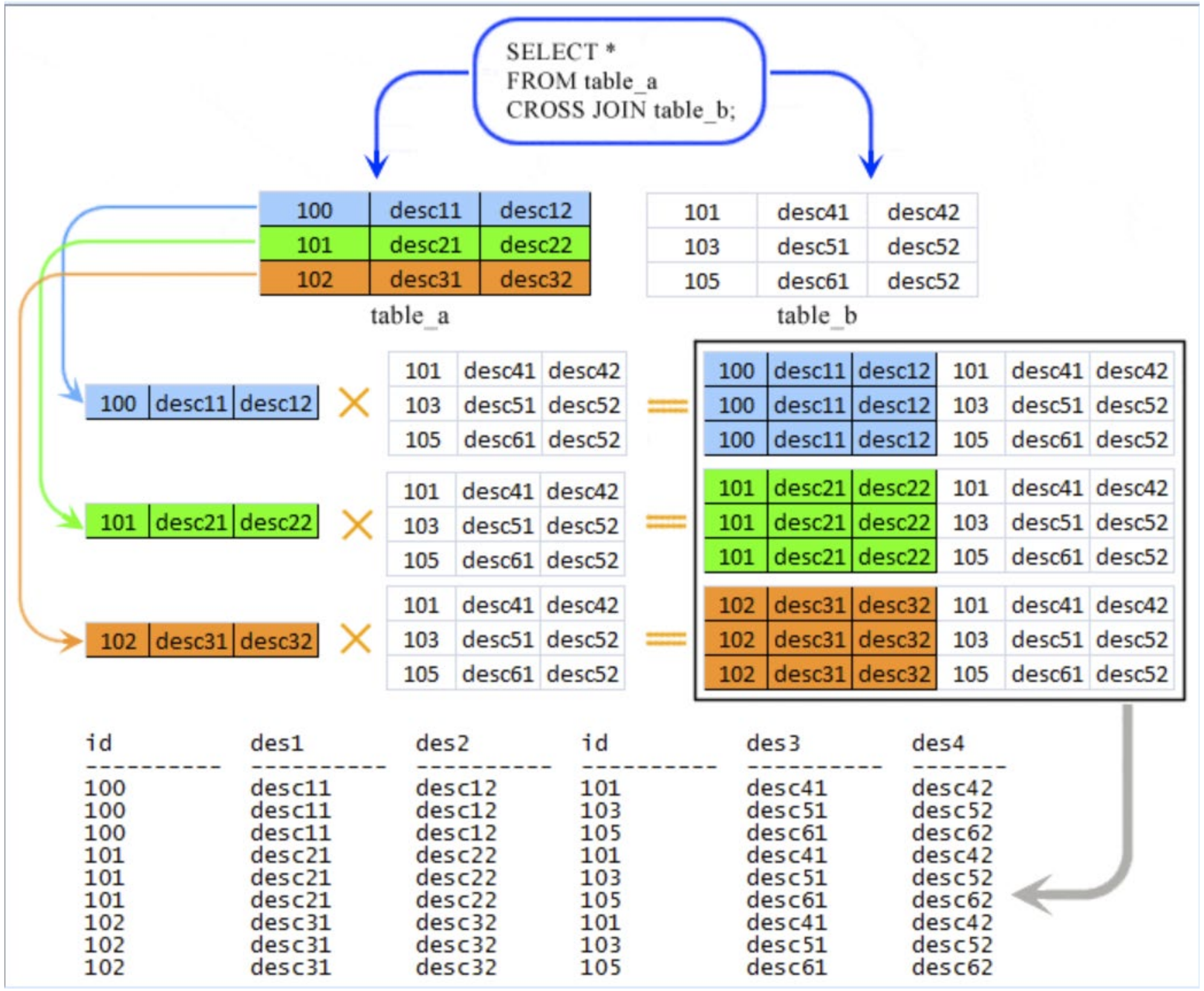
In SQLite to query data from more than one table you can use INNER JOIN, LEFT JOIN or CROSS JOIN[11]. Each clause determines how rows from one table are "linked" to rows in another table. There is no explicit support for RIGHT JOIN or FULL OUTER JOIN. The expression OUTER is optional and does not alter the definition of the JOIN.

Cross join

Cross join is more accurately called the extended Cartesian product. If A and B are the tables from evaluation of the 2 table references then A CROSS JOIN B evaluates to a table consisting of all possible rows R such that R is the concatenation of a row from A and a row from B. In fact, the A CROSS JOIN B join expression is semantically equivalent to the following select-expression:

(SELECT A.*, B.* FROM A,B)

[11] You can also use UNION or UNION ALL – see later in this document


```sql
SELECT *
FROM table_a
CROSS JOIN table_b;
```

```
id          des1        des2        id          des3        des4
----------  ----------  ----------  ----------  ----------  -------
100         desc11      desc12      101         desc41      desc42
100         desc11      desc12      103         desc51      desc52
100         desc11      desc12      105         desc61      desc62
101         desc21      desc22      101         desc41      desc42
101         desc21      desc22      103         desc51      desc52
101         desc21      desc22      105         desc61      desc62
102         desc31      desc32      101         desc41      desc42
102         desc31      desc32      103         desc51      desc52
102         desc31      desc32      105         desc61      desc62
```

(taken from [SQLite CROSS JOIN - w3resource](#))

Other joins

Table-reference [NATURAL] [join-type] JOIN table-reference
    [ON conditional-expression | using(column-commalist)]

Join type can be any of

- INNER[12]
- LEFT [OUTER]
- RIGHT [OUTER]
- FULL [OUTER]
- UNION[13]

With the following restrictions:

- NATURAL and UNION cannot both be specified
- If either NATURAL or UNION is specified, neither an ON clause nor a USING clause can be specified
- If neither NATURAL nor UNION is specified, then either an ON clause or a USING clause must be specified
- If join-type is omitted, INNER is assumed by default

It is important to realize that OUTER in LEFT, RIGHT and FULL has no effect on the overall semantics of the expression and is thus completely unnecessary. LEFT, RIGHT, FULL and UNION all have to do with NULLs so let's examine the other ones first:

1. Table-reference JOIN table-reference ON conditional-expression
2. Table-reference JOIN table-reference USING (column-commalist)
3. Table-reference NATURAL JOIN table-reference

Case 1 is equivalent to the following select-expression where cond is the conditional-expression:
(SELECT A.*, B.* FROM A,B WHERE cond)

In case 2, let the commalist of columns in the USING clause be unqualified C1, C2, .., Cn, then it is equivalent to a case 1 with the following ON clause:
ON A.C1 = B.C1 AND A.C2 = B.C2 And … A.Cn = B.Cn.

Finally case 3 is equivalent to case 2 where the USING clause contains all the columns that have the same names in A and B.

Joins having to do with NULLs (i.e. OUTER JOINS)

In the INNER joins, when we try to construct the ordinary join of 2 tables A and B, then any row that matches no row in the other table (under the relevant join condition) does not participate in the result. In an outer join such a row participates in the result: it appears exactly once, and the column positions that would have been filled with values from the other table, if such a mapping row had in fact existed, are filled with nulls instead. Therefore the outer join preserves nonmatching rows in the result whereas the inner join excludes them.

[12] Default value if non specified (i.e., INNER JOIN is equivalent to JOIN)
[13] We examine this concept later in this document

A LEFT OUTER JOIN of A and B, preserves rows from A with no matching rows from B. A RIGHT OUTER JOIN of A and B, preserves rows from B with no matching rows from A. A FULL OUTER JOIN preserves both. Lets analyze the particular cases for LEFT OUTER JOIN being that the other cases are similar:
We have three options in which to write our LEFT JOIN:

1. Table-reference LEFT JOIN table-reference ON conditional-expression
2. Table-reference LEFT JOIN table-reference USING (column-commalist)
3. Table-reference NATURAL LEFT JOIN table-reference

Case 1 can be represented as the following select statement:

SELECT A.*, B.* FROM A INNER JOIN B ON B.fk = A.pkey
UNION ALL
SELECT A.*, NULL, NULL, …,NULL FROM A WHERE A.pkey NOT IN (SELECT A.pkey FROM A INNER JOIN B ON B.fk = A.pkey)

Code:

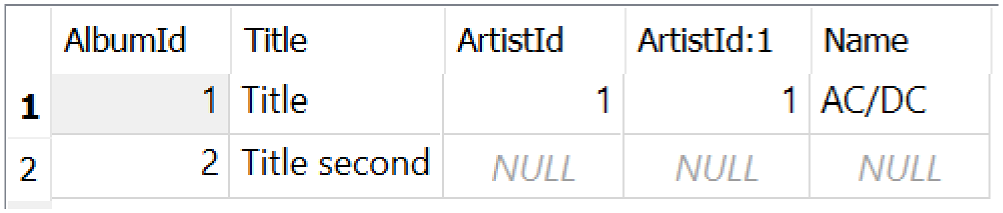
```
constexpr std::optional<int> null_int;
auto rows = storage.select(union_all(select(columns(asterisk<Employee>(), &EmploymentHistory::id,
&EmploymentHistory::fk_employee), from<Employee>(),
inner_join<EmploymentHistory>(on(c(&EmploymentHistory::fk_employee) == &Employee::id))),
select(columns(asterisk<Employee>(), null_int, null_int), from<Employee>(), where(not_in(&Employee::id,
select(&Employee::id, from<Employee>(), inner_join<EmploymentHistory>(on(c(&EmploymentHistory::fk_employee) ==
&Employee::id))))))));
```

Which means the UNION ALL of (a) the corresponding inner join and (b) the collection of rows excluded from the inner join, where there are as many NULL columns as there are columns in B.

For case 2, let the commalist of columns in the USING clause be C1, C2,…, Cn, all Ci unqualified and identifying a common column of A and B. Then the case becomes identical to a case 1 in which the condition has the form:
ON (A.C1 = B.C1 AND A.C2 = B.A2, …, A.Cn = B.Cn)

For case 3, the commalist of colums to be used for case 2 is the collection of all common columns from A and B.

Example for Left Join:



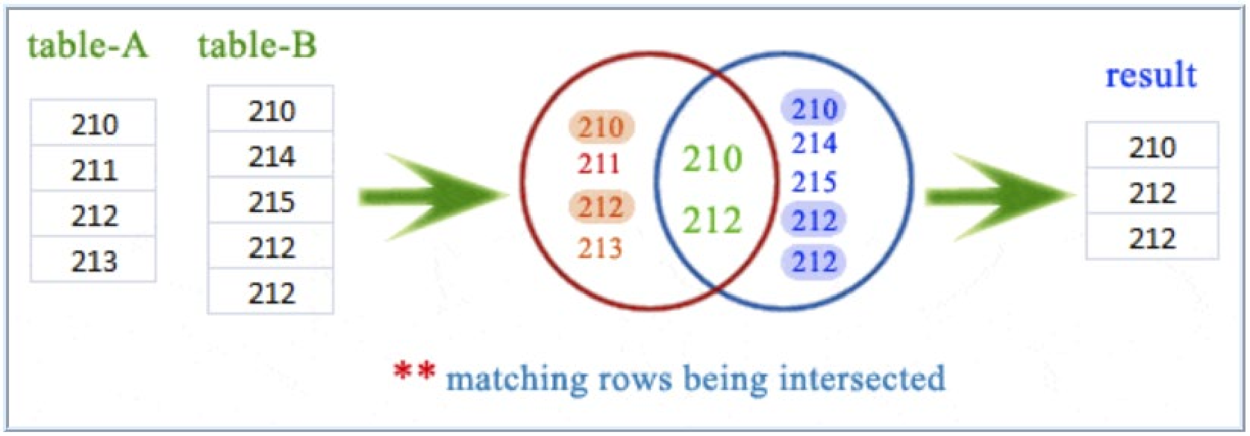
	AlbumId	Title	ArtistId	ArtistId:1	Name
1	1	Title	1	1	AC/DC
2	2	Title second	NULL	NULL	NULL

```
//  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));
```

Example for Inner Join

```
//  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)));
```

In this example, each row from tracks table is matched with a row from albums table according to the on clause. When this clause is true, then columns from the corresponding tables are displayed as an “extended row” – we are actually creating an anonymous type with attributes from the joined tables. The relationship between these tables is N tracks per 1 album. All N tracks with the same albumId are joined with the 1 album with matching columns as per the on clause.



Example for Natural Join

```
//  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"));
```

Self join

```
//  SELECT m.FirstName || ' ' || m.LastName,
//      employees.FirstName || ' ' || employees.LastName
//  FROM employees
//  INNER JOIN employees m
//  ON m.ReportsTo = employees.EmployeeId
```

Legacy

```
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(" ") || c(&Employee::lastName)),
            inner_join<als>(on(alias_column<als>(&Employee::reportsTo) == c(&Employee::employeeId))));
```

Modern: 2 options

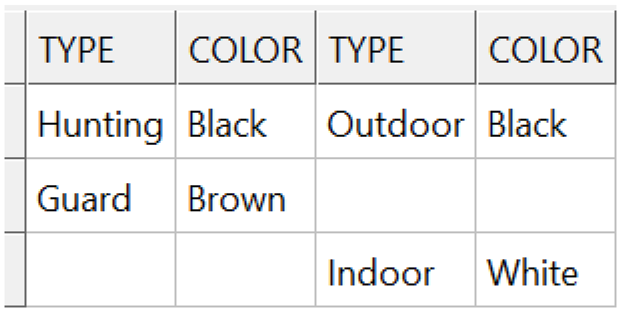
```
inline constexpr sqlite_orm::orm_table_reference auto employee = sqlite_orm::c<Employee>();
constexpr orm_table_alias auto m = "m"_alias.for_<Employee>();

auto firstNames = storage.select(columns(m->*&Employee::firstName || " " || m->*&Employee::lastName,
                                 employee->*&Employee::firstName || " " || employee->*&Employee::lastName),
                                 inner_join<m>(on(m->*&Employee::reportsTo == employee->*&Employee::employeeId)));


constexpr orm_table_alias auto emp = custom_alias<Employee>{};
auto firstNames = storage.select(columns(emp->*&Employee::firstName || " " || emp->*&Employee::lastName,
                              employee->*&Employee::firstName || " " || employee->*&Employee::lastName),
                              inner_join<emp>(on(emp->*&Employee::reportsTo == employee->*&Employee::employeeId)));
```

Full outer join

While SQLite does not support FULL OUTER JOIN, it is very easy to simulate it. Take these 2 classes/tables as an example, insert some data and do the "full outer join":

```
struct Dog
{
	std::optional<std::string> type;
	std::optional<std::string> color;
};

struct Cat
{
	std::optional<std::string> type;
	std::optional<std::string> color;
};

using namespace sqlite_orm;

auto storage = make_storage(
        { "full_outer.sqlite" },
        make_table("Dogs", make_column("type", &Dog::type), make_column("color", &Dog::color)),
        make_table("Cats", make_column("type", &Cat::type), make_column("color", &Cat::color)));

storage.sync_schema();
storage.remove_all<Dog>();
storage.remove_all<Cat>();

storage.insert(into<Dog>(), columns(&Dog::type, &Dog::color), values(
                                std::make_tuple("Hunting", "Black"), std::make_tuple("Guard", "Brown")));

storage.insert(into<Cat>(), columns(&Cat::type, &Cat::color), values(
                                std::make_tuple("Indoor", "White"), std::make_tuple("Outdoor", "Black")));
```

```
// FULL OUTER JOIN simulation:

//     SELECT d.type,
//     d.color,
//     c.type,
//     c.color
//     FROM dogs d
//     LEFT JOIN cats c USING(color)
//     UNION ALL
//     SELECT d.type,
//    d.color,
//    c.type,
//    c.color
//    FROM cats c
//    LEFT JOIN dogs d USING(color)
//    WHERE d.color IS NULL;

auto rows = storage.select(
        union_all(select(columns(&Dog::type, &Dog::color, &Cat::type, &Cat::color),
            left_join<Cat>(using_(&Cat::color))),
            select(columns(&Dog::type, &Dog::color, &Cat::type, &Cat::color), from<Cat>(),
                left_join<Dog>(using_(&Dog::color)), where(is_null(&Dog::color)))));
```



TYPE	COLOR	TYPE	COLOR
Hunting	Black	Outdoor	Black
Guard	Brown		
		Indoor	White

Grouping data

The group by clause is an optional clause of the select statement and enables us to take a selected group of rows into summary rows by values of one or more columns. It returns one row for each group and it is possible to apply an aggregate function such as MIN,MAX,SUM,COUNT or AVG – or one that you program yourself in sqlite_orm[14]!

The syntax is:

SELECT column_1, aggregate_function(column_2) FROM table GROUP BY column_1, column_2;

Group by

```
//  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));
```

Group by date example:

```
// SELECT (STRFTIME(“%Y”, "Invoices"."invoiceDate")) AS InvoiceYear,
// (COUNT("Invoices"."id")) AS InvoiceCount FROM 'Invoices' GROUP BY InvoiceYear
// ORDER BY InvoiceYear DESC
struct InvoiceYearAlias : alias_tag {
      static const std::string& get() {
            static const std::string res = "INVOICE_YEAR";
            return res;
      }
};

auto rows = storage.select(columns(as<InvoiceYearAlias>(strftime("%Y", &Invoice::invoiceDate)),
as<InvoiceCountAlias>(count(&Invoice::id))), group_by(get<InvoiceYearAlias>()),
order_by(get<InvoiceYearAlias>()).desc());
```

[14] Take a peek at `create_scalar_function` and `create_aggregate_function` for details on how to define your custom functions in sqlite_orm

Modern

```
constexpr auto i = "i"_col;
constexpr auto j = "j"_col;


auto rows = storage.select(columns( as<i>( strftime("%Y", &Invoice::invoiceDate), as<j>(count(&Invoice::id))),
group_by(i), order_by(i).desc());
```

Having

While the where clause restricts the rows selected, the having clause selects data at the group level. For instance:

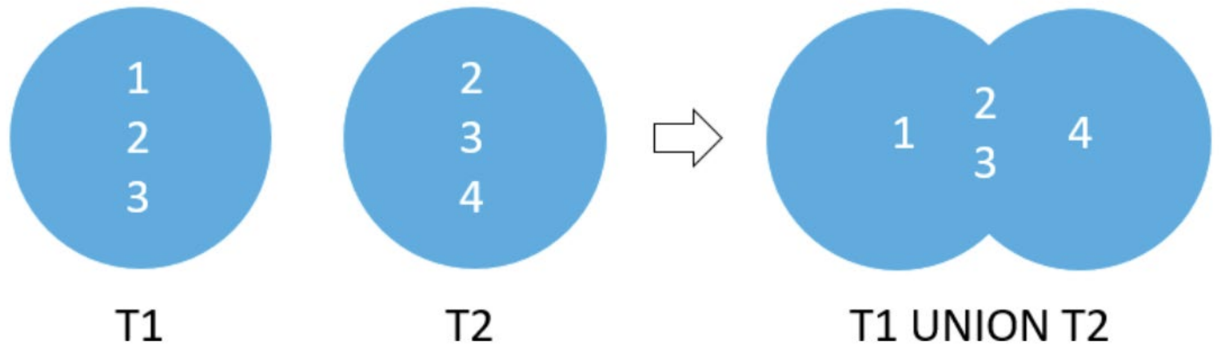
```
//  SELECT NAME, SUM(SALARY)
//  FROM COMPANY
//  WHERE NAME is like "%l%"
//  GROUP BY NAME
//  HAVING SUM(SALARY) > 10000
auto namesWithHigherSalaries = storage.select(columns(&Employee::name, sum(&Employee::salary)),
                                    where(like(&Employee::name, "%l%")),
                                    group_by(&Employee::name).having(sum(&Employee::salary) > 10000));
```

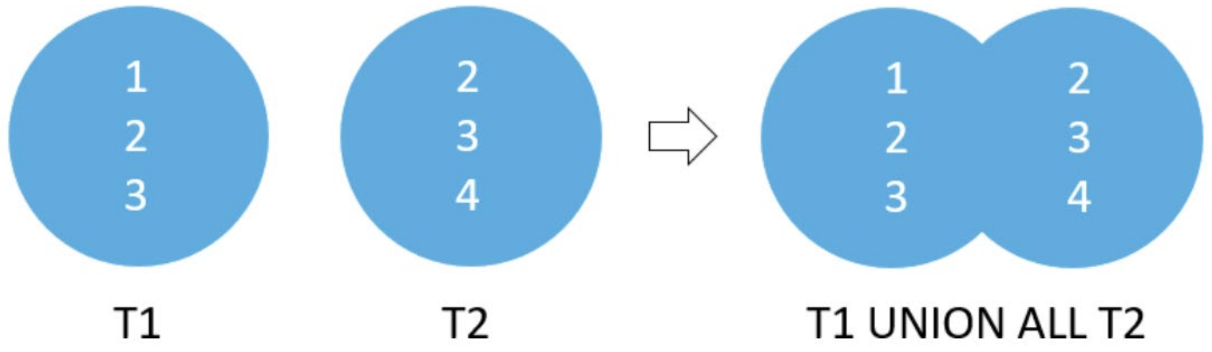
Set operators

Union

The difference between UNION and JOIN Is that the JOIN clause combines columns from multiple related tables while UNION combines rows from multiple similar tables. The UNION operator removes duplicate rows, whereas the UNION ALL operator does not. The rules for using UNION are as follows:

- Number of columns in all queries must be the same
- The corresponding columns must have compatible data types
- The column names of the first query determine the column names of the combined result set
- The group by and having clauses are applied to each individual query, not the final result set
- The order by apply to the combined result set, not within the individual result set





```
//  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))))));

Union ALL with order by:

//  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
//  ORDER BY NAME
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))),
                   order_by(&Employee::name))));
```

Stacking one resultset on top of another

```
// SELECT "Dept"."deptname" AS ENAME_AND_DNAME, "Dept"."deptno" FROM 'Dept'
// UNION ALL
// SELECT (QUOTE("------------------")), NULL
// UNION ALL
// SELECT "Emp"."ename" AS ENAME_AND_DNAME, "Emp"."deptno" FROM 'Emp'
auto rows = storage.select(
             union_all(
                    select(columns(as<NamesAlias>(&Department::m_deptname), &Department::m_deptno)),
                    select(union_all(
                          select(columns(quote("--------------------"), std::optional<int>())),
                          select(columns(as<NamesAlias>(&Employee::m_ename), &Employee::m_deptno))))));
```

Except

Compares the result sets of 2 queries and retains rows that are present only in the first result set. These are the rules:

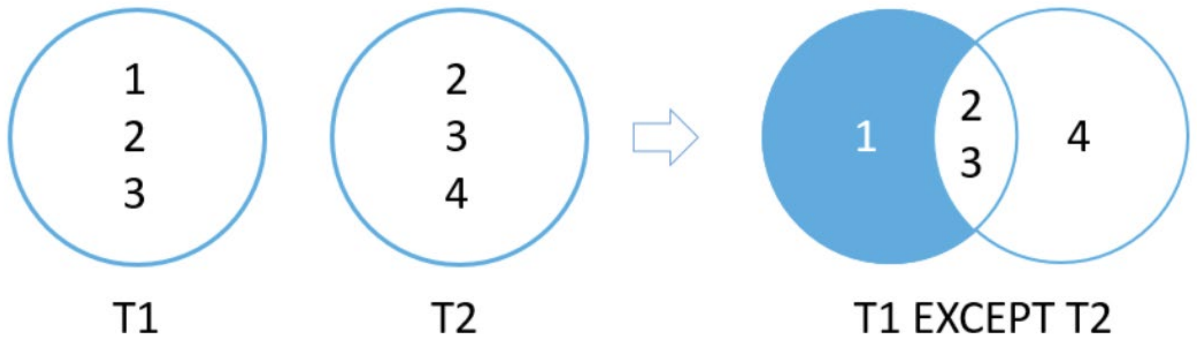
- Number of columns in each query must be the same
- The order of the columns and their types must be comparable

Find all the dept_id in dept_master but not in emp_master:

```
//  SELECT dept_id
//  FROM dept_master
//  EXCEPT
//  SELECT dept_id
//  FROM emp_master
auto rows = storage.select(except(select(&DeptMaster::deptId), select(&EmpMaster::deptId)));
```

Find all artists ids of artists who do not have any album in the albums table:

```
// SELECT ArtistId FROM artists EXCEPT SELECT ArtistId FROM albums;
auto rows = storage.select(except(select(&Artist::m_id), select(&Album::m_artist_id)));
```



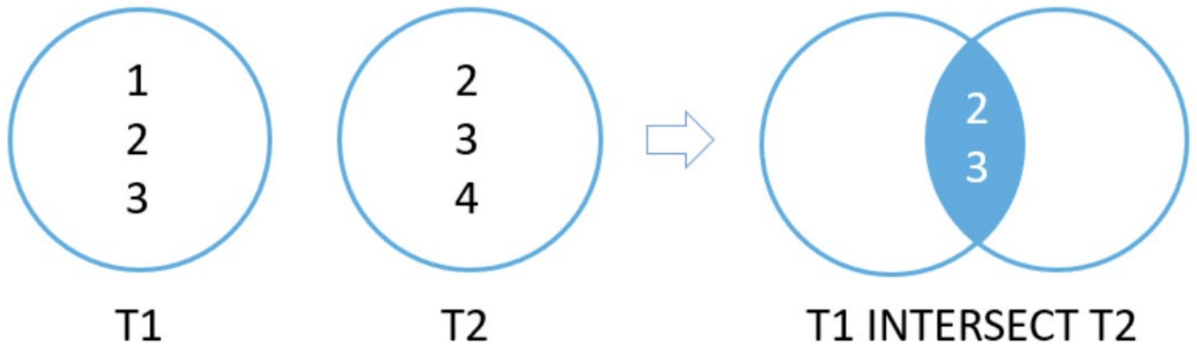
Intersect

Compares the result sets of 2 queries and returns distinct rows that are output by both queries. Syntax:

SELECT select_list1 FROM table1 INTERSECT SELECT select_list2 FROM table2

These are the rules:

- Number of columns in each query must be the same
- The order of the columns and their types must be comparable



```
//  SELECT dept_id
//  FROM dept_master
//  INTERSECT
//  SELECT dept_id
//  FROM emp_master
auto rows = storage.select(intersect(select(&DeptMaster::deptId), select(&EmpMaster::deptId)));

// SELECT "Emp"."empno", "Emp"."ename", "Emp"."job", "Emp"."salary", "Emp"."deptno" FROM 'Emp' WHERE
// (("Emp"."ename", "Emp"."job", "Emp"."salary") IN (
// SELECT "Emp"."ename", "Emp"."job", "Emp"."salary" FROM 'Emp'
// INTERSECT
// SELECT "Emp"."ename", "Emp"."job", "Emp"."salary" FROM 'Emp' WHERE (("Emp"."job" = “Clerk”))))
auto rows = storage.select(columns
(&Employee::m_empno, &Employee::m_ename, &Employee::m_job, &Employee::m_salary, &Employee::m_deptno),
      where(c(std::make_tuple( &Employee::m_ename, &Employee::m_job, &Employee::m_salary))
      .in(intersect(
            select(columns(&Employee::m_ename, &Employee::m_job, &Employee::m_salary)),
            select(columns(&Employee::m_ename, &Employee::m_job, &Employee::m_salary),
                  where(c(&Employee::m_job) == "Clerk")
)))));
```

which of course can be simplified to:

```
// SELECT "Emp"."empno", "Emp"."ename", "Emp"."job", "Emp"."salary", "Emp"."deptno" FROM 'Emp'
// WHERE(("Emp"."ename", "Emp"."job", "Emp"."salary")
// IN(SELECT "Emp"."ename", "Emp"."job", "Emp"."salary" FROM 'Emp' WHERE(("Emp"."job" = “Clerk”))))
auto rows = storage.select(columns(
            &Employee::m_empno, &Employee::m_ename, &Employee::m_job, &Employee::m_salary, &Employee::m_deptno),
            where(
                  in(std::make_tuple(&Employee::m_ename, &Employee::m_job, &Employee::m_salary),
                  select(columns(&Employee::m_ename, &Employee::m_job, &Employee::m_salary),
                        where(c(&Employee::m_job) == "Clerk")))));
```

To find the customers who have invoices:

SELECT CustomerId FROM customers INTERSECT SELECT CustomerId FROM invoices ORDER BY CustomerId;

Subquery

Subquery

A subquery is a nested SELECT within another statement such as:

SELECT column_1 FROM table_1 WHERE column_1 = (SELECT column_1 FROM table_2);

Source SQL:

```
//  SELECT first_name, last_name, salary
//  FROM employees
//  WHERE salary >(
//          SELECT salary
//          FROM employees
//          WHERE first_name='Alexander');
```

Code:

```
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"))))));
```

Source SQL:

```
//  SELECT employee_id,first_name,last_name,salary
//  FROM employees
//  WHERE salary > (SELECT AVG(SALARY) FROM employees);
```

Code:

```
auto rows = storage.select(columns(
      &Employee::id, &Employee::firstName, &Employee::lastName, &Employee::salary),
      where(greater_than(&Employee::salary,
             select(avg(&Employee::salary)))));
```

Source SQL:

```
//  SELECT first_name, last_name, department_id
//  FROM employees
//  WHERE department_id IN
//      (SELECT DEPARTMENT_ID FROM departments
//      WHERE location_id=1700);
```

Code:

```
auto rows = storage.select(
            columns(&Employee::firstName, &Employee::lastName, &Employee::departmentId),
            where(in(
                   &Employee::departmentId,
                   select(&Department::id, where(c(&Department::locationId) == 1700)))));
```

Source SQL:

```
//  SELECT first_name, last_name, department_id
//  FROM employees
//  WHERE department_id IN (10,20,30)
```

Code:

```
std::vector<int> ids{ 10,20,30 };
auto rows = storage.select(columns(&Employee::firstName, &Employee::departmentId),
                          where(in(&Employee::departmentId, ids)));
```

Source SQL:

```
//  SELECT * FROM employees
//  WHERE department_id IN (10,20,30)
```

Code:

```
auto objects = storage.get_all<Employee>(where(in(&Employee::departmentId, {10,20,30} )));
```

Source SQL:

```
//  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);
```

Code:

```
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))))));
```

Source SQL:

```
SELECT "e"."lastName", "e"."salary", "e"."departmentId" FROM "employees" "e" WHERE ("e"."salary" > (SELECT
AVG("employees"."salary") FROM "employees", "employees" "e" WHERE ("employees"."departmentId" =
"e"."departmentId")))
```

Code:

```
constexpr auto e_als = "e"_alias.for_<Employee>();
auto rows = storage.select(
        columns( e_als->*&Employee::lastName,
            e_als->*&Employee::salary,
            e_als->*&Employee::departmentId),
        from<e_als>(),
        where(greater_than(
            e_als->*&Employee::salary,
            select(avg(&Employee::salary),
                where(is_equal(&Employee::departmentId, e_als->*&Employee::departmentId))))));
```

Source SQL:

```
//  SELECT first_name, last_name, employee_id, job_id
//  FROM employees
//  WHERE 1 <=
//      (SELECT COUNT(*) FROM Job_history
//      WHERE employee_id = employees.employee_id);
```

Code:

```
auto rows = storage.select(
      columns(&Employee::firstName, &Employee::lastName, &Employee::id, &Employee::jobId), from<Employee>(),
            where(lesser_or_equal(
                  1,
                  select(count<JobHistory>(), where(is_equal(&Employee::id, &JobHistory::employeeId))))));
```

SELECT albumid, title, (SELECT count(trackid) FROM tracks WHERE tracks.AlbumId = albums.AlbumId)   tracks_count FROM albums ORDER BY tracks_count DESC;

Exists

Logical operator that checks whether subquery returns any rows. The subquery is a select statement that returns 0 or more rows. Syntax:

EXISTS (subquery)

```
//  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),
      from<Agent>(),
            where(exists(select(asterisk<Customer>(), from<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 "orders"."AGENT_CODE", "orders"."ORD_NUM", "orders"."ORD_AMOUNT", "orders"."CUST_CODE", 'c'."PAYMENT_AMT"
// FROM 'orders' INNER JOIN  'customer' 'c' ON('c'."AGENT_CODE" = "orders"."AGENT_CODE")
// WHERE(NOT(EXISTS
// (
//    SELECT 'd'."AGENT_CODE" FROM 'customer' 'd' WHERE((('c'."PAYMENT_AMT" = 7000) AND('d'."AGENT_CODE" =
//           'c'."AGENT_CODE")))))
// )
// ORDER BY 'c'."PAYMENT_AMT"

using als = alias_c<Customer>;
using als_2 = alias_d<Customer>;

double amount = 2000;
auto where_clause = select(alias_column<als_2>(&Customer::agentCode), from<als_2>(),
      where(is_equal(alias_column<als>(&Customer::paymentAmt), std::ref(amount)) and
      (alias_column<als_2>(&Customer::agentCode) == c(alias_column<als>(&Customer::agentCode)))));

amount = 7000;

auto rows = storage.select(columns(
      &Order::agentCode, &Order::num, &Order::amount,&Order::custCode,alias_column<als>(&Customer::paymentAmt)),
      from<Order>(),
      inner_join<als>(on(alias_column<als>(&Customer::agentCode) == c(&Order::agentCode))),
      where(not exists(where_clause)), order_by(alias_column<als>(&Customer::paymentAmt)));
```

Common Table Expressions

Introduction

[15]This will be mainly pure SQL to begin with. Lets take the simplest query:

```
sqlite> SELECT 1;
```

```
1
```

It returns one column and one row. Then create the simplest subquery:

```
sqlite> SELECT * FROM ( SELECT 1 );
```

```
1
```

It returns all rows and columns in the subquery – in this case only one row and one column.

A Common Table Expression is basically the same as a subquery, except assigned a name and defined prior to the query in which it's referenced. The simplest CTE version of the previous example would be:

```
sqlite> WITH one AS ( SELECT 1 )SELECT * FROM one;
```

```
1
```

- We have defined a CTE as one
- We have filled the CTE with the rows from SELECT 1 – just one row
- We selected everything from one
- Final result is a single value, 1

A CTE can have multiple columns and can be assigned names in case we want to refer to them later

```
sqlite> WITH twoCol( a, b ) AS ( SELECT 1, 2 )
```

[15] This introduction taken from https://use.expensify.com/blog/the-simplest-sqlite-common-table-expression-tutorial

```
        SELECT a, b FROM twoCol;
1|2
```

The result is one row with 2 columns.

A CTE can query other tables

```
sqlite> CREATE TABLE foo ( bar INTEGER );
sqlite> INSERT INTO foo VALUES(1);
sqlite> INSERT INTO foo VALUES(2);
sqlite> SELECT * FROM foo;
1
2
sqlite> WITH fooCTE AS (SELECT * FROM foo)
        SELECT * FROM fooCTE;
1
2
```

You can also define several CTEs in a sinqle query:

```
sqlite> WITH aCTE AS (SELECT 'a'),
    bCTE AS (SELECT 'b')
    SELECT * FROM aCTE, bCTE;
a|b
```

So, common table expressions can be used to restructure a query to make it more readable, by moving the subqueries out in front.

Compound Select Statements

The real power comes when using compound select statements like UNION ALL

```
sqlite> SELECT 1, 2
    UNION ALL
    SELECT 3, 4;
1|2
3|4
```

Now look at another example of using recursion of CTE:

```
sqlite> WITH RECURSIVE infinite AS (
     SELECT 1
     UNION ALL
     SELECT * FROM infinite
```

```
        )
        SELECT * FROM infinite;
1
1
1
1
1
```

This query never finishes!! We have to force it with Ctrl-C to finish!

- WITH RECURSIVE infinite: defines a recursive CTE
- SELECT 1 seeds that CTE's output with a single row – containing 1
- UNION ALL combines the output of what is on the left with what is on the right
- SELECT * FROM infinite: selects everything currently in the common table expression
- The result is that we are defining a CTE as the union of a single row with all other rows
- We have not specified a stopping criteria via WHERE or LIMIT so it runs forever!

To make it finite, we can add a LIMIT such as:

```
sqlite> WITH RECURSIVE finite AS (
          SELECT 1
          UNION ALL
          SELECT * FROM finite LIMIT 2
```

```
      )
      SELECT * FROM finite;
1
1
```

We can now generate a table that contains the numbers 1 through 10 like this using where or limit to stop it:

```
sqlite> WITH RECURSIVE ten(x) AS (
        SELECT 1
        UNION ALL
        SELECT x+1 FROM ten WHERE x<10
      )
      SELECT * FROM ten;
1
2
3
4
```

```
5
6
7
8
9
10
```

We can generate a wide array of different data such as date ranges:

```
sqlite> WITH RECURSIVE dates(x) AS (
        SELECT '2015-01-01'
        UNION ALL
        SELECT DATE(x, '+1 MONTHS') FROM dates WHERE x<'2016-01-01'
       )
       SELECT * FROM dates;
2015-01-01
2015-02-01
```

```
2015-03-01
2015-04-01
2015-05-01
2015-06-01
2015-07-01
2015-08-01
2015-09-01
2015-10-01
2015-11-01
2015-12-01
2016-01-01
```

CTEs are a powerful programming language that can be used to perform complex transformations of data. See this example where we change a comma separated list into a table that can be joined against:

```
sqlite> WITH RECURSIVE list( element, remainder ) AS (
        SELECT NULL AS element, '1,2,3,4,5' AS remainder
        UNION ALL
SELECT
 CASE WHEN INSTR( remainder, ',' )>0 THEN
  SUBSTR( remainder, 0, INSTR( remainder, ',' ) )
 ELSE
  remainder
 END AS element,
 CASE WHEN INSTR( remainder, ',' )>0 THEN
  SUBSTR( remainder, INSTR( remainder, ',' )+1 )
 ELSE
  NULL
 END AS remainder
FROM list
WHERE remainder IS NOT NULL
)
```

```
SELECT element FROM list WHERE element IS NOT NULL;
```

```
1
2
3
4
5
```

This is explained like so:

- A table has 2 columns element and remainder. The element contains a single element of the list while remainder contains a list of everything that comes after
- We start by placing a single row with NULL in element and the list in remainder “1,2,3,4,5”
- Then we generate a new row with the first element of the list (everything before the first comma) and the remainder (everything after the first comma)
- The logic is: if there is a comma, everything before is an element and everything after is the remainder, else the whole thing is the element and the remainder is NULL
- Keep recursing until there is no more remainder
- The result set is a table with one row per element, plus one NULL element which was the seed; this is removed by the WHERE statement in the outer query

Processing hierarchical information in SQL

With CTEs we can join a table against itself as many times as it takes to complete the analysis.

```
sqlite> CREATE TABLE company ( name, approver );
sqlite> INSERT INTO company
   VALUES
```

```
        ( 'David', NULL ),
        ( 'Matt', 'David' ),
        ( 'Jason', 'David' ),
        ( 'Ryan', 'David' ),
        ( 'Mike', 'Matt' ),
        ( 'Carlos', 'Matt' ),
        ( 'Garrett', 'Jason' ),
        ( 'Puneet', 'Jason' ),
        ( 'Joanie', 'Ryan' );
sqlite> WITH RECURSIVE approvers(x) AS (
        SELECT 'Joanie'
        UNION ALL
        SELECT company.approver
        FROM company, approvers
```

```
    WHERE company.name=approvers.x
     AND company.approver IS NOT NULL
)
SELECT * FROM approvers;
```

```
Joanie
Ryan
David
```

Using CTE in sqlite_orm

Processing comma separated list of values into rows

Source SQL:

```
WITH RECURSIVE list( element, remainder ) AS (
              SELECT NULL AS element, '1,2,3,4,5' AS remainder
              UNION ALL
              SELECT
                CASE WHEN INSTR( remainder, ',' )>0 THEN
                  SUBSTR( remainder, 0, INSTR( remainder, ',' ) )
                ELSE
                  remainder
                END AS element,
                CASE WHEN INSTR( remainder, ',' )>0 THEN
                  SUBSTR( remainder, INSTR( remainder, ',' )+1 )
                ELSE
                  NULL
                END AS remainder
              FROM list
              WHERE remainder IS NOT NULL
            )
            SELECT element FROM list WHERE element IS NOT NULL;
```

CTE code:

```
        auto storage = make_storage("");
        constexpr orm_cte_moniker auto list = "list"_cte;
        constexpr orm_column_alias auto element = "element"_col;
        constexpr orm_column_alias auto remainder = "remainder"_col;
        constexpr std::optional<int> null_int;

        auto ast = with_recursive(
            list(element, remainder)
            .as(union_all(select(columns(null_int, "1,2,3,4,5")),
                select(columns(case_<std::string>()
                    .when(greater_than(instr(list->*remainder, ","), 0),
                        then(substr(list->*remainder, 0, instr(list->*remainder, ","))))
                    .else_(list->*remainder)
                    .end(),
                    case_<std::string>()
                    .when(greater_than(instr(list->*remainder, ","), 0),
                        then(substr(list->*remainder, instr(list->*remainder, ",") + 1)))
                    .else_(nullptr)
                    .end()),
                    where(is_not_null(list->*remainder))))),
            select(list->*element, where(is_not_null(list->*element))));

        auto sql = storage.dump(ast);
        auto stmt = storage.prepare(ast);
        auto rows = storage.execute(stmt);
```

Generated SQL:

WITH RECURSIVE "list"("element", "remainder") AS (SELECT null, '1,2,3,4,5' UNION ALL SELECT CASE WHEN INSTR("list"."remainder", ',') > 0 THEN SUBSTR("list"."remainder", 0, INSTR("list"."remainder", ',')) ELSE "list"."remainder" END, CASE WHEN INSTR("list"."remainder", ',') > 0 THEN SUBSTR("list"."remainder", INSTR("list"."remainder", ',') + 1) ELSE null END FROM "list" WHERE ("list"."remainder" IS NOT NULL)) SELECT "list"."element" FROM "list" WHERE ("list"."element" IS NOT NULL)

Generating integer tables

Source SQL:

```
    // variant 1, where-clause, implicitly numbered column
        //WITH RECURSIVE
        //    cnt(x) AS(VALUES(1) UNION ALL SELECT x + 1 FROM cnt WHERE x < 1000000)
        //    SELECT x FROM cnt;
```

CTE code:

```
        constexpr orm_cte_moniker auto cnt = "cnt"_cte;
        auto ast = with_recursive(
            cnt().as(union_all(select(from), select(cnt->*1_colalias + 1, where(cnt->*1_colalias < end)))),
            select(cnt->*1_colalias));
```

Generated SQL:

WITH RECURSIVE "cnt"("1") AS (SELECT 1 UNION ALL SELECT "cnt"."1" + 1 FROM "cnt" WHERE ("cnt"."1" < 10)) SELECT "cnt"."1" FROM "cnt"

Source SQL:

```
    // variant 2, limit-clause, implicit column
        //WITH RECURSIVE
        //    cnt(x) AS(
        //        SELECT 1
        //        UNION ALL
        //        SELECT x + 1 FROM cnt
        //        LIMIT 1000000
        //    )
        //    SELECT x FROM cnt;
```

CTE code:

```
        constexpr orm_cte_moniker auto cnt = "cnt"_cte;
        constexpr orm_column_alias auto x = "x"_col;
        auto ast =
            with_recursive(cnt().as(union_all(select(from >>= x), select(cnt->*x + 1, limit(end)))), select(cnt-
>*x));
```

Generated SQL:

WITH RECURSIVE "cnt"("x") AS (SELECT 1 AS "x" UNION ALL SELECT "cnt"."x" + 1 FROM "cnt" LIMIT 10) SELECT "cnt"."x" FROM "cnt"

Source SQL:

```
    // variant 3, limit-clause, explicit column spec
```

CTE code:

```
        constexpr orm_cte_moniker auto cnt = "cnt"_cte;
        constexpr orm_column_alias auto x = "x"_col;
        auto ast = with_recursive(cnt(x).as(union_all(select(from), select(cnt->*x + 1, limit(end)))), select(cnt-
>*x));
```

Generated SQL:

WITH RECURSIVE "cnt"("x") AS (SELECT 1 UNION ALL SELECT "cnt"."x" + 1 FROM "cnt" LIMIT 10) SELECT "cnt"."x" FROM "cnt"

Generating Hierarchical Queries

1. **C++ declarations (mapped objects)**

```
    struct Org {
        std::string name;
        std::optional<std::string> boss;
    };

    auto storage = make_storage("",
                                make_table<Org>("org",
                                                make_column("name", &Org::name, primary_key()),
                                                make_column("boss", &Org::boss),
                                                foreign_key(&Org::boss).references(&Org::name)));
    storage.sync_schema();

    storage.replace<Org>({"Alice", nullopt});
    storage.replace<Org>({"Bob", "Alice"});
    storage.replace<Org>({"Cindy", "Alice"});
    storage.replace<Org>({"Dave", "Bob"});
    storage.replace<Org>({"Emma", "Bob"});
    storage.replace<Org>({"Fred", "Cindy"});
    storage.replace<Org>({"Gail", "Cindy"});
```

Source SQL:

```
        //WITH RECURSIVE
        //    chain AS(
        //        SELECT * from org WHERE name = 'Fred'
        //        UNION ALL
        //        SELECT parent.* FROM org parent, chain
        //        WHERE parent.name = chain.boss
        //    )
        //    SELECT name FROM chain;
```

CTE code:

```
        constexpr orm_cte_moniker auto chain = "chain"_cte;
        constexpr orm_table_alias auto parent = "parent"_alias.for_<Org>();
        auto ast = with_recursive(
            chain().as(union_all(select(asterisk<Org>(), where(&Org::name == c("Fred"))),
                                 select(asterisk<parent>(), where(parent->*&Org::name == chain->*&Org::boss)))),
            select(chain->*&Org::name));
```

Generated SQL:

WITH RECURSIVE "chain"("name", "boss") AS (SELECT "org".* FROM "org" WHERE ("org"."name" = 'Fred') UNION ALL SELECT "parent".* FROM "chain", "org" "parent" WHERE ("parent"."name" = "chain"."boss")) SELECT "chain"."name" FROM "chain"

2. C++ Declarations (mapped objects)

```
    struct Org {
        std::string name;
        std::optional<std::string> boss;
        double height;
    };

    auto storage = make_storage("",
                                make_table<Org>("org",
                                                make_column("name", &Org::name, primary_key()),
                                                make_column("boss", &Org::boss),
                                                make_column("height", &Org::height),
                                                foreign_key(&Org::boss).references(&Org::name)));
    storage.sync_schema();

    storage.replace<Org>({"Alice", nullopt, 160.});
    storage.replace<Org>({"Bob", nullopt, 177.});
    storage.replace<Org>({"Dave", "Alice", 169.});
    storage.replace<Org>({"Cindy", "Dave", 165.});
    storage.replace<Org>({"Bar", "Bob", 159.});
```

Source SQL:

```
        //WITH RECURSIVE
        //    works_for_alice(n) AS(
        //        VALUES('Alice')
        //        UNION
        //        SELECT name FROM org, works_for_alice
        //        WHERE org.boss = works_for_alice.n
        //    )
        //    SELECT avg(height) FROM org
        //    WHERE org.name IN works_for_alice;
```

CTE code:

```
        constexpr orm_cte_moniker auto works_for_alice = "works_for_alice"_cte;
        auto ast = with_recursive(
            works_for_alice(&Org::name)
                .as(union_(select("Alice"), select(&Org::name, where(&Org::boss == works_for_alice->*&Org::name)))),
            select(avg(&Org::height), from<Org>(), where(in(&Org::name, select(works_for_alice->*&Org::name)))));
        string sql = storage.dump(ast);
        auto stmt = storage.prepare(ast);
        auto results = storage.execute(stmt);
        cout << "Average height of Alice's team: " << results.at(0) << endl;
```

Generated SQL:

WITH RECURSIVE "works_for_alice"("name") AS (SELECT 'Alice' UNION SELECT "org"."name" FROM "org", "works_for_alice" WHERE ("org"."boss" = "works_for_alice"."name")) SELECT AVG("org"."height") FROM "org" WHERE ("org"."name" IN (SELECT "works_for_alice"."name" FROM "works_for_alice"))

3. **C++ declarations:**

```
struct Family {
    std::string name;
    std::optional<std::string> mom;
    std::optional<std::string> dad;
    time_t born;
    std::optional<time_t> died;
};

auto storage = make_storage("",
                            make_table<Family>("family",
                                               make_column("name", &Family::name, primary_key()),
                                               make_column("mom", &Family::mom),
                                               make_column("dad", &Family::dad),
                                               make_column("born", &Family::born),
                                               make_column("died", &Family::died),
                                               foreign_key(&Family::mom).references(&Family::name),
                                               foreign_key(&Family::dad).references(&Family::name)));
storage.sync_schema();

storage.replace<Family>({"Grandma (Mom)", nullopt, nullopt, 0, nullopt});
storage.replace<Family>({"Granddad (Mom)", nullopt, nullopt, 1, 1});
storage.replace<Family>({"Grandma (Dad)", nullopt, nullopt, 0, 0});
storage.replace<Family>({"Granddad (Dad)", nullopt, nullopt, 1, nullopt});
storage.replace<Family>({"Mom", "Grandma (Mom)", "Granddad (Mom)", 2, nullopt});
storage.replace<Family>({"Dad", "Grandma (Dad)", "Granddad (Dad)", 3, nullopt});
storage.replace<Family>({"Alice", "Mom", "Dad", 4, nullopt});
```

Source SQL:

```
//WITH RECURSIVE
```

```
    //    parent_of(name, parent) AS
    //    (SELECT name, mom FROM family UNION SELECT name, dad FROM family),
    //    ancestor_of_alice(name) AS
    //    (SELECT parent FROM parent_of WHERE name = 'Alice'
    //        UNION ALL
    //        SELECT parent FROM parent_of JOIN ancestor_of_alice USING(name))
    //    SELECT family.name FROM ancestor_of_alice, family
    //    WHERE ancestor_of_alice.name = family.name
    //    AND died IS NULL
    //    ORDER BY born;
```

CTE code:

```
    constexpr orm_cte_moniker auto parent_of = "parent_of"_cte;
    constexpr orm_cte_moniker auto ancestor_of_alice = "ancestor_of_alice"_cte;
    constexpr orm_column_alias auto parent = "parent"_col;
    constexpr orm_column_alias auto name = "name"_col;
    auto ast = with_recursive(
        make_tuple(
            parent_of(&Family::name, parent)
                .as(union_(select(columns(&Family::name, &Family::mom)), select(columns(&Family::name, 
&Family::dad)))),
            ancestor_of_alice(name).as(
                union_all(select(parent_of->*parent, where(parent_of->*&Family::name == "Alice")),
                          select(parent_of->*parent, join<ancestor_of_alice>(using_(parent_of->*&Family::name)))))),
        select(&Family::name,
               where(ancestor_of_alice->*name == &Family::name && is_null(&Family::died)),
               order_by(&Family::born)));
```

Generated SQL:

WITH RECURSIVE "parent_of"("name", "parent") AS (SELECT "family"."name", "family"."mom" FROM "family" UNION SELECT "family"."name", "family"."dad" FROM "family"), "ancestor_of_alice"("name") AS (SELECT "parent_of"."parent" FROM "parent_of" WHERE ("parent_of"."name" = 'Alice') UNION ALL SELECT "parent_of"."parent" FROM "parent_of" JOIN "ancestor_of_alice" USING ("name")) SELECT "family"."name" FROM "ancestor_of_alice", "family" WHERE (("ancestor_of_alice"."name" = "family"."name") AND "family"."died" IS NULL) ORDER BY "family"."born"

4. **C++ declarations:**

```
struct Org {
    std::string name;
    std::optional<std::string> boss;
    double height;
};

auto storage = make_storage("",
                            make_table<Org>("org",
                                            make_column("name", &Org::name, primary_key()),
                                            make_column("boss", &Org::boss),
                                            foreign_key(&Org::boss).references(&Org::name)));
storage.sync_schema();

storage.replace<Org>({"Alice", nullopt});
storage.replace<Org>({"Bob", "Alice"});
storage.replace<Org>({"Cindy", "Alice"});
storage.replace<Org>({"Dave", "Bob"});
storage.replace<Org>({"Emma", "Bob"});
storage.replace<Org>({"Fred", "Cindy"});
storage.replace<Org>({"Gail", "Cindy"});
```

Breadth-first pattern

Source SQL:

```
    //WITH RECURSIVE
    //    under_alice(name, level) AS(
    //        VALUES('Alice', 0)
    //        UNION ALL
    //        SELECT org.name, under_alice.level + 1
    //        FROM org JOIN under_alice ON org.boss = under_alice.name
    //        ORDER BY 2
    //    )
    //    SELECT substr('..........', 1, level * 3) || name FROM under_alice;
```

CTE code:

```
    constexpr orm_cte_moniker auto under_alice = "under_alice"_cte;
    constexpr orm_column_alias auto level = "level"_col;
    auto ast =
        with_recursive(under_alice(&Org::name, level)
                           .as(union_all(select(columns("Alice", 0)),
                                         select(columns(&Org::name, under_alice->*level + 1),
                                                join<under_alice>(on(under_alice->*&Org::name == &Org::boss)),
                                     order_by(2)))),
                       select(substr("..........", 1, under_alice->*level * 3) || under_alice->*&Org::name));
    string sql = storage.dump(ast);

    auto stmt = storage.prepare(ast);
    auto results = storage.execute(stmt);

    cout << "List of organization members, breadth-first:\n";
    for(const string& name: results) {
        cout << name << '\n';
    }
```

Generated SQL:

```
WITH RECURSIVE "under_alice"("name", "level") AS (SELECT 'Alice', 0 UNION ALL SELECT "org"."name",
"under_alice"."level" + 1 FROM "org" JOIN "under_alice" ON "under_alice"."name" = "org"."boss"  ORDER BY 2) SELECT
SUBSTR('..........', 1, "under_alice"."level" * 3) || "under_alice"."name" FROM "under_alice"
```

Generated output:

```
List of organization members, breadth-first:
Alice
...Bob
...Cindy
......Dave
......Emma
......Fred
......Gail
```

Depth-first pattern

Source SQL:

```
        //WITH RECURSIVE
        //    under_alice(name, level) AS(
        //        VALUES('Alice', 0)
        //        UNION ALL
        //        SELECT org.name, under_alice.level + 1
        //        FROM org JOIN under_alice ON org.boss = under_alice.name
        //        ORDER BY 2 DESC
        //    )
        //    SELECT substr('..........', 1, level * 3) || name FROM under_alice;
```

CTE code:

```
        constexpr orm_cte_moniker auto under_alice = "under_alice"_cte;
        constexpr orm_column_alias auto level = "level"_col;
        auto ast =
            with_recursive(under_alice(&Org::name, level)
                                  .as(union_all(select(columns("Alice", 0)),
                                                select(columns(&Org::name, under_alice->*level + 1),
                                                       join<under_alice>(on(under_alice->*&Org::name == &Org::boss)),
                                                       order_by(2).desc()))),
                              select(substr("..........", 1, under_alice->*level * 3) || under_alice->*&Org::name));
```

Generated SQL:

WITH RECURSIVE "under_alice"("name", "level") AS (SELECT 'Alice', 0 UNION ALL SELECT "org"."name", "under_alice"."level" + 1 FROM "org" JOIN "under_alice" ON "under_alice"."name" = "org"."boss" ORDER BY 2 DESC) SELECT SUBSTR('..........', 1, "under_alice"."level" * 3) || "under_alice"."name" FROM "under_alice"

```
List of organization members, depth-first:
Alice
...Bob
......Dave
......Emma
...Cindy
......Fred
......Gail
```

Alternative way of writing subselects

C++ declarations:

```
    struct Employee {
        int m_empno;
        std::string m_ename;
        double m_salary;
        std::optional<double> m_commission;
    };
    auto storage = make_storage("",
                                make_table("Emp",
                                           make_column("empno", &Employee::m_empno, primary_key().autoincrement()),
                                           make_column("ename", &Employee::m_ename),
                                           make_column("salary", &Employee::m_salary),
                                           make_column("comm", &Employee::m_commission)));
    storage.sync_schema();
    storage.transaction([&storage]() {
        storage.insert<Employee>({1, "Patel", 4000, nullopt});
        storage.insert<Employee>({2, "Jariwala", 19300, nullopt});
        return true;
    });
```

Source SQL:

Original with subselect:

```
    // SELECT * FROM (SELECT ename, salary, comm AS commmission FROM emp) WHERE salary < 5000
```

Written with CTE:

```
      With sub as (SELECT ename, salary, comm as commission FROM emp) SELECT * from sub WHERE salary < 5000;
```

Generated SQL:

WITH "sub"("ename", "salary", "comm") AS (SELECT "Emp"."ename", "Emp"."salary", "Emp"."comm" FROM "Emp") SELECT "sub".* FROM "sub" WHERE ("sub"."salary" < 5000)

CTE Code:

```
    constexpr orm_cte_moniker auto sub = "sub"_cte;
    auto expression = with(sub().as(select(columns(&Employee::m_ename, &Employee::m_salary,
&Employee::m_commission))), select(asterisk<sub>(), where(sub->*&Employee::m_salary < 5000)));
```

Output:

List of employees with a salary less than 5000:
Patel

More querying techniques

Case

We can add conditional logic to a query (an if else or switch statement in C++) by using the CASE expression. There are two syntaxes available and either can have column aliases (see below).

```
CASE case_expression
    WHEN case_expression  = when_expression_1 THEN result_1
    WHEN case_expression  = when_expression_2 THEN result_2
    ...
    [ ELSE result_else ]
END
```

```
// SELECT CASE "users"."country" WHEN “USA” THEN “Domestic” ELSE “Foreign” END
// FROM 'users' ORDER BY "users"."last_name" , "users"."first_name"
auto rows = storage.select(columns(
                    case_<std::string>(&User::country)
                    .when("USA", then("Domestic"))
                    .else_("Foreign").end()),
                    multi_order_by(order_by(&User::lastName), order_by(&User::firstName)));
```

```
CASE
    WHEN when_expression_1 THEN result_1
    WHEN when_expression_2 THEN result_2
    ...
    [ ELSE result_else ]
END
```

```
//  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()));
```

Create aliases

Easiest:

```
/** @short Create a table alias.
 *
 *  Examples:
 *  constexpr auto z_alias = "z"_alias.for_<User>();
 */
/** @short Create a column alias.
 *  column_alias<'a'[, ...]> from a string literal.
 *  E.g. "a"_col, "b"_col
 */
 // table alias
constexpr orm_table_alias auto z_alias = "zink"_alias.for_<MarvelHero>();
 // column alias
constexpr orm_column_alias auto k = "kay"_col;

auto rowsss = storage.select(columns( z_alias->*&MarvelHero::name),
                                     as<k>(instr( z_alias->*&MarvelHero::abilities), "o"))),
                            where(greater_than(k, 0)),
                            order_by(k));
```

```
        auto expression = select(columns( z_alias->*&MarvelHero::name),
                                 as<k>(instr( z_alias->*&MarvelHero::abilities), "o"))),
                         where(greater_than(k, 0)),
                         order_by(k));
        auto sql = storage.dump(expression);

    Resulting SQL:
SELECT "zink"."name", INSTR("zink"."abilities", 'o') AS "kay" FROM "marvel" "zink" WHERE ("kay" > 0) ORDER BY "kay"
```

Create a column alias

Easiest:

```
        /** @short Create a column alias.
         *  column_alias<'a'[, ...]> from a string literal.
         *  E.g. "a"_col, "b"_col
         */

        constexpr orm_column_alias auto j = "j"_col;

        //  SELECT name, instr(abilities, 'o') j
        //  FROM marvel
        //  WHERE j > 0
        //  ORDER BY j
        auto rows = storage.select(columns(&MarvelHero::name, as<j>(instr(&MarvelHero::abilities, "o"))),
                           where(j > 0),
                           order_by( j ));
```

More difficult:

```
        /**
         *  column_alias<'1'[, ...]> from a numeric literal.
         *  E.g. 1_colalias, 2_colalias
         */

        //  SELECT name, instr(abilities, 'o') i
        //  FROM marvel
        //  WHERE i > 0
        //  ORDER BY i
        auto rows = storage.select(columns(&MarvelHero::name, as<1_colalias>(instr(&MarvelHero::abilities, "o"))),
                           where(greater_than(get<1_colalias>(), 0)),
                           order_by(get<1_colalias>()));
```

```
    for(auto& row: rows) {
        cout << get<0>(row) << '\t' << get<1>(row) << '\n';
    }
```

Aliases for columns and tables

For tables:

```
//  SELECT “e”."empno", “e”."ename", “e”."hiredate", "d"."deptname"
//  FROM "Dept" "d", "Emp" “e” WHERE((“e”."deptno" = "d"."mgr"))
constexpr orm_table_alias auto e = "e"_alias.for_<Employee>();
constexpr orm_table_alias auto d = "d"_alias.for_<Department>();

auto rows = storage.select(columns( e->*&Employee::m_empno),
                                    e->*&Employee::m_ename),
                                    e->*&Employee::m_hiredate),
                                    d->*&Department::m_deptname)),
where(is_equal( e->*&Employee::m_deptno, d->*&Department::m_manager)));
```

For columns:

```
struct EmployeeIdAlias : alias_tag {
      static const std::string& get() {
            static const std::string res = "EMPLOYEE_ID";
            return res;
      }
};

struct EmployeeNameAlias : alias_tag {
      static const std::string& get() {
            static const std::string res = "EMPLOYEE_NAME";
            return res;
       }
};
```

```
// SELECT "Emp"."empno" AS "EMPLOYEE_ID", "Emp"."ename" AS "EMPLOYEE_NAME", "Emp"."hiredate", "Dept"."deptname"
// FROM "Dept", "Emp" WHERE(("EMPLOYEE_ID" = "Dept"."mgr"))
auto rows = storage.select(columns(as<EmployeeIdAlias>(&Employee::m_empno),
                           as<EmployeeNameAlias>(&Employee::m_ename),
                           &Employee::m_hiredate,
                           &Department::m_deptname),
                           where(is_equal(get<EmployeeIdAlias>(), &Department::m_manager)));
```

For columns and tables:

```
      // SELECT "e"."empno" AS "EMPLOYEE_ID", "e"."ename" AS "EMPLOYEE_NAME",
      // "e"."hiredate", "d"."deptname"
      // FROM "Dept" "d", "Emp" "e" WHERE(("e"."empno" = "d"."mgr"))
      constexpr orm_table_alias auto e = "e"_alias.for_<Employee>();
      constexpr orm_table_alias auto d = "d"_alias.for_<Department>();

      auto rowsWithBothTableAndColumnAliases = storage.select(columns(
             as<EmployeeIdAlias>( e->*&Employee::m_empno),
             as<EmployeeNameAlias>( e->*&Employee::m_ename),
             e->*&Employee::m_hiredate,
             d->*&Department::m_deptname),
        where(is_equal( e->*&Employee::m_empno, d->*&Department::m_manager)));
```

Applying aliases to CASE

```
struct GradeAlias : alias_tag {
      static const std::string& get() {
            static const std::string res = "Grade";
            return res;
      }
};

//  SELECT ID, NAME, MARKS,
//      CASE
//      WHEN MARKS >=95 THEN 'A+'
//      WHEN MARKS >=90 THEN 'A'
//      WHEN MARKS >=80 THEN 'B'
//      WHEN MARKS >=70 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, 95), then("A+"))
            .when(greater_or_equal(&Student::marks, 90), then("A"))
            .when(greater_or_equal(&Student::marks, 80), then("B"))
            .when(greater_or_equal(&Student::marks, 70), then("C"))
            .else_("Sorry!! Failed")
            .end())));
```

Changing data

Inserting a single row into a table

INSERT INTO table (column1,column2 ,..) VALUES(value1, value2 ,...);

```
storage.insert(into<Invoice>(), columns(
       &Invoice::id, &Invoice::customerId, &Invoice::invoiceDate),
       values(std::make_tuple(1, 1, date("now")))));
```

Inserting an object

```
struct User {
    int id;                // primary key
    std::string name;
    std::vector<char> hash;  //  binary format
};
User alex{
        0,
        "Alex",
        {0x10, 0x20, 0x30, 0x40},
    };
alex.id = storage.insert(alex);  // inserts all non primary key columns, returns primary key when integral
```

Inserting several rows

```
storage.insert(into<Invoice>(),
       columns(&Invoice::id, &Invoice::customerId, &Invoice::invoiceDate),
       values(std::make_tuple(1, 1, date("now")),
             std::make_tuple(2, 1, date("now", "+1 year")),
             std::make_tuple(3, 1, date("now")),
             std::make_tuple(4, 1, date("now", "+1 year"))));
```

Inserting several objects via containers

If we want to insert or replace a group of persistent atoms, we can insert them into a container and provide iterators to the beginning and end of the desired range of objects, by means of the ***insert_range*** or ***replace_range*** methods of the storage type.
For instance:

```
std::vector<Department> des =
{
	Department{10, "Accounting", "New York"},
	Department{20, "Research", "Dallas"},
	Department{30, "Sales", "Chicago"},
	Department{40, "Operations", "Boston"}
};

std::vector<EmpBonus> bonuses =
{
	EmpBonus{-1, 7369, "14-Mar-2005", 1},
	EmpBonus{-1, 7900, "14-Mar-2005", 2},
	EmpBonus{-1, 7788, "14-Mar-2005", 3}
};

storage.replace_range(des.begin(), des.end());
storage.insert_range(bonuses.begin(), bonuses.end());
```

Recall that insert like statements do not set the primary keys while replace like statements copy all columns including primary keys. That should explain why we chose to replace the departments because they have explicit primary key values, and why we chose to insert the bonuses letting the database generate the primary key values.

Inserting several rows (becomes an update if primary key already exists)

```
//  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))));
```

Inserting only certain columns (provided the rest have either default_values, are nullable, are autoincrement or are generated):

```
// INSERT INTO Invoices("customerId") VALUES((2), (4), (8))
storage.insert(into<Invoice>(),
      columns(&Invoice::customerId),
      values(
             std::make_tuple(2),
             std::make_tuple(4),
             std::make_tuple(8)));
```

Inserting from select – getting rowid (since primary key is integral)

```
// INSERT INTO users SELECT "user_backup"."id", "user_backup"."name", "user_backup"."hash" FROM 'user_backup'

storage.insert(into<User>(),
        select(columns(&UserBackup::id, &UserBackup::name, &UserBackup::hash))));
auto r = storage.select(last_insert_rowid());
```

Inserting default values:

```
storage.insert(into<Artist>(), default_values());
```

Non-standard extension in SQLITE

Applies to UNIQUE, NOT NULL, CHECK and PRIMARY_KEY constraints, but not to FOREIGN KEY constraints[16].
For insert and update commands[17], the syntax is INSERT OR Y or UPDATE OR Y where Y may be any of the following algorithms and the default conflict resolution algorithm is ABORT:

- ROLLBACK:
  - Aborts current statement with SQLITE_CONSTRAINT error and rolls back the current transaction; if no transaction active then behaves as ABORT
- ABORT
  - When constraint violation occurs returns SQLITE_CONSTRAINT error and the current SQL statement undoes any changes made by it but changes caused by prior statements within the same transaction are preserved and the transaction remains active. This is the default conflict resolution algorithm.
- FAIL
  - Same as abort except that it does not undo prior changes of the current SQL statement… a foreign key constraint causes an ABORT
- IGNORE
  - Skips the one row that contains the constraint violation and continues processing subsequent rows of the SQL statement as if nothing went wrong: rows before and after the row with constraint violation are inserted or updated normally… a foreign key constraint violation causes an ABORT behavior
- REPLACE
  - When the constraint violation occurs in the UNIQUE or PRIMARY KEY types, the pre-existing rows causing the violation are deleted prior to inserting or updating the current row and the command continues executing normally. If a NOT NULL violation occurs, the NULL is replaced

[16] FK constraint violations always behave like Abort algorithm regardless of Y setting
[17] The on conflict clause is used in the create table command with the same semantics

with the default value for that column if any exists, else the ABORT algorithm is used. For CHECK or foreign key violations, the algorithm works like ABORT. For the deleted rows, the delete triggers (if any) are fired if and only if recursive triggers[18] are enabled.

```
auto rows = storage.insert(or_abort(),
      into<User>(),
      columns(&User::id, &User::name),
      values(std::make_tuple(1, "The Weeknd")));

auto rows = storage.insert(or_fail(),
      into<User>(),
      columns(&User::id, &User::name),
      values(std::make_tuple(1, "The Weeknd")));

auto rows = storage.insert(or_ignore(),
      into<User>(),
      columns(&User::id, &User::name),
      values(std::make_tuple(1, "The Weeknd")));
auto rows = storage.insert(or_replace(),
      into<User>(),
      columns(&User::id, &User::name),
      values(std::make_tuple(1, "The Weeknd")));
auto rows = storage.insert(or_rollback(),
      into<User>(),
      columns(&User::id, &User::name),
      values(std::make_tuple(1, "The Weeknd")));
```

Update

This enables us to update data of existing rows in the table. The general syntax is like this:

UPDATE table SET column_1 = new_value_1, column_2 = new_value_2 WHERE search_condition;

[18] See PRAGMA recursive_triggers

Update several rows

```
//  UPDATE COMPANY SET ADDRESS = 'Texas', SALARY = 20000.00 WHERE AGE < 30
storage.update_all(set(
       c(&Employee::address) = "Texas", c(&Employee::salary) = 20000.00),
       where(c(&Employee::age) < 30));

//  UPDATE contacts
//  SET phone = REPLACE[19](phone, '410', '+1-410')
storage.update_all(set(
       c(&Contact::phone) = replace(&Contact::phone, "410", "+1-410")));
```

Update one row

```
//  UPDATE products
//  SET quantity = 5 WHERE id = 1;
storage.update_all(set(
       c(&Product::quantity) = 5),
       where(c(&Product::id) == 1));
```

Update an object

If student exists then update, else insert:

```
if(storage.count<Student>(where(c(&Student::id) == student.id))) {
       storage.update(student);
} else {
       studentId = storage.insert(student);   // returns primary key
}

auto employee6 = storage.get<Employee>(6);

//  UPDATE 'COMPANY' SET "NAME" = val1, "AGE" = val2, "ADDRESS" = “Texas” , "SALARY" = val4 WHERE "ID" = 6
employee6.address = "Texas";
storage.update(employee6);  //  actually this call updates all non-primary-key columns' values to passed object's
                            //  fields
```

[19] See Core functions for definition of replace()

Delete Syntax

Since delete is a C++ keyword, remove and remove_all are used instead in sqlite_orm. The general syntax for DELETE is in SQL:

DELETE FROM table-name [WHERE expr]

Delete rows that satisfy a condition

```
//  DELETE FROM artist WHERE artistname = 'Sammy Davis Jr.';
storage.remove_all<Artist>(
      where(c(&Artist::artistName) == "Sammy Davis Jr."));
```

Delete all objects of a certain type

```
//  DELETE FROM Customer
storage.remove_all<Customer>();
```

Delete a certain object by giving its primary key

```
//  DELETE FROM Customer WHERE id = 1;
storage.remove<Customer>(1);
```

Replace

If we want to set the primary key columns as well as the rest, we need to use replace instead of insert:

```
User john{
        2,
        "John",
        {0x10, 0x20, 0x30, 0x40},
    };

// REPLACE INTO 'Users ("id", "name", "hash") VALUES (2, “John”, {0x10, 0x20, 0x30, 0x40})
storage.replace(john);
```

Transactions

Transactions

SQLite is transactional in the sense that all changes and queries are atomic, consistent, isolated and durable, better known as ACID:

1. Atomic: the change cannot be broken into smaller ones: committing a transaction either applies every statement in it or none at all.
2. Consistent: the data must meet all validation rules before and after a transaction
3. Isolation: assume 2 transactions executing at the same time attempting to modify the same data. One of the 2 must wait until the other completes in order to maintain isolation
4. Durability: consider a transaction that commits but then the program crashes or the operating system crashes or there is a power failure to the computer. A transaction must ensure that the committed changes will persist even under such situations.

Sqlite has some pragmas that define exactly how these transactions are done and what level of durability they offer. For better durability less performance. Please see **PRAGMA** *schema*.**journal_mode in** Pragma statements supported by SQLite… and Write-Ahead Logging (sqlite.org) for detailed discussion.

NOTE: Changes to the database are faster if done within a transaction as in what follows:

```
storage.begin_transaction();
storage.replace(Employee{
        1,
        "Adams",
        "Andrew",
        "General Manager",
        {},
        "1962-02-18 00:00:00",
        "2002-08-14 00:00:00",
        "11120 Jasper Ave NW",
        "Edmonton",
        "AB",
        "Canada",
        "T5K 2N1",
        "+1 (780) 428-9482",
        "+1 (780) 428-3457",
        "andrew@chinookcorp.com",
});
storage.replace(Employee{
        2,
        "Edwards",
        "Nancy",
        "Sales Manager",
        std::make_unique<int>(1),
        "1958-12-08 00:00:00",
        "2002-05-01 00:00:00",
        "825 8 Ave SW",
        "Calgary",
        "AB",
        "Canada",
        "T2P 2T3",
        "+1 (403) 262-3443",
        "+1 (403) 262-3322",
        "nancy@chinookcorp.com",
});
storage.commit();  // or storage.rollback();
storage.transaction([&storage] {
        storage.replace(Student{1, "Shweta", "shweta@gmail.com", 80});
        storage.replace(Student{2, "Yamini", "rani@gmail.com", 60});
        storage.replace(Student{3, "Sonal", "sonal@gmail.com", 50});
        return true;      // commits
});
```

NOTE: use of transaction guard implements RAII idiom

```
auto countBefore = storage.count<Object>();
try {
      auto guard = storage.transaction_guard();
      storage.insert(Object{0, "John"});
      storage.get<Object>(-1);  // throws exception!
      REQUIRE(false);
} catch(...) {
      auto countNow = storage.count<Object>();
      REQUIRE(countBefore == countNow);
}
```

```
auto countBefore = storage.count<Object>();
try {
	auto guard = storage.transaction_guard();
	storage.insert(Object{0, "John"});
	guard.commit();
	storage.get<Object>(-1);  // throws exception but transaction is not rolled back!
	REQUIRE(false);
} catch(...) {
	auto countNow = storage.count<Object>();
	REQUIRE(countNow == countBefore + 1);
}
```

Core functions

```
//  SELECT name, LENGTH(name)
//  FROM marvel
auto namesWithLengths = storage.select(
	columns(&MarvelHero::name,
	length(&MarvelHero::name)));  //  namesWithLengths is std::vector<std::tuple<std::string, int>>

//  SELECT ABS(points)
//  FROM marvel
auto absPoints = storage.select(
	abs(&MarvelHero::points));  //  std::vector<std::unique_ptr<int>>
cout << "absPoints: ";
for(auto& value: absPoints)
{
	if(value) {
		cout << *value;
	} else {
		cout << "null";
	}
	cout << " ";
}
cout << endl;

//  SELECT LOWER(name)
//  FROM marvel
auto lowerNames = storage.select(
	lower(&MarvelHero::name));
```

```
//  SELECT UPPER(abilities)
//  FROM marvel
auto upperAbilities = storage.select(
      upper(&MarvelHero::abilities));

storage.transaction([&] {
       storage.remove_all<MarvelHero>();
      { //  SELECT changes()

            auto rowsRemoved = storage.select(changes()).front();
            cout << "rowsRemoved = " << rowsRemoved << endl;
            assert(rowsRemoved == storage.changes());
      }
      { //  SELECT total_changes()
            auto rowsRemoved = storage.select(total_changes()).front();
            cout << "rowsRemoved = " << rowsRemoved << endl;
            assert(rowsRemoved == storage.changes());
      }
      return false;      // rollback
});
//  SELECT CHAR(67, 72, 65, 82)
auto charString = storage.select(
      char_(67, 72, 65, 82)).front();
cout << "SELECT CHAR(67,72,65,82) = *" << charString << "*" << endl;

//  SELECT LOWER(name) || '@marvel.com'
//  FROM marvel
auto emails = storage.select(
      lower(&MarvelHero::name) || c("@marvel.com"));

//  SELECT TRIM('   TechOnTheNet.com   ')
auto string = storage.select(
      trim("   TechOnTheNet.com   ")).front();

//  SELECT TRIM('000123000', '0')
storage.select(
      trim("000123000", "0")).front()
```

```
//  SELECT * FROM marvel ORDER BY RANDOM()
for(auto& hero: storage.iterate<MarvelHero>(order_by(sqlite_orm::random()))) {
        cout << "hero = " << storage.dump(hero) << endl;
}
```

NOTE: Use iterate for large result sets because it does not load all the rows into memory

```
//  SELECT ltrim('   TechOnTheNet.com    is great!');
storage.select(ltrim("   TechOnTheNet.com    is great!")).front();
```

Core functions can be used within prepared statements:

```
auto lTrimStatement = storage.prepare(select(
      ltrim("000123", "0")));

//  SELECT ltrim('123123totn', '123');
get<0>(lTrimStatement) = "123123totn";
get<1>(lTrimStatement) = "123";
cout << "ltrim('123123totn', '123') = " << storage.execute(lTrimStatement).front() << endl;

//  SELECT rtrim('TechOnTheNet.com   ');
cout << "rtrim('TechOnTheNet.com   ') = *" << storage.select(rtrim("TechOnTheNet.com   ")).front() << "*" << endl;

//  SELECT rtrim('123000', '0');
cout << "rtrim('123000', '0') = *" << storage.select(rtrim("123000", "0")).front() << "*" << endl;
//  SELECT coalesce(NULL,20);
cout << "coalesce(NULL,20) = " << storage.select(coalesce<int>(std::nullopt, 20)).front() << endl;
cout << "coalesce(NULL,20) = " << storage.select(coalesce<int>(nullptr, 20)).front() << endl;

//  SELECT substr('SQLite substr', 8);
cout << "substr('SQLite substr', 8) = " << storage.select(substr("SQLite substr", 8)).front() << endl;

//  SELECT substr('SQLite substr', 1, 6);
cout << "substr('SQLite substr', 1, 6) = " << storage.select(substr("SQLite substr", 1, 6)).front() << endl;

//  SELECT hex(67);
cout << "hex(67) = " << storage.select(hex(67)).front() << endl;

//  SELECT quote('hi')
cout << "SELECT quote('hi') = " << storage.select(quote("hi")).front() << endl;
```

```
//  SELECT hex(randomblob(10))
cout << "SELECT hex(randomblob(10)) = " << storage.select(hex(randomblob(10))).front() << endl;

//  SELECT instr('something about it', 't')
cout << "SELECT instr('something about it', 't') = " << storage.select(instr("something about it", "t")).front();

struct o_pos : alias_tag {
      static const std::string& get() {
            static const std::string res = "o_pos";
                  return res;
            }
      };

//  SELECT name, instr(abilities, 'o') o_pos
//  FROM marvel
//  WHERE o_pos > 0
auto rows = storage.select(columns(
      &MarvelHero::name, as<o_pos>(instr(&MarvelHero::abilities, "o"))),
            where(greater_than(get<o_pos>(), 0)));

//  SELECT replace('AA B CC AAA','A','Z')
cout << "SELECT replace('AA B CC AAA','A','Z') = " << storage.select(replace("AA B CC AAA", "A", "Z")).front();

//  SELECT replace('This is a cat','This','That')
cout << "SELECT replace('This is a cat','This','That') = "
      << storage.select(replace("This is a cat", "This", "That")).front() << endl;

//  SELECT round(1929.236, 2)
cout << "SELECT round(1929.236, 2) = " << storage.select(round(1929.236, 2)).front() << endl;

//  SELECT round(1929.236, 1)
cout << "SELECT round(1929.236, 1) = " << storage.select(round(1929.236, 1)).front() << endl;

//  SELECT round(1929.236)
cout << "SELECT round(1929.236) = " << storage.select(round(1929.236)).front() << endl;

//  SELECT unicode('A')
cout << "SELECT unicode('A') = " << storage.select(unicode("A")).front() << endl;

//  SELECT typeof(1)
cout << "SELECT typeof(1) = " << storage.select(typeof_(1)).front() << endl;
```

```
// SELECT firstname, lastname, IFNULL(fax, 'Call:' || phone) fax
// FROM customers ORDER BY firstname
auto rows = storage.select(columns(
      &Customer::firstName, &Customer::lastName,
      ifnull<std::string>(&Customer::fax, "Call:" || c(&Customer::phone))),
      order_by(&Customer::firstName));
cout << "SELECT last_insert_rowid() = " << storage.select(last_insert_rowid()).front() << endl;
```

User defined functions

A scalar function produces an output for each row of input, for example, taking the ABS of a column or expression. An aggregate function accepts values from multiple rows and produces an output, for example, taking the maximum of a column or expression.

Scalar functions

Scalar functions must be defined as a dedicated class with at least two functions:
1. `operator()` which takes any amount of arguments that can be obtained using row_extractor and returns the result of any type that can be bound using statement_binder
2. static `name` which return a name. Return type can be any and must have operator<<(std::ostream &) overload (this includes std::string, std::string_view, const char*)

The function can accept variadic arguments by making `operator()` accept a single argument of type `const arg_values&`

Example

```
struct NegateFunction {

    double operator()(double arg) const {
        return -1 * arg;
    }

    static const char* name() {
        return "NEGATE";
    }
};
inline constexpr auto negate = func<NegateFunction>;
```

```
storage.create_scalar_function<negate>();
auto res = storage.select(negate(&Table::b));
for(auto val : res)
{
    cout << val << endl;
}
```

```
Alternative shortcut approach:
```

```
constexpr orm_quoted_scalar_function auto triple = "triple"_scalar.quote([](double arg)
{
     return 3 * arg;
});
storage.create_scalar_function<triple>();
auto res = storage.select(triple(&Table::b));
```

Aggregate functions

```
Aggregate functions must be defined as a dedicated class with at least three functions:
   1. `void step(...)` which can be called 0 or more times during one call inside single SQL query (once per row).
      It can accept any number of arguments that can be obtained using row_extractor.
   2. `fin() const` which is called once per call inside single SQL query. `fin` can have any result type that can
      be bound using statement_binder. Result of `fin` is the result of an aggregate function.
   3. static `name` which return a name. Return type can be any and must have operator<<(std::ostream &) overload.
      You can use `std::string`, `std::string_view` or `const char *` for example.
```

Example:

```
struct MeanFunction {
    double total = 0;
    int count = 0;

    MeanFunction() {}

    MeanFunction(const MeanFunction&) = delete;

    ~MeanFunction() {}
```

```cpp
    void step(double value) {
        total += value;
        ++count;
    }

    double fin() const {
        return total / count;
    }

    static std::string name() {
        return "MEAN";
    }
};

inline constexpr auto mean = func<MeanFunction>;
storage.create_aggregate_function<mean>();
auto rows = storage.select(mean(&Table::a));
auto val = rows[0];
```

Data Mapping

Sqlite data types

SQLITE uses dynamic type system: the value stored in a column determines its data type, not the column's data type. You can even declare a column without specifying a data type. However columns created by sqlite_orm do have a declared data type.
SQLite provides primitive data types we call storage classes which are more general than a data type: INTEGER storage class includes 6 different types of integers.

Storage class	Meaning
NULL	NULL values mean missing information or unknown
Integer	Whole numbers vith variable sizes such as 1,2,3,4 or 8 bytes
REAL	Real numbers with decimal values using 8 byte floats
TEXT	Stores character data of unlimited length. Supports various character encodings
BLOB	Binary large object that can store any kind of data of any length

The data type of a value is taken by these rules:

- If a literal has no enclosing quotes and decimal point or exponent, SQLite assigns the INTEGER storage class
- If a literal is inclosed by single or double quotes, SQLite assigns the TEXT storage class

- If a literal does not have quotes nor decimal points nor exponent, SQLite assigns the REAL storage class
- If a literal is NULL without quotes, it is assigned a NULL storage class
- If a literal has the format X'ABCD' or x'ábcd' SQLIte assignes BLOB storage class.
- Date and time can be stored as TEXT, INTEGER or REAL

How is data sorted when there are different storage classes?
Following these rules:

- NULL storage class has the lowest value… between NULL values there is no order
- The next higher storage classes are INTEGER and REAL, comparing them numerically
- The next higher storage class is TEXT, comparing them according to collation
- Highest storage class is BLOB, using the C function ***memcmp()*** to compare BLOB values

When using ORDER BY 2 steps are followed:

- Group values based on storage class: NULL, INTEGER, REAL, TEXT, BLOB
- Sort the values in each group

Therefore, even if the engine allows different types in one column, it is not a good idea!

Manifest typing and type affinity

- Manifest typing means that a data type is a property of a value stored in a column, not the property of the column in which the value is stored.. values of any type can be stored in a column
- Type affinity is the recommended type for data stored in that column – recommended, not required

SELECT typeof(100), typeof(10.0), typeof('100'), typeof(x'1000'), typeof(NULL);
In sqlite_orm typeof is typeof_.

Create table

In sqlite_orm we create the tables, indices, unique constraints, check constraints and triggers using the make_storage() function. Each data field of the struct we want to persist is mapped to one column in the table – but we don't have to add all of them: a struct may have non-storable fields. So first we define the types, normalize[20] them and create the structs. These structs can be called "persistent atoms".

```
struct User {
    int id;
    std::string name;
    std::vector<char> hash;  //  binary format
};

int main(int, char**) {
    using namespace sqlite_orm;
```

[20] As in relational normalization

```
    auto storage = make_storage("blob.sqlite",
                                make_table("users",
                                           make_column("id", &User::id),
                                           make_column("name", &User::name, default_value("?")),
                                           make_column("hash", &User::hash)));
    storage.sync_schema();
}
```

This creates a database with name blob.sqlite and one table called users with 3 columns. The sync_schema() synchronizes the schema with the database but does not always work for existing tables. A workaround is to drop the tables and start the schema from cero. Adding uniqueness constraints to existing tables usually won't work… you need to version tables for doing some schema changes. It also defines a default value for column "name".

CHECK constraint

Ensure values in columns meet specified conditions defined by an expression:

```
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),
            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();
```

This adds a check constraint and a column with default value.

Columns with specific collation and tables with Primary Key

```
struct User {
    int id;
    std::string name;
    time_t createdAt;
};

struct Foo {
    std::string text;
    int baz;
};

int main(int, char**) {

      using namespace sqlite_orm;
      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();
}
```

This creates a case insensitive text column (other collations exist: collate_rtrim and collate_binary).

FOREIGN KEY constraint

```
struct Artist {
    int artistId;
    std::string artistName;
};

struct Track {
    int trackId;
    std::string trackName;
    std::optional<int> trackArtist;  //  must map to &Artist::artistId
};

int main(int, char** argv) {
      cout << "path = " << argv[0] << endl;

      using namespace sqlite_orm;
      {  //  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)));
            auto syncSchemaRes = storage.sync_schema();
            for (auto& p : syncSchemaRes) {
                  cout << p.first << " " << p.second << endl;
            }
      }
}
```

This one defines a simple foreign key but with actions.

```
struct User {
    int id;
    std::string firstName;
    std::string lastName;
};

struct UserVisit {
    int id;
    int userId;
    std::string userFirstName;
    time_t time;
};
```

```
int main() {
      using namespace sqlite_orm;

      auto storage = make_storage(
            {},
            make_table("users",
                  make_column("id", &User::id, primary_key(), autoincrement()),
                  make_column("first_name", &User::firstName),
                  make_column("last_name", &User::lastName),
                  primary_key(&User::id, &User::firstName)),
            make_table("visits",
                  make_column("id", &User::id, primary_key(), autoincrement()),
                  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).on_delete.restrict_().on_update.cascade()));
      storage.sync_schema();
}
```
This defines a compound foreign key and a corresponding compound primary key.

AUTOINCREMENT property

```
struct DeptMaster {
    int deptId = 0;
    std::string deptName;
};

struct EmpMaster {
    int empId = 0;
    std::string firstName;
    std::string lastName;
    long salary;
    decltype(DeptMaster::deptId) deptId;
};
```

```
int main() {
	using namespace sqlite_orm;

	auto storage = make_storage("",  // empty db name means in memory db
				make_table("dept_master",
					make_column("dept_id", &DeptMaster::deptId, primary_key(), autoincrement()),
					make_column("dept_name", &DeptMaster::deptName)),
				make_table("emp_master",
					make_column("emp_id", &EmpMaster::empId, autoincrement(), primary_key()),
					make_column("first_name", &EmpMaster::firstName),
					make_column("last_name", &EmpMaster::lastName),
					make_column("salary", &EmpMaster::salary),
					make_column("dept_id", &EmpMaster::deptId)));
	storage.sync_schema();
}
```

This defines both primary keys to be autoincrement(), so if you do not specify aa value for the primary key one is created in a sequence. You may also determine the primary key explicitly using replace.

GENERATED COLUMNS

```
struct Product {
	int id = 0;
	std::string name;
	int quantity = 0;
	float price = 0;
	float totalValue = 0;
};
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)))));
storage.sync_schema();
```

This defines a generated column!

Databases may be created in memory if desired

By using the special name “:memory:” or just an empty name, SQLITE is instructed to create the database in memory.

```
struct RapArtist {
    int id;
    std::string name;
};

int main(int, char**) {

      auto storage = make_storage(":memory:",
                          make_table("rap_artists",
                                make_column("id", &RapArtist::id, primary_key()),
                                make_column("name", &RapArtist::name)));
      cout << "in memory db opened" << endl;
      storage.sync_schema();
}
```

This one is stored in memory (can also leave the dbname empty to achieve the same effect).

INDEX, UNIQUE INDEX, PARTIAL INDEX

```
struct Contract {
    std::string firstName;
    std::string lastName;
    std::string email;
};

using namespace sqlite_orm;
```

```
//  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)));
```

This one allows you to create an index and a unique index.
This code allows you to create a partial index.

```
create index partial_status on txn_table (status)
where status in ('A', 'P', 'W');
```

```
auto storage = make_storage(
       "index.sqlite",
       make_index("partial_status", &Contract::status,
              where( in( &Contract::status, {‘A’,’P’, ‘W’}))
```

DEFAULT VALUE for DATE columns

```
struct Invoice
{
       int id;
       int customerId;
       std::optional<std::string> invoiceDate;
};

using namespace sqlite_orm;


int main(int, char** argv) {
       cout << argv[0] << endl;

       auto storage = make_storage("aliases.sqlite",
make_table("Invoices", make_column("id", &Invoice::id, primary_key(), autoincrement()),
      make_column("customerId", &Invoice::customerId),
      make_column("invoiceDate", &Invoice::invoiceDate, default_value(date("now")))));
```

this one defines the default value of invoiceDate to be the current date at the moment of insertion.

```
/**
 *  This is just a mapped type.
 */
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 {};
      }
}
```

Implements a persistent map.

GETTERS and SETTERS

```
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;
    }
};
int main(int, char**) {
      using namespace sqlite_orm;
      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,  //  order between setter and getter doesn't matter.
                                        &Player::setName)));
      storage.sync_schema();
}
```

This one uses getters and setters (note the order does not matter).

DEFINING THE SHEMA FOR SELF-JOINS

```cpp
struct Employee {
    int employeeId;
    std::string lastName;
    std::string firstName;
    std::string title;
    std::unique_ptr<int> reportsTo;    // can also be std::optional<int> for nullable columns
    std::string birthDate;
    std::string hireDate;
    std::string address;
    std::string city;
    std::string state;
    std::string country;
    std::string postalCode;
    std::string phone;
    std::string fax;
    std::string email;
};
int main() {
      using namespace sqlite_orm;
      auto storage = make_storage("self_join.sqlite",
                  make_table("employees",
                        make_column("EmployeeId", &Employee::employeeId, autoincrement(), primary_key()),
                        make_column("LastName", &Employee::lastName),
                        make_column("FirstName", &Employee::firstName),
                        make_column("Title", &Employee::title),
                        make_column("ReportsTo", &Employee::reportsTo),
                        make_column("BirthDate", &Employee::birthDate),
                        make_column("HireDate", &Employee::hireDate),
                        make_column("Address", &Employee::address),
                        make_column("City", &Employee::city),
                        make_column("State", &Employee::state),
                        make_column("Country", &Employee::country),
                        make_column("PostalCode", &Employee::postalCode),
                        make_column("Phone", &Employee::phone),
                        make_column("Fax", &Employee::fax),
                        make_column("Email", &Employee::email),
                        foreign_key(&Employee::reportsTo).references(&Employee::employeeId)));
      storage.sync_schema();
}
```

SUBENTITIES

```
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;
};

using namespace sqlite_orm;
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),
                         foreign_key(&Mark::student_id).references(&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
}
```

This one implements a sub-entity.

UNIQUENESS AT THE COLUMN AND TABLE LEVEL

```
struct Entry {
    int id;
    std::string uniqueColumn;
    std::unique_ptr<std::string> nullableColumn;
};

int main(int, char**) {
      using namespace sqlite_orm;
      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)));
```

this one implements uniqueness at the column and table levels.

NOT NULL CONSTRAINT

Every data field of a persistent struct is by default not null. If we desire to allow nulls in a column, the type for the corresponding field must be one of these:

1. Std::unique_ptr<T>
2. Std::shared_ptr<T>
3. Std::optional<T>

VACUUM

Why do we need vacuum?

- Dropping database objects such as tables, views, indexes, or triggers marks them as free but the database size does not decrease.
- Every time you insert or delete data from tables, the index and tables become fragmented
- Insert, update and delete operations reduces the number of rows that can be stored in a single page => increases the number of pages necessary to hold a table => decreases cache performance and time to read/write
- Vacuum defragments the database objects, repacks individual pages ignoring the free spaces – it rebuilds the database and enables one to change database specific configuration parameters such as page size, page format and default encoding… just set new values using pragma and proceed with vacuum.

```
storage.vacuum();
```

Triggers

What is a Trigger?

A named database code that is executed automatically when an INSERT, UPDATE or DELETE statement is issued against the associated table.

Why do we need them?

- Auditing: log the changes in sensitive data (e.g. salary, email)
- To enforce complex business rules at the database level and prevent invalid transactions

Syntax:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
  [BEFORE|AFTER|INSTEAD OF[21]] [INSERT|UPDATE|DELETE]
  ON table_name
  [WHEN condition]
BEGIN
 statements;
END;
```

Accessing old and new column values according to action

Action	**Availability**
INSERT	NEW is available
UPDATE	Both NEW and OLD are available
DELETE	OLD is available

[21] Only allowed for views

Examples of Triggers

```
//  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())
```

```
//  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())
```

```
// CREATE TRIGGER validate_fields_before_insert_fondos
//     BEFORE INSERT
//            ON Fondos
//            BEGIN
// SELECT CASE WHEN NEW.abrev = '' THEN RAISE(ABORT, "Fondo abreviacion empty") WHEN LENGTH(NEW.nombre) = 0 // THEN
RAISE(ABORT, "Fondo nombre empty") END;
// END;
make_trigger("validate_fields_before_insert_fondos",
       before()
       .insert()
       .on<Fondo>()
       .begin(select(case_<int>()
              .when(is_equal(new_(&Fondo::abreviacion),""),
                     then(raise_abort("Fondo abreviacion empty")))
              .when(is_equal(length(new_(&Fondo::nombre)), 0),
                     then(raise_abort("Fondo nombre empty")))
              .end()))
       .end())
```

```
// CREATE TRIGGER validate_fields_before_update_fondos
//     BEFORE UPDATE
//            ON Fondos
//            BEGIN
// SELECT CASE WHEN NEW.abrev = '' THEN RAISE(ABORT, "Fondo abreviacion empty") WHEN LENGTH(NEW.nombre) = 0 // THEN
RAISE(ABORT, "Fondo nombre empty") END;
// END;
make_trigger("validate_fields_before_update_fondos",
       before()
       .update()
       .on<Fondo>()
       .begin(select(case_<int>()
              .when(is_equal(new_(&Fondo::abreviacion), ""),
                     then(raise_abort("Fondo abreviacion empty")))
              .when(is_equal(length(new_(&Fondo::nombre)), 0),
                     then(raise_abort("Fondo nombre empty")))
              .end()))
       .end())
```

Data migration

Sqlite_orm supports automatic schema migration to a certain degree but there are a few caveats. Currently you can make changes to the storage schema and call *storage.sync_schema(true)* which will attempt to apply the current schema to the database . There are limits to what it can do and currently we do not support data migration primitives like add_column(), drop_column(), etc. However, the sync_schema method will attempt to detect the changes between the storage schema and the database schema and try to reconcile without data loss most of the time. There are however very clear exceptions, and we are working on them.

First, not all attributes of a column nor attributes of a table are compared to the database schema. This is currently a limitation of the table_xinfo pragma of sqlite3, not of sqlite_orm itself. Second, there is a common source of problems with the foreign key checking mechanism that makes it very difficult to do backups of tables that need to be dropped and recreated. As it happens, when a table has dependent rows, it cannot normally be dropped. There are only two solutions that we know of at present: one is to remove all foreign key constraints of tables towards the table at hand temporarily using a sqlite client like SqliteStudio or DB Browser for Sqlite (see the section on tools). This will allow the backing up of the current table since that process requires a drop table as one of its steps. The other method is controversial but used by these tools and some developers to simplify the process. It has to do with disabling FK checking before doing the backup and restoring it immediately afterwards. This does not require to remove the foreign key constraints that target the table at hand. For more detailed information about how to automate the schema migration process, please feel free to get in contact with the author (see my details at the end of the document). If data preservation and schema evolution are important to you, you need this additional information.

What are the aspects of a column that are comparable to the database schema? First, whether the column is part of the primary key of the table. Second, whether the column has a default value. Third, whether the column is nullable (i.e. if it accepts null values). Fourth, whether the column is hidden (meaning the column is generated_always_as()). Period. All other changes, like what the default value of the column is, or what the generated value of the column is, or whether the column is unique, or has a check constraint, are simply not detected when compared with the physical database schema. For a change in any of these properties to be incorporated at the physical database, we require to drop and recreate the table. The easiest and more secure way to provoke this behavior is to remove the primary key constraint of a table temporarily: this will ensure the drop and recreation of the table using a backup, preserving the data. Putting aside how we deal with foreign key constraints, be it by means of sqlite client tools or at the database configuration level, the essence of data preservation and schema evolution deals with provoking a drop and recreate of the table at hand with a backup process in place.

What actions on the storage schema are detected by *sync_schema* and how exactly does it respond? This next section deals with this topic.

Schema Actions Detected by sync_schema()

1. Adding a column to the storage schema that does not exist in the database
    a. If the column has no default value nor is nullable nor is generated, then there is no way data in that table can be preserved. Just think about it: what value could be inserted in that column for each existing row?
        i. This is therefore strictly prohibited unless you do not care about losing the table's data. If you wish to add such a column you must first add a lossless column (see next) and then tweak its properties as you like (thus a two-step process is unavoidable)
    b. If the column has a default value, is nullable or is generated[22], then there will be an ALTER TABLE ADD COLUMN command that is efficient and effective. There will be no loss of table's data and no backup will be needed
        i. *sync_schema_simulate(true)* will return `sync_schema_result::new_columns_added` for that table
2. Removing a column from storage schema that exists in the database
    a. An ALTER TABLE DROP COLUMN command will be issued and no data loss will occur

[22]If *generated_always_as().stored()* then a drop and recreated will be triggered with backup so no data loss either

3. Remove primary key constraint[23] on a table
    a. This will generate a drop and recreate with backup, thus no data loss will occur
4. Adding a primary key constraint on a table
    a. This will generate a drop and recreate with backup, thus no data loss will occur
5. Adding nullability to a column
    a. This will generate a drop and recreate with backup, thus no data loss will occur
6. Removing nullability to a column
    a. This will generate a drop and recreate with backup, thus no data loss will occur **but**
        i. Make sure every existing row has a value distinct from null in this column prior to removing nullability, otherwise an exception will occur and interrupt the update process, rolling it back to the initial state
7. Adding or removing a default value to a column
    a. This will generate a drop and recreate with backup, thus no data loss will occur

Schema Actions Not Detected by sync_schema()

1. Changing the default value of a column that already had a default value
    a. Will go unnoticed
2. Changing the generated value of a column that already was generated
    a. Will go unnoticed
3. Adding or removing a check clause to a column or a table
    a. Will go unnoticed
    b. If you provoke the drop and recreation with backup by toggling the primary key constraint, for example, then you must ensure that the existing rows pass the check clause if adding one or else the process will be rollbacked to initial state
4. Adding or removing a unique clause to a column or a table (more than one column)
    a. Will go unnoticed
    b. If you provoke the drop and recreation with backup by toggling the primary key constraint, for example, then you must ensure that the column's values are distinct if adding a unique clause otherwise you should not be concerned if removing uniqueness
5. Adding or removing foreign key constraints
    a. Will go unnoticed

Aspects to Consider when Synching a Schema

The method *storage.sync_schema(true)* tries to synchronize the on memory schema (called the storage schema) defined by the *make_storage* call, with the database schema. We are going to explore what this method can handle and what changes it takes care of and what changes it doesn't and what to do when it is not enough for our needs[24]. When we don't care about the data and only want to synchronize schema, then use *storage.sync_schema(false)*.

[23] Be sure to remove autoincrement() as well if there is any (this cannot exist by itself: it is a property of a primary key not a constraint itself)

[24] For proper management of all schemata the availability of a sqlite client like SQLiteStudio may be indispensable depending on the way you approach foreign key constraint management

1. Tables present in the database are not altered in any way nor dropped if they are not mentioned in the *make_table()* calls of *make_storage()* – therefore your C++ project is capable of dealing with a subset of the tables in a database if desired
2. Every table from storage is compared with its database analog and the following rules determine the outcome:
    a. If table does not exist, it will be created
    b. If table exists with excess columns the table will drop the columns to match those defined in storage schema
    c. If table exists with less columns, the table will add the columns to match those defined in storage schema
    d. If the difference in schemas is a detectable one (see above), then if *sync_schema(true)* is called data will be preserved
    e. Otherwise, the difference between schemas will remain in conflict
3. For differences not detected between storage and database schemas, it will be necessary to provoke the drop and recreation of the table; this can be accomplished by one of the following:
    a. Change nullability of a column: if removing nullability make sure there are no null values in the column's rows prior to triggering
    b. Change the presence of a default value of a column (if temporary, remember to restore)
    c. Change the primary key constraint of a column (you must restore it afterwards!)
    d. Add a generated column of the stored type (be sure to remove it afterwards!)

About correct order of dropping/loading tables[25]

Another approach to reapplying schema changes is to load the contents of each table into vectors[26] and drop each table with *storage.drop_table()*. Then drop all in an order that guarantees the absence of dependent rows, and recreate the tables by calling storage.sync_schema(). The order will be determined by the following algorithm: We must create a graph of all tables connected by edges from the table with a foreign key to a table referenced by that foreign key and start dropping from the leaves. We will recreate all tables and reload them in inverse order from that used in the dropping.

Example

Consider a simple schema with two tables, one for User and one for Job and suppose we want to drop and recreate them (or sync_schema is going to try doing this for us). We must be sure that tables with dependent rows do not exist when the referenced table is dropped, else we will get an exception.

```
static auto storage = make_storage(dbFilePath,
    make_unique_index("name_unique.idx", &User::name ),
    make_table("user_table",
        make_column("id", &User::id, primary_key()),
        make_column("name", &User::name),
        make_column("born", &User::born),
        make_column("job", &User::job),
```

[25] When there are cyclic dependencies, we will not find a correct order and schema migration must be done manually in a sqlite client tool such as SqliteStudio or much better, by using `foreign_key_disable_checking` (see example below)
[26] Assuming the tables are not gigantic

```
        make_column("blob", &User::blob),
        make_column("age", &User::age ),
        check(length(&User::name) > 2),
        check(c(&User::born) != 0),
        foreign_key(&User::job).references(&Job::id)),
    make_table("job",
        make_column("id", &Job::id, primary_key(), autoincrement()),
        make_column("name", &Job::name, unique()),
        make_column("base_salary", &Job::base_salary)));
```

Now load and drop tables in correct order:

```
std::vector<User> users = storage.get_all<User>();
storage.drop_table(storage.tablename<User>());
std::vector<Job> jobs = storage.get_all<Job>();
storage.drop_table(storage.tablename<Job>());
```

now call sync_schema() to propagate changes to database:

```
auto m = storage.sync_schema(false);   // we may inspect the return ‘m’ for information of actions performed
```

and reload tables in correct order (inverse of that used in dropping):

```
storage.replace_range(jobs.begin(), jobs.end());
storage.replace_range(users.begin(), users.end());
```

How to drop data without losing it

When we have a difference between the storage schema and the database that is detectable by the *sync_schema()* function and it triggers table to be dropped and recreated or when the difference is not detectable we must do the following to keep schemas synchronized:

1. Call *make_storage()*
2. If the change would add a column that is not nullable and does not have a default value nor is generated, then consider creating an intermediate column with one of these properties and then change the properties; *do not* add such a column in only one step because you will lose all the table data!
3. Load all data from the transitive dependent tables of the current table and the current table in the order specified by About correct order of dropping/loading tables[27]
4. If case is moving from a non-nullable to nullable column decide what (if at all) we are going to interpret as null values and modify the loaded data to change them to null values as described in Interpret values in non-nullable column as nullable
5. Drop tables in order given by point 2 above
6. Call *sync_schema(false)* – we are not using the backup feature of sync_schema() so use false as an argument
7. Replace all data into tables from the std::vectors in reverse order from that in which we dropped them

Interpret values in non-nullable column as nullable

If we are adding nullable to an existing column that is not nullable in the database, then we need to decide if we are going to interpret certain values as nullable and modify the vector's elements. For instance:

a. For integer or real, is 0 to be taken as null?
b. For text, is "" to be taken as null?
c. For blob, is size() of std::vector<char> == 0 to be taken as null?
d. If we decide these values should be treated as nulls, then we must transform the nullable column to std::nullopt following this pattern before replacing the vector into the table. For instance if the type of column job is integer or real, then we check whether its value is 0 and if so we replace it with std::nullopt which is interpreted as NULL in SQL:

```
std::transform(users.begin(), users.end(), users.begin(), [](User& user) {
      if (user.job && user.job.value() == 0) { user.job = std::nullopt; }    return user;
});
```

[27] This will be possible in all cases except when adding a column with following attributes: not nullable, does not provide a default value, is not generated. This case should ALWAYS be avoided. Direct creation of such column types should not be done because we lose all data in the table. The workaround is: add the column desired but give it a nullable quality, or a default value or a generate value; then change the storage schema to remove the condition that allowed adding the column (i.e., nullable or default value or *generated value*) or use Migration (see below ***Beyond Sync_schema***)

Making a backup of the entire database

```
template<typename T>
void backup_db(T& storage, std::string db_name)
{
    namespace fs = std::filesystem;

    auto path_to_db_name = fs::path(db_name);
    auto stem = path_to_db_name.stem().string();
    auto backup_stem = stem + "_backup1.sqlite";
    auto backup_full_path = path_to_db_name.parent_path().append(backup_stem).string();
    storage.backup_to(backup_full_path);
}
```

Ensuring that a column contains unique values before making the column unique

If we have a persistent struct User with a column age which we want to declare unique, we must first detect if there are duplicates in the table. Consider:

1. loading the table into a vector
2. sorting the table by age
3. find if there are repeated values by using adjacent_find algorithm
4. compare return value to end iterator of vector: if it is different then we have a duplicate which we must correct!

The code could be like this:

```
std::vector<User> users = storage.get_all<User>();
std::sort(users.begin(), users.end(), [](const User& l, const User& r) { return l.age < r.age; });
auto it = std::adjacent_find(users.begin(),users.end(),[](const User& l, const User& r) { return l.age == r.age; });
if( it!= users.end()) {
       // there are duplicates!
        User user = *it;         // points to duplicate
        auto age = user.age;     // duplicate age
}
```

Beyond sync_schema – Migration Case Studies

Introduction

For changes in the storage schema not handled by sync_schema(), for example when adding a column that is not nullable nor has default value nor is generated, it becomes necessary to backup, drop, recreate the table, and transform its rows in the restoration of the table. A prerequisite is that this table must have no dependent rows in other tables. If there are FKs in other tables pointing to the table at hand, we must somehow “silence” the relevant FKs or remove the rows themselves from the table temporarily and restore them afterwards. One can at first think that the use of a sqlclient[28] could provide a solution but upon closer examination the idea quickly fades away. First, the update should occur automatically and should take into account the current versus final version of the schemas (sqlcient tools cannot be automated and don’t even exist in the mobile environment).
The other possibility is to leave the FKs untouched but instead remove the dependent rows themselves, after making a backup of them, and while this table has no dependents of its own, and so on in a recursive fashion. The amount of data to be backed up can be substantial and the complexity of the database schema can make this a daunting chore in terms of memory requirements and performance. We demonstrate a simple 2 table case in one of the case studies in this section. When a table is finally free from dependents, then we can apply the migration that transforms the schema and or data from one version to another. Another option for schemas with complex and/or circular dependencies is to silence FK checking in the database connection during the critical steps of a migration, making sure that all data is properly restored so that the FK constraints are correctly preserved. This is discussed in the fourth case study below. For this last scenario, Sqlite_ORM provides a RAII object called `foreign_key_disable_checking` that automatically restores the FK checking upon scope’s end.

Case study: Adding a Self-referential FK to a Table with no Dependent Rows

Suppose we have a type v0::User like this:

```
namespace v0 {
    struct User {
        int id = 0;
        std::string name;
    };
    auto userTable =
        make_table("users", make_column("id", &User::id, primary_key()), make_column("name", &User::name));
  }
```

And we want to migrate to v1::User like this:

[28] Like SqliteStudio or DB Browser for Sqlite

```
    namespace v1 {
        struct User {
            int id = 0;
            std::string name;
            std::unique_ptr<int> reports_to;
        };
        auto userTable = make_table("users",
                                    make_column("id", &User::id, primary_key()),
                                    make_column("name", &User::name),
                                    make_column("reports_to", &User::reports_to),
                                    foreign_key(&User::reports_to).references(&User::id));
}
```

This change adds a self-referential FK which is not synchronized with the DB schema since changes in FKs are not detected by ***sync_schema***. Thus to generate this FK we must drop v0::User and recreate v1::User and provide a transformation function (e.g. a lambda) that will convert each v0::User to v1::User. The complete code to be called is a registered migration like this:

```
        auto migration0_1 = [](const connection_container& connection) {
            auto oldStorage = connection.make_storage(v0::userTable);
            auto newStorage = connection.make_storage(v1::userTable);
            auto lambda = [](const v0::User& user) -> v1::User {
                v1::User nuser{user.id, user.name, nullptr};  // or pass data to set the FK column values
                return nuser;
            };
            backup_drop_restore<v1::User, v0::User, true>(oldStorage, newStorage, lambda);
        };
```

Where the template `backup_drop_restore is:`

```
template<typename NewType, typename OldType, bool Verify = false, typename OldStorage, typename NewStorage>
void backup_drop_restore(OldStorage& oldStorage, NewStorage& newStorage, std::function<NewType(const OldType&)>
transform) {

    std::vector<OldType> oldRows = oldStorage.template get_all<OldType>();

    newStorage.drop_table(newStorage.template tablename<NewType>());
    newStorage.sync_schema();

    for (int i = 0; i < oldRows.size(); ++i) {
        auto& orow = oldRows[i];
        NewType nrow = transform(orow);
        newStorage.replace(nrow);
    }

    if constexpr (Verify) {
        auto newRows = newStorage.template get_all<NewType>();
        for (int i = 0; i < newRows.size(); ++i) {
            auto& orow = oldRows[i];
            auto& nrow = newRows[i];
            REQUIRE(nrow == orow);           // Must defined operator== to receive v0::User and v1::User
        }
        REQUIRE(oldRows.size() == newRows.size());
    }
}
```

Case Study: Adding a Table to Storage with FK Targeting an Existing Table

One very simple schema change is to add another table to storage. In this case sync_schema(true) is perfectly adequate, and migration is trivial[29].

```
    namespace v2 {
        struct User {
            int id = 0;
            std::string name;
            std::unique_ptr<int> reports_to;
        };

        struct Dept {
            int id = 0;
            std::string name;
            std::unique_ptr<int> fkey_manager;  // nullable
        };


        auto userTable = make_table("users",
            make_column("id", &User::id, primary_key()),
            make_column("name", &User::name),
            make_column("reports_to", &User::reports_to),
            foreign_key(&User::reports_to).references(&User::id));
```

[29] Although trivial it still needs to be stored amongst the migrations because it is of vital importance!

```
auto deptTable = make_table("dep",
    make_column("id", &Dept::id, primary_key()),
    make_column("name", &Dept::name),
    make_column("fkey_manager", &Dept::fkey_manager),
    foreign_key(&Dept::fkey_manager).references(&User::id));
```

Migration can be performed like this:

```
auto migration1_2 = [](const connection_container& connection) {
    auto newStorage = connection.make_storage(v2::userTable, v2::deptTable);
    // new table created is handled by sync_schema(true)
    newStorage.sync_schema(true);
};
```

Case Study: Adding a FK to an Existing Table so that a Cyclic Dependency is Created

We have a User and a Dept and there are these FK relations between them:

1. User::reports_to ➔ User::id
2. Dept::manager ➔ User::id

This schema allows us to define an ordering when dropping becomes necessary. Imagine we want to add a relation between User::works_in ➔ Dept::id, which implies adding a FK to User and therefore dropping User is a requirement (since changes in FK are not recognized by ***sync_schema***). However, Dept depends on User through its FK ***manager*** and therefore we cannot drop the User table while there are rows in Dept. Dept however can be dropped immediately since User has no FK into Dept. Adding FK ***works_in*** will create a cycle of dependencies between the two existing tables and will complicate things in the future. However, before this schema change, there is no ***works_in*** FK in User yet and therefore we can proceed.

The steps required to accomplish this migration are as follows[30]:

1. Backup all the rows in Dept
2. Drop Dept
3. Backup all the rows in User
4. Drop User[31]
5. Call sync_schema(false)[32]

[30] You can look for the template function `backup_drop_restore` in unit tests for the implementation
[31] There are no rows in Dept
[32] We have no data to preserve since we dropped both our tables but we have their contents in the backups

6. Restore all users assigning NULL to the new FK
7. Restore all departments
8. Load all departments and compare with backup
9. Load all users and compare with backup

Case Study: Need to Drop a Table that Participates in a Cyclic Dependency

Continuing with our example from previous case study, suppose we need to drop the User table to change something in the schema of that table that is not recognized by ***sync_schema***. How can we do this? Dept has a FK into User so if Dept has rows, we cannot drop User. We need to “disable” this FK. There are only 2 ways that we know of. The simplest one is to disable FK checking in the db connection, drop User, recreate and restore User contents and reenable FK checking. This can be done because we are going to restore the contents of User so no FK constraint is going to be broken during the migration. As mentioned in the introduction, we have created a RAII object that encapsulates the setting of FK checking; it is called ***foreign_key_holder*** and must be defined in the migration steps prior to the critical section to ensure FK checking is restored at the end of the migration. This object’s destructor is declared ***noexcept(false)*** to avoid program termination should there be an exception in the destructor.

Specific case: Add a Check Clause to a Table that Participates in the Cycle

Adding a check clause to User is one of those schema changes not detectable by ***sync_schema*** and thus we need to backup, drop, recreate, and restore User. But Dept has a FK into User and User has a FK into Dept: there is a cycle of dependencies. We cannot drop either table without breaking the FK constraints. In this case the solution is to use ***foreign_key_holder*** to disable FK checking so we can drop User ignoring the FK from Dept. The User table will be restored to its original contents and no FK constraints will have been violated. The migration looks like this:

```
auto migration3_4 = [](const connection_container& connection) {
    auto oldStorage = connection.make_storage(v3::userTable, v3::deptTable);
    auto newStorage = connection.make_storage(v4::userTable, v4::deptTable);

    // to copy User we need to treat the unique_ptr<>s since they are not copyable
    auto transform_user = [](const v3::User& o) -> v3::User {
        auto rep_to = o.reports_to ? std::make_unique<int>(*o.reports_to) : std::unique_ptr<int>();
        auto works_in = o.works_in ? std::make_unique<int>(*o.works_in) : std::unique_ptr<int>();
        v3::User u{o.id, o.name, std::move(rep_to), std::move(works_in)};
        return u;
    };

    // make sure check constraint is valid with current contents
    auto users = oldStorage.get_all<v3::User>();
    bool passes_check_constraint = true;
    for(auto& u: users) {
```

```
                if(u.name.length() == 0) {
                    passes_check_constraint = false;    // abort migration: invalid data!
                    break;
                }
            }
            REQUIRE(passes_check_constraint);

            foreign_key_holder fk_off(newStorage);  // turn FK checking OFF during this object's lifetime!

            backup_drop_restore<v3::User, v3::User, true>(oldStorage, newStorage, transform_user);
        };
```

Helper Functions

```
template<typename NewType, typename Storage, typename OldType>
inline void restore(Storage& storage, const std::vector<OldType>& backup, std::function<NewType(const OldType&)>
transform) {
    for (int i = 0; i < backup.size(); ++i) {
        auto& orow = backup[i];
        NewType nrow = transform(orow);
        storage.replace(nrow);
    }
}

template<typename NewType, typename Storage, typename OldType>
inline bool verify(Storage& storage, const std::vector<OldType>& oldRows) {
    auto newRows = storage.template get_all<NewType>();
    for (int i = 0; i < newRows.size(); ++i) {
        auto& orow = oldRows[i];
        auto& nrow = newRows[i];
          bool eq = nrow == orow;
        REQUIRE(eq);
    }
    REQUIRE(oldRows.size() == newRows.size());
    return true;
}
```

Extensions

Virtual Tables

A virtual table is an object that is registered with an open database connection. From the perspective of an SQL statement, it looks just like a normal table or view but behind the scenes, queries and updates on a virtual table invoke callback methods of the virtual table object instead of reading/writing to the database file.

This mechanism allows an application to publish interfaces that are accessible from SQL statements as if they were tables, except:

1. Impossible to create triggers
2. Impossible to create additional indices
3. Impossible to run ALTER TABLE …. ADD COLUMN commands on the table
4. Additional constraints imposed by the virtual table implementation

Examples of virtual tables include:

1. Full text search interface
2. Spatial indices using R-Trees
3. Introspect the disk content of an SQLITE database (must #define `SQLITE_ENABLE_DBSTAT_VTAB)`
4. Read or write the content of a comma separated value file (CSV)
5. Access the filesystem of the host computer as if it was a database table
6. Enabling SQL manipulation of data in statistics packages like R
7. Etc…[33]

A most valuable and implemented extension to sqlite is FTS5: Full text search interface!

FTS5

FTS5 is an SQLITE virtual table module that provides full-text search functionality to database applications. These engines allow the user to efficiently search a large collection of documents for the subset that contain one or more instances of a search term. It is the functionality behind Google searches and it is supported by sqlite_orm.

Once populated, there are three ways to execute a full-text query against the contents of a FTS5 table:

[33] See https://www.sqlite.org/vtablist.html

1. Using a MATCH operator in the WHERE clause of a SELECT statement
2. Using an equals (‘=’) operator in the WHERE clause of a SELECT statement
3. Using the table valued function syntax

Create virtual table with at least one column, e.g.:

```
// CREATE VIRTUAL TABLE email USING fts5(sender, title, body);
// populate it!
// Query for all rows that contain at least one instance of the term
// "fts5" (in any column). The following three queries are equivalent.
// SELECT * FROM email WHERE email MATCH 'fts5';
// SELECT* FROM email WHERE email = 'fts5';
// SELECT* FROM email('fts5');
```

To sort results by relevance use the ‘ORDER BY rank’ clause, e.g.:

```
// Query for all rows that contain at least once instance of the term
// "fts5" (in any column). Return results in order from best to worst
// match.
// SELECT * FROM email WHERE email MATCH 'fts5' ORDER BY rank;
```

As well as the column values and rowid of a matching row, an application may use FTS5 auxiliary functions to retrieve extra information regarding the matched row, provided the table name is given as first parameter, e.g.:

```
// Query for rows that match "fts5". Return a copy of the "body" column
// of each row with the matches surrounded by <b></b> tags.
//SELECT highlight(email, 2, '<b>', '</b>') FROM email('fts5');
```

C++ code:

```
struct Post {
    std::string title;
    std::string body;

    bool operator==(const Post& other) const {
        return this->title == other.title && this->body == other.body;
    }
};

/// CREATE VIRTUAL TABLE posts
/// USING FTS5(title, body);
auto storage = make_storage(
    "",
        make_virtual_table("posts", using_fts5(make_column("title", &Post::title), make_column("body",
&Post::body))));

    storage.sync_schema();
```

Populate table:

```
    const std::vector<Post> postsToInsert = {
        Post{"Learn SQlite FTS5", "This tutorial teaches you how to perform full-text search in SQLite using FTS5"},
        Post{"Advanced SQlite Full-text Search", "Show you some advanced techniques in SQLite full-text searching"},
        Post{"SQLite Tutorial", "Help you learn SQLite quickly and effectively"},
    };
    storage.insert_range(postsToInsert.begin(), postsToInsert.end());
```

Query table:

```
     /// SELECT *
     /// FROM posts
     /// WHERE posts MATCH 'fts5';
     auto specificPosts = storage.get_all<Post>(where(match<Post>("fts5")));
     decltype(specificPosts) expectedSpecificPosts = {
         {"Learn SQlite FTS5", "This tutorial teaches you how to perform full-text search in SQLite using FTS5"},
     };
     REQUIRE(specificPosts == expectedSpecificPosts);

     ///    SELECT *
     ///    FROM posts
     ///    WHERE posts = 'fts5';
     auto specificPosts2 = storage.get_all<Post>(where(is_equal<Post>("fts5")));
     REQUIRE(specificPosts2 == specificPosts);

     ///    SELECT *
     ///    FROM posts
     ///    WHERE posts MATCH 'text'
     ///    ORDER BY rank;
     auto orderedPosts = storage.get_all<Post>(where(match<Post>("fts5")), order_by(rank()));

     ///    SELECT highlight(posts,0, '<b>', '</b>'),
     ///           highlight(posts,1, '<b>', '</b>')
     ///    FROM posts
     ///    WHERE posts MATCH 'SQLite'
     ///    ORDER BY rank;
```

```
    ///
    auto highlightedPosts =
        storage.select(columns(highlight<Post>(0, "<b>", "</b>"), highlight<Post>(1, "<b>", "</b>")),
                       where(match<Post>("SQLite")),
                       order_by(rank()));
    // gives this output:

<b>SQLite</b> Tutorial Help you learn <b>SQLite</b> quickly and effectively
Advanced <b>SQlite</b> Full-text Search Show you some advanced techniques in <b>SQLite</b> full-text searching
Learn <b>SQlite</b> FTS5 This tutorial teaches you how to perform full-text search in <b>SQLite</b> using FTS


   // to specify that a column is not to be part of the FTS index, we can use unindexed() in its creation, e.g.:
   // CREATE VIRTUAL TABLE posts1 USING fts5(title, body UNINDEXED);
    auto storage = make_storage("",
           make_virtual_table("posts1",
                 using_fts5(make_column("title", &Post::title),
                        make_column("body", &Post::body, unindexed()))));

      // to create prefix indexes that records the location of all instances of prefix tokens of a certain length
      // of characters we create the virtual table like this:
      auto storage2 = make_storage( "",
             make_virtual_table("posts2",
                 using_fts5(make_column("title", &Post::title), make_column("body", &Post::body), prefix(2))));

      // to set the tokenizer we create the virtual table like this:
      // CREATE VIRTUAL TABLE posts3 USING FTS5("title", "body", tokenize = 'porter ascii')
      auto storage3 = make_storage("",
             make_virtual_table("posts3",
                   using_fts5(make_column("title", &Post::title), make_column("body", &Post::body),
                         tokenize("porter ascii"))));

      // to create a content-less table (i.e. only the full-text index entries are stored in the virtual table and
      // only the rowid can be obtained from a row) - they do not support UPDATE or DELETE commands and only INSERT
      // commands where a non-null rowid is given
      // CREATE VIRTUAL TABLE posts4 USING FTS5("title", "body", content='')
      auto storage4 = make_storage(
             "",
             make_virtual_table(
                 "posts4",
                 using_fts5(make_column("title", &Post::title), make_column("body", &Post::body), content(""))));
```

Pointer Passing Interface

To allow extension components to pass private information to one another securely and without introducing pointer leaks requires new interfaces:

1. **sqlite3_bind_pointer(S,I,P,T,D)**
2. **sqlite3_result_pointer(C,P,T,D)**
3. **sqlite3_value_pointer(V,T)**

1-> Bind pointer P of type T to the I-th parameter of prepared statement S. D is an optional destructor function for P.
2-> Return pointer P of type T as the argument of function C. D is an optional destructor function for P.
3-> Return the pointer of type T associated with value V, or if V has no associated pointer, or if the pointer on V is of a type different from T, then return NULL

To SQL, the values created by sqlite3_bind_pointer() and sqlite3_result_pointer() are indistinguishable from NULL. An SQL statement that tries to use the hex() function to read the value of a pointer will get an SQL NULL answer. The only way to discover whether or not a value has an associated pointer is to use the sqlite3_value_pointer() interface with the appropriate type string T.
Pointer values read by sqlite3_value_pointer() cannot be generated by pure SQL. Hence, it is not possible for SQL to forge pointers.
Pointer values generated by sqlite3_bind_pointer() and sqlite3_result_pointer() cannot be read by pure SQL. Hence, it is not possible for SQL to leak the value of pointers.
In this way the new pointer-passing interface seems to solve all of the security problems associated with passing pointer values from one extension to another in SQLite.

Importance of the pointer type

When a system includes more than one extension, then it would be possible to pass one pointer from one extension to another as in the following example:

```
SELECT ca.value FROM t1, carray(t1,10) AS ca WHERE cx MATCH $pattern
```

Here the pointer to the cursor object from the FTS3 extension generated by the match operator is sent to the carray() table-valued function and could be read if the types would be the same (or if there were no explicit types). Because of the existence of string pointer types, the system can quickly deny to copy the original pointer into a different extension – it is just not allowed, the types do not match. Thus no pointer can leak from one extension to another…
The pointer generated by the MATCH operator is of the type “fts3cursor”, but the carray() function expects to receive a pointer of the “carray” type. This occurs because the pointer type on the sqlite3_result_pointer() does not match the type on the sqlite3_value_pointer() call, the sqlite3_value_pointer() in carray() return NULL and thus signals the CARRAY extension that an invalid pointer has been received. The power of the carray() function that returns a pointer to a C array of items is thus not available for misuse.

Pointer types are static strings

Static strings implies a zero-terminated array of bytes that is fixed and unchanging for the life of the program, e.g. constexpr strings. Why?

1. Pointer types are not intended to be flexible and dynamic – rather a design time constant.
2. All string values at the SQL level are dynamic strings so its difficult to create a SQL function that can synthesize a pointer of an arbitrary type. The requirement to use static strings helps to defend the integrity of the pointer-passing interfaces against ill-designed SQL functions. Its not a perfect guarantee but it does complicates things for the ill doers.
3. Having Sqlite take ownership of the type strings implies a performance cost for all applications even for those that do not use pointer passing.

Sqlite_Orm implementation of pointer types

```
/** @short Specifies that a type is an integral constant string usable as a pointer type.
 */
template<class T>
concept orm_pointer_type = requires {
    typename T::value_type;
    { T::value } -> std::convertible_to<const char*>;
};

Wraps a pointer and tags it with a pointer type,
used for accepting function parameters, facilitating the 'pointer-passing interface'.
Template parameters:
     - P: The value type, possibly const-qualified.
     - T: An integral constant string denoting the pointer type, e.g. `"carray"_pointer_type`.
pointer_arg<P, T>

template<class P, orm_pointer_type auto tag>
using pointer_arg_t = pointer_arg<P, decltype(tag)>;

/**
 *  Forward a pointer value from an argument.
 */
template<class P, class T>
auto rebind_statically(const pointer_arg<P, T>& pv) noexcept -> static_pointer_binding<P, T> {
    return bind_pointer_statically<T>(pv.ptr());
}
```

```
/**
 *  Wrap a pointer and its type for binding it to a statement.
 *
 *  Note: 'Static' means that ownership of the pointed-to-object won't be transferred
 *  and sqlite assumes the object pointed to is valid throughout the lifetime of a statement.
 */
template<class T, class P>
auto bind_pointer_statically(P* p) noexcept -> static_pointer_binding<P, T> {
    return bind_pointer<T>(p, null_xdestroy_f);
}
/**
 *  Wrap a pointer, its type and its deleter function for binding it to a statement.
 *
 *  Unless the deleter yields a nullptr 'xDestroy' function the ownership of the pointed-to-object
 *  is transferred to the pointer binding, which will delete it through
 *  the deleter when the statement finishes.
 */
template<class T, class P, class D>
auto bind_pointer(P* p, D d) noexcept -> pointer_binding<P, T, D> {
    return {p, std::move(d)};
}

/**
 *  Alias template for a static pointer value binding.
 *  'Static' means that ownership won't be transferred to sqlite,
 *  sqlite doesn't delete it, and sqlite assumes the object
 *  pointed to is valid throughout the lifetime of a statement.
 */
template<typename P, typename T>
using static_pointer_binding = pointer_binding<P, T, null_xdestroy_t>;

/**
 *  Pointer value with associated deleter function,
 *  used for returning or binding pointer values
 *  as part of facilitating the 'pointer-passing interface'.
 *
 *  Template parameters:
 *    - P: The value type, possibly const-qualified.
 *    - T: An integral constant string denoting the pointer type, e.g. `carray_pointer_type`.
 *    - D: The deleter for the pointer value;
 *         can be one of:
 *         - function pointer
 *          - integral function pointer constant
 *          - state-less (empty) deleter
 *          - non-capturing lambda
 *          - structure implicitly yielding a function pointer
 *
 *  @note Use one of the factory functions to create a pointer binding,
 *  e.g. bindable_carray_pointer or statically_bindable_carray_pointer().
 *
 *  @example
 *  ```
 *  int64 rememberedId;
 *  storage.select(func<remember_fn>(&Object::id, statically_bindable_carray_pointer(&rememberedId)));
 *  ```
 */
template<typename P, typename T, typename D>
class pointer_binding {
```

carray pointer type binding

```
inline constexpr orm_pointer_type auto carray_pointer_tag = "carray"_pointer_type;

template<typename P>
using carray_pointer_arg = pointer_arg_t<P, carray_pointer_tag>;
template<typename P, typename D>
using carray_pointer_binding = pointer_binding_t<P, carray_pointer_tag, D>;
template<typename P>
using static_carray_pointer_binding = static_pointer_binding_t<P, carray_pointer_tag>;

/**
 *  Wrap a pointer of type 'carray' and its deleter function for binding it to a statement.
 *
 *  Unless the deleter yields a nullptr 'xDestroy' function the ownership of the pointed-to-object
 *  is transferred to the pointer binding, which will delete it through
 *  the deleter when the statement finishes.
 */
 template<class P, class D>
 carray_pointer_binding<P, D> bind_carray_pointer(P* p, D d) noexcept {
      return bind_pointer<carray_pointer_tag>(p, std::move(d));
 }
```

```
template<class P>
 static_carray_pointer_binding<P> bind_carray_pointer_statically(P* p) noexcept {
        return bind_pointer_statically<carray_pointer_tag>(p);
 }
```

Use carray pointer type passing

```
// accept and return a pointer of type "carray"
struct pass_thru_pointer_fn {
    using int64_pointer_binding = static_carray_pointer_binding<int64>;

    int64_pointer_binding operator()(carray_pointer_arg<int64> pv) const {
      return rebind_statically(pv);
    }
};
```

It must return a pointer_binding to be recognized by row_extractor<> and for the statement to know how to delete it – if at all… the deletion information is part of the type of the pointer binding, as in delete_int64.

```
// create and return a pointer of type "carray"
struct make_pointer_fn {
    using int64_pointer_binding = carray_pointer_binding<int64, delete_int64>;

    int64_pointer_binding operator()() const {
        return bind_pointer<int64_pointer_binding>(new int64{-1});
    }
};

// return value from a pointer of type "carray"
struct fetch_from_pointer_fn {
    int64 operator()(carray_pointer_arg<const int64> pv) const {
        if(const int64* v = pv) {
            return *v;
        }
        return 0;
    }
};
```

Another application of pointer-binding: error categories and error codes

```
inline constexpr orm_pointer_type auto ecat_pointer_tag = "ecat"_pointer_type;
inline constexpr orm_pointer_type auto ecode_pointer_tag = "ecode"_pointer_type;

using ecat_arg = pointer_arg_t<const std::error_category, ecat_pointer_tag>;
using ecode_arg = pointer_arg_t<std::error_code, ecode_pointer_tag>;

// function returning a pointer to a std::error_category,
// which is only visible to functions accepting pointer values of type "ecat"
struct get_error_category_fn {
    using ecat_binding = static_pointer_binding_t<const std::error_category, ecat_pointer_tag>;

    ecat_binding operator()(unsigned int errorCategory) const {
        size_t idx = min<size_t>(errorCategory, ecat_map.size());
        const error_category* ecat = idx != ecat_map.size() ? &get<const error_category&>(ecat_map[idx]) : nullptr;
            return bind_pointer_statically<ecat_pointer_tag>(ecat);
        }
};
```

In this case, the return type is ecat_binding: a static pointer binding for std::error_category using the ecat_pointer_tag (i.e. a "typed" pointer). Since it is static, ownership is not transferred to Sqlite. Since it is a pointer binding, it works with row_extractor<>.

```
// function accepting a pointer to a std::error_category,
// returns the category's name
struct error_category_name_fn {
    std::string operator()(ecat_arg pv) const {
        if(const error_category* ec = pv) {
            return ec->name();
        }
        return {};
    }
};

// function accepting a pointer to a std::error_category and an error code,
// returns the error message
struct error_category_message_fn {
    std::string operator()(ecat_arg pv, int errorValue) const {
        if(const error_category* ec = pv) {
            return ec->message(errorValue);
        }
        return {};
    }
};

// function returning an error_code object from an error value
struct make_error_code_fn {
    using ecode_binding =
        pointer_binding_t<std::error_code, ecode_pointer_tag, std::default_delete<std::error_code>>;

    ecode_binding operator()(int errorValue, unsigned int errorCategory) const {
        size_t idx = min<size_t>(errorCategory, ecat_map.size());
        error_code* ec = idx != ecat_map.size()
               ? new error_code{errorValue, get<const error_category&>(ecat_map[idx])} : nullptr;
        return bind_pointer<ecode_binding>(ec, default_delete<error_code>{});
    }
};

// function comparing two error_code objects
struct equal_error_code_fn {
    bool operator()(ecode_arg pv1, ecode_arg pv2) const {
        error_code *ec1 = pv1, *ec2 = pv2;
        if(ec1 && ec2) {
            return *ec1 == *ec2;
        }
        return false;
    }
};
```

Before using these scalar functions we need to register them with sqlite_orm:

```
    storage.create_scalar_function<get_error_category_fn>();
    storage.create_scalar_function<error_category_name_fn>();
    storage.create_scalar_function<error_category_message_fn>();
    storage.create_scalar_function<make_error_code_fn>();
    storage.create_scalar_function<equal_error_code_fn>();
```

After inserting some Results into the database we can now query the storage using “typed” pointers passed around via the pointer passing interface that sqlite_orm implements:

```
auto rows =
      storage.select(columns(&Result::id,
          as<str_alias<'o', 'k'>>(c(&Result::errorValue) == 0),
          &Result::errorValue,
          &Result::errorCategory,
          as<str_alias<'e', 'q'>>(func<equal_error_code_fn>(
                                      func<make_error_code_fn>(&Result::errorValue, &Result::errorCategory),
                                      bind_pointer<ecode_pointer_tag>(make_unique<error_code>()))),

func<error_category_name_fn>(func<get_error_category_fn>(&Result::errorCategory)),
func<error_category_message_fn>(func<get_error_category_fn>(&Result::errorCategory),
                                                                   &Result::errorValue)),
                           multi_order_by(order_by(c(&Result::errorValue) == 0),
                                          order_by(&Result::errorCategory),
                                          order_by(&Result::errorValue),
                                          order_by(&Result::id)));
```

This last sample comes from pointer_passing_interface.cpp in the project with the same name in the Examples directory of the source code distribution from Github.com.

SQLite tools

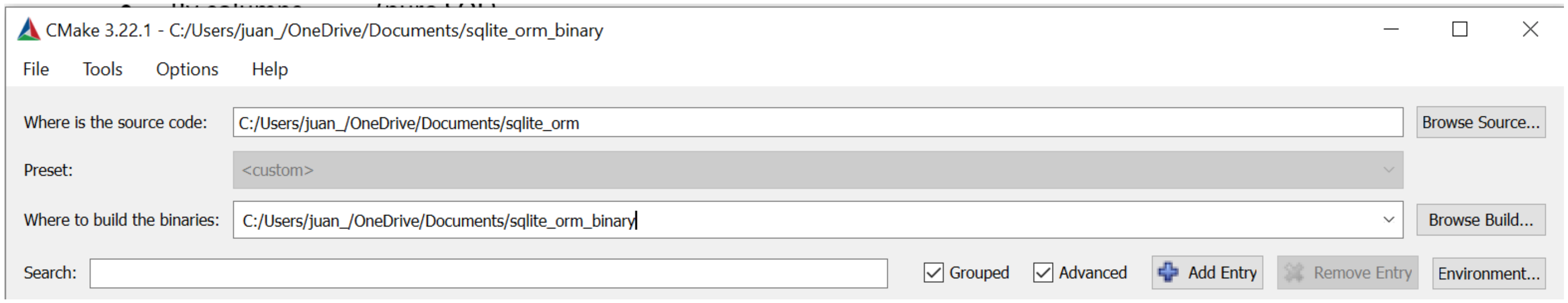
SQLiteStudio and Sqlite3 command shell

GUI open source full featured SQLite client downloadable from SQLiteStudio, runs on Windows, Linux and MacOS X written in C++ using Qt 5.15.2 and SQLite 3.35.4.

Sqlite3.exe command shell and other command line utilities and even source code downloadable from SQLite Download Page.

Installing the sqlite_orm library and DSL

Go to fnc12/sqlite_orm:



Where is the source code: C:/Users/juan_/OneDrive/Documents/sqlite_orm

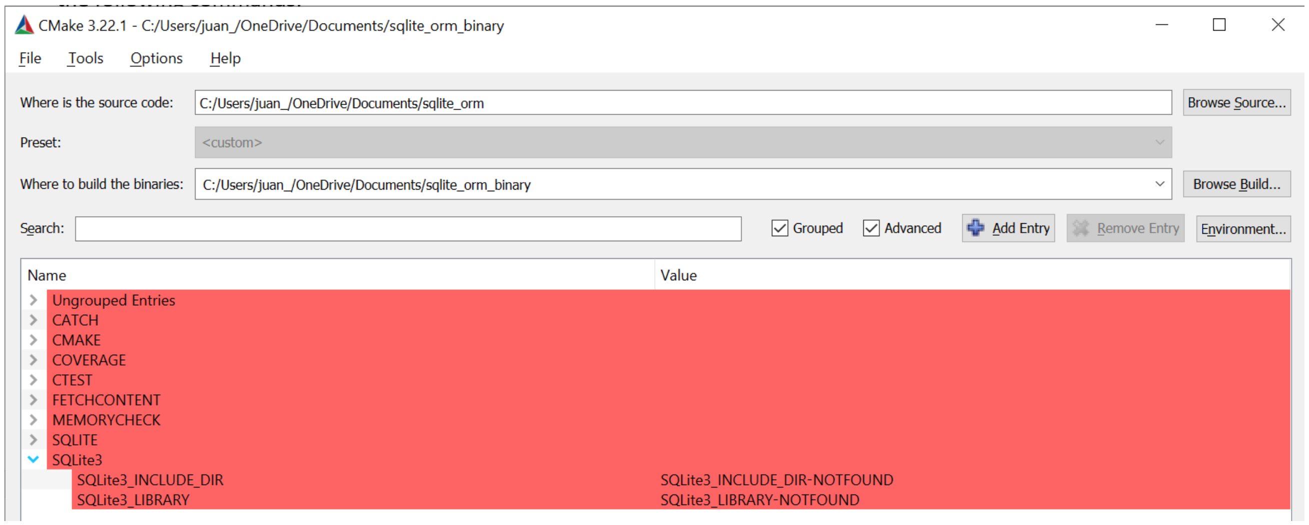
Preset: <custom>

Where to build the binaries: C:/Users/juan_/OneDrive/Documents/sqlite_orm_binary

Search:

2. Use your folder where you placed the source code and create a directory for the binaries (here I have chosen an out of project directory called sqlite_orm_binary).
3. Press configure button. You will get an error regarding the location of sqlite3.h and sqlite3.lib like this:

```
CMake Error at C:/Program Files/CMake/share/cmake-3.22/Modules/FindPackageHandleStandardArgs.cmake:230 (message):
Could NOT find SQLite3 (missing: SQLite3_INCLUDE_DIR SQLite3_LIBRARY)
```



Where is the source code: C:/Users/juan_/OneDrive/Documents/sqlite_orm

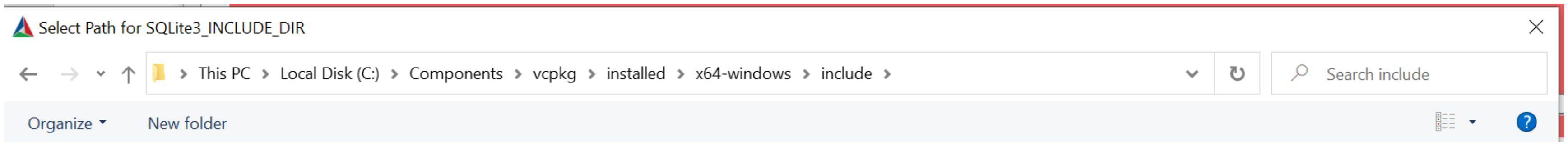
Preset: <custom>

Where to build the binaries: C:/Users/juan_/OneDrive/Documents/sqlite_orm_binary

Search:

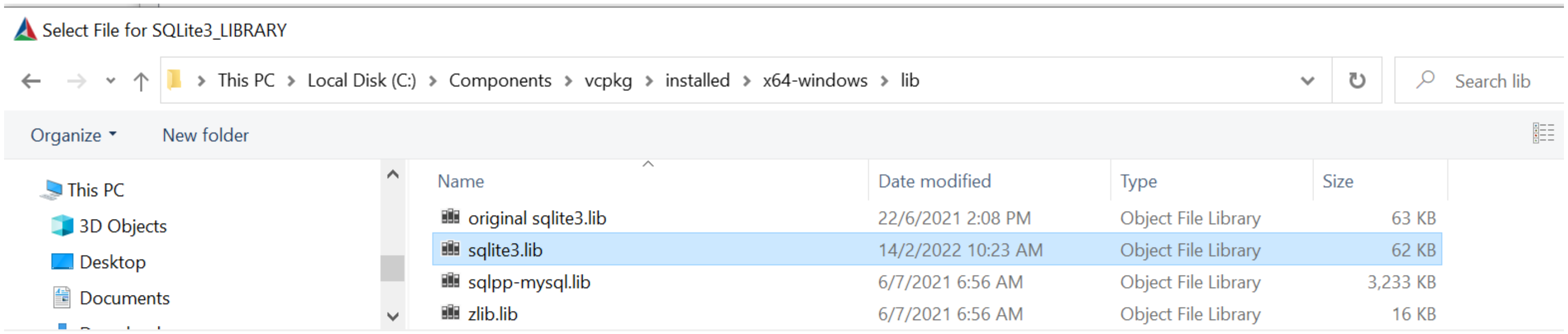
Name	Value
Ungrouped Entries	
CATCH	
CMAKE	
COVERAGE	
CTEST	
FETCHCONTENT	
MEMORYCHECK	
SQLITE	
SQLite3	
SQLite3_INCLUDE_DIR	SQLite3_INCLUDE_DIR-NOTFOUND
SQLite3_LIBRARY	SQLite3_LIBRARY-NOTFOUND

4. Press the right hand button on the SQLite3_INCLUDE_DIR and locate the path where sqlite3.h is found, for example:


Select Path for SQLite3_INCLUDE_DIR

This PC > Local Disk (C:) > Components > vcpkg > installed > x64-windows > include >

5. Then press the right hand button on the SQLite3_LIBRARY and locate the sqlite3.lib, for example:


Select File for SQLite3_LIBRARY

This PC > Local Disk (C:) > Components > vcpkg > installed > x64-windows > lib

Name	Date modified	Type	Size
original sqlite3.lib	22/6/2021 2:08 PM	Object File Library	63 KB
sqlite3.lib	14/2/2022 10:23 AM	Object File Library	62 KB
sqlpp-mysql.lib	6/7/2021 6:56 AM	Object File Library	3,233 KB
zlib.lib	6/7/2021 6:56 AM	Object File Library	16 KB

6. Press Configure again. It should compile without errors.
7. Press Generate and the binaries will be created in the binary chosen folder.
8. You can now go to that folder and open a Visual Studio 2022 .sln file that contains all the unit-tests called: sqlite_orm.sln
9. Open that file in VS 2022 and compile it and run the tests… Everything should work as expected.
10. Your library is now available for use.

Installing SQLite using vcpkg in Windows

Microsoft offers a tool for open source library management called Microsoft/vcpkg available at microsoft/vcpkg: C++ Library Manager for Windows, Linux, and MacOS (github.com) and installable by following instructions at microsoft/vcpkg: C++ Library Manager for Windows, Linux, and MacOS (github.com). After installed, run at the command line the following:

```
> .\vcpkg\vcpkg install sqlite3:x64-windows
```

When you open Visual Studio 2022 the projects created will automatically find sqlite3.dll and sqlite3.lib.

SQLite import and export CSV

It is possible to import and export between comma separated texts and tables. This can be done with the command shell or with the GUI SQLiteStudio program (see Import a CSV File Into an SQLite Table (sqlitetutorial.net) and Export SQLite Database To a CSV File (sqlitetutorial.net)).

SQLite resources

SQLite Resources (sqlitetutorial.net)
SQLite Tutorial - An Easy Way to Master SQLite Fast
SQLite Home Page
SQLite Tutorial - w3resource
SQLite Exercises, Practice, Solution - w3resource
fnc12/sqlite_orm:

Debugging tips

Sync_schema return value

For information as to what storage.sync_schema() has done we can capture its return type which is a std::map like so:

```
auto m = storage.sync_schema[36](true);
std::ostringstream oss;
for (auto& n : m) {
       oss << n.first << " " << n.second << "\t";
}
auto s = oss.str();
```

Access to Generated SQL

For any statement you can obtain the generated SQL with the following steps:

First let's see a SELECT:

```
auto expression = select(columns(&Employee::m_ename, &Employee::m_salary, &Employee::m_commission,
                   &Employee::m_job),
                   order_by(case_<double>()
                          .when(is_equal(&Employee::m_job, "SalesMan"),
                          then(&Employee::m_commission))
                          .else_(&Employee::m_salary).end()).desc());
std::string sql = storage.dump(expression);
auto statement = storage.prepare(expression);
auto rows = storage.execute(statement);
```

[36] For a simulation of what sync_schema would do without performing any changes, try using `sync_schema_simulate(preserve=false)` and examine the return map

Now let’s see an INSERT:

```
auto expression = insert(into<Employee>(),
      columns(&Employee::m_ename, &Employee::m_salary, &Employee::m_commission, &Employee::m_job),
      values(std::make_tuple("Juan", 224000, 200, "Eng")));
std::string sql = storage.dump(expression);
auto statement = storage.prepare(expression);
storage.execute(statement);
```

Now an UPDATE:

```
auto expression = update_all(set(c(&Employee::m_salary) = c(&Employee::m_salary) * 1.3),
                                                    where(c(&Employee::m_job) == "Clerk"));
std::string sql = storage.dump(expression);
auto statement = storage.prepare(expression);
storage.execute(statement);
```

Finally a DELETE:

```
auto expression = remove_all<Employee>(where(c(&Employee::m_empno) == 6));
std::string sql = storage.dump(expression);
auto statement = storage.prepare(expression);
storage.execute(statement);
```

For the object version of these calls, we cannot access so readily the corresponding SQL but it is very predictable:

For object SELECT:

```
auto objects = storage.get_all<Employee>();         // SELECT * FROM EMP
auto employee = storage.get<Employee>(7499);        // SELECT * FROM EMP WHERE id = 7499
```

For object INSERT:

```
// INSERT INTO EMP ( 'ALL COLUMNS EXCEPT PRIMARY KEY COLUMNS' )
// VALUES (  'VALUES TAKEN FROM emp OBJECT')
Employee emp{ -1, "JOSE", "ENG", std::nullopt, "17-DEC-1980", 32000, std::nullopt, 10 };
emp.m_empno = storage.insert(emp);
```

For object UPDATE:

```
//     UPDATE Emp
//     SET
//            column_name = emp.field_name     // for all columns except primary key columns
//            // ....
//     WHERE empno = emp.m_empno;
emp.m_salary *= 1.3;
storage.update(emp);
```

For object DELETE:

```
// DELETE FROM Emp WHERE empno = emp.m_empno
storage.remove<Employee>(emp.m_empno);

// DELETE FROM Emp WHERE 'where clause'
storage.remove_all<Employee>(where(c(&Employee::m_salary) < 1000));

// DELETE FROM Emp
storage.remove_all<Employee>();
```

The Future of sqlite_orm

The following features are not yet currently available in sqlite_orm but are in different degrees of development:

1. Views
2. Window functions
3. Full support for subselects
4. Working with multiple attached databases (aka. Schemas)

References

[CPPTMP,2005] David Abrahams, Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison Wesley, 2005

Author Contact Information

The author of this guide Juan Dent-Herrera can be contacted at juandent@mac.com or by phone at (506) 8718-1237.
Feel free to contact me. I am more than willing to help you with any concern or doubt you may have!