A GUIDE TO SQLITE_ORM FOR SQL USERS

By Juan Dent © 2022 Version 2

March 12, 2022

Table of Contents

Simple query

Select from a single table

Simple calculation:

- Select 10/5;

```
auto rows = storage.select(10/5);
```

```
produces std::vector<int>.
```

- Select 10/5, 2*4;

```
auto rows = storage.select(columns(10/5, 2*4));
```

```
produces std::vector<std::tuple<int, int>>
```

general 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 a,b FROM Table;

```
auto rows = storage.select(columns(&Table::a, &Table::b));
```

```
auto rows = storage.select(columns(&Table::a, &Table::b), from<Table>());
```

- SELECT * FROM Table;

```
auto rows = storage.select(asterisk<Table>());
```

```
produces std::vector<std::tuple<int,int,int>>
```

```
auto objects = storage.get_all<Table>();
```

produces std::vector<Table>

When dealing with large resultsets

We don’t have to load whole result set into memory!

```
    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;
    }
```

Sorting rows

Order by

General syntax:

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

```
auto rows = storage.select(columns(&Table::a, &Table::b), multi_order_by(order_by(&Table::a).asc(),
order_by(&Table::b).desc()));
```

```
auto objects = storage.get_all<Table>(multi_order_by(order_by(&Table::a).asc(), order_by(&Table::b).desc()));
```

Simple order by:

```
SELECT "t"."a", "t"."b" FROM 't' ORDER BY "t"."b" COLLATE NOCASE DESC
```

```
auto rows = storage.select(columns(&Table::a, &Table::b),
order_by(&Table::b).desc().collate_rtrim().collate_nocase());
```

```
SELECT "t"."a", "t"."b", "t"."c" FROM 't'  ORDER BY "t"."b" DESC
auto objects = storage.get_all<Table>(order_by(&Table::b).desc());
```

Dynamic Order by

```
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 statement = storage.prepare(select(columns(&Employee::m_ename, &Employee::m_job),
order_by(substr(&Employee::m_job, length(&Employee::m_job) - 1, 2))));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```

Dealing with NULLs when sorting[1]

```
// SELECT "Emp"."ename", "Emp"."salary", "Emp"."comm" FROM 'Emp' ORDER BY CASE WHEN "Emp"."comm" IS NULL THEN 0 ELSE
// 1 END DESC
auto statement = storage.prepare(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()));

auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```

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

Sorting on a data dependent key

```
auto statement = storage.prepare(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()));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```

[1] See CASE in this document

Filtering Data

Select distinct

SELECT DISTINCT select_list FROM table;

Assume Employee::name is nullable:

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

std::vector<std::optional<std::string>> expected;

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

std::vector<std::tuple<std::optional<std::string>>> expected;
```

Where

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

Logical operator that tests whether a value is inside a range of values including the boundaries. 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 })));


// 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 statement = storage.prepare(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(
      select(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")
)))))));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);

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 statement = storage.prepare(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"))))));
```

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

```
auto rows = storage.select(columns(&Employee::lastName), 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 statement = storage.prepare(select(columns(coalesce<double>(&Employee::m_commission, 0),
&Employee::m_commission)));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```

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)]

| (join-table-expression)

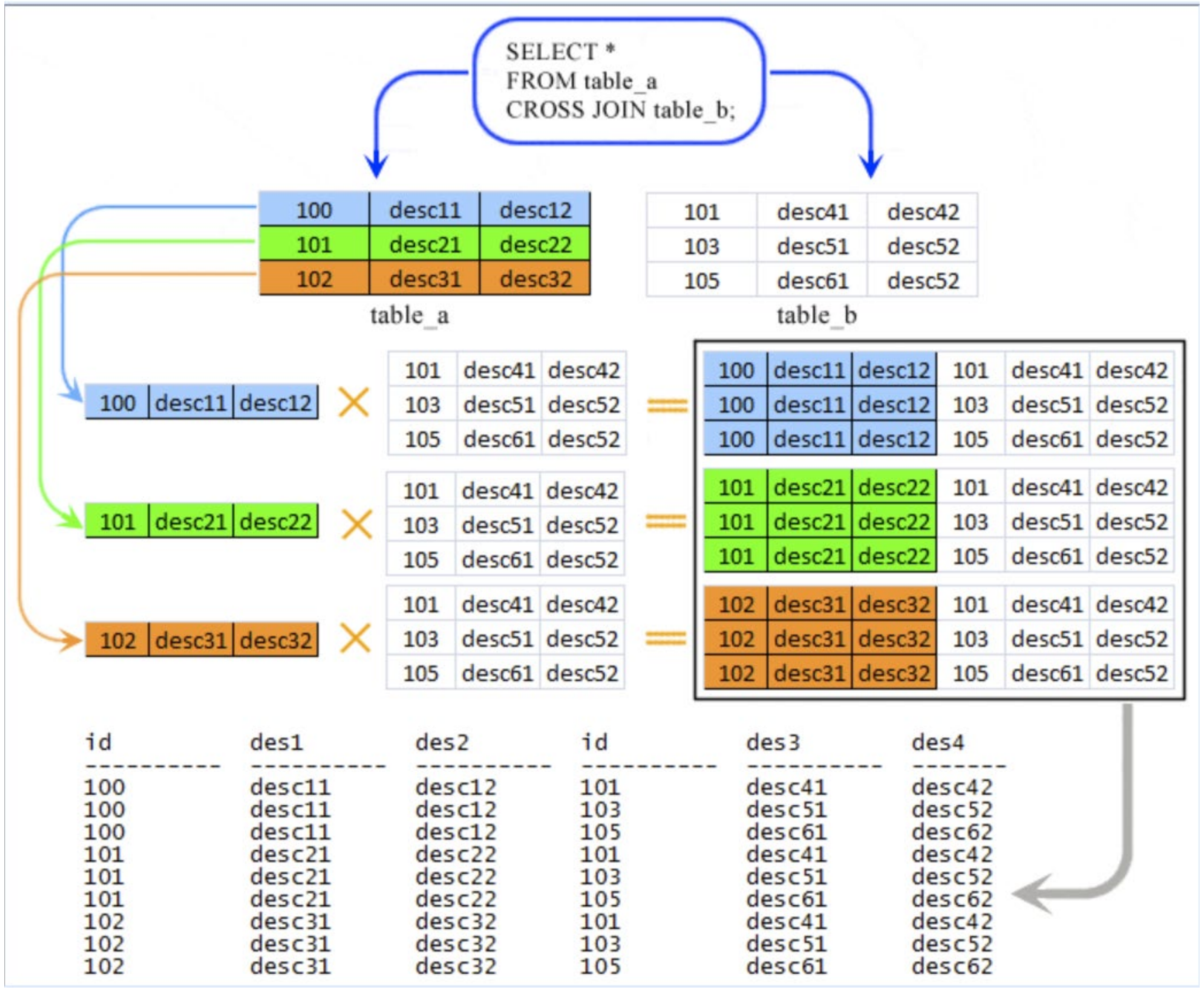
SQLite join

In SQLite to query data from more than one table you can use INNER JOIN, LEFT JOIN or CROSS JOIN. 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)



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

100	desc11	desc12
101	desc21	desc22
102	desc31	desc32

table_a

101	desc41	desc42
103	desc51	desc52
105	desc61	desc52

table_b

100	desc11	desc12	101	desc41	desc42
100	desc11	desc12	103	desc51	desc52
100	desc11	desc12	105	desc61	desc52
101	desc21	desc22	101	desc41	desc42
101	desc21	desc22	103	desc51	desc52
101	desc21	desc22	105	desc61	desc52
102	desc31	desc32	101	desc41	desc42
102	desc31	desc32	103	desc51	desc52
102	desc31	desc32	105	desc61	desc52

```
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[2]
- LEFT [OUTER]
- RIGHT [OUTER]
- FULL [OUTER]
- UNION[3]

With the following restrictions:

- NATURAL and UNION cannot both be specified
- If either NATURAL or UNION is specified, neither an ON clause nor a USIGN 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:

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

(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.

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,B WHERE condition

UNION ALL

SELECT A.*, NULL, NULL, …,NULL FROM A WHERE A.pkey NOT IN (SELECT A.pkey FROM A,B WHERE condition)

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.

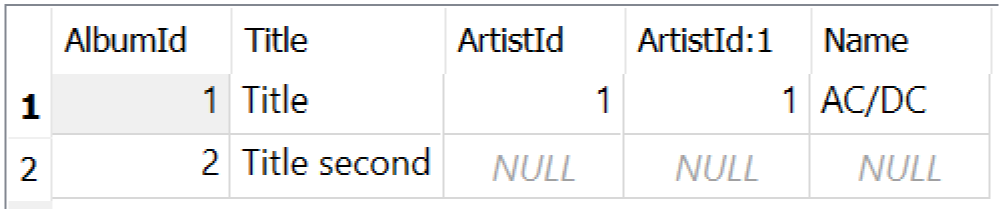
For case 2, let the commalist of columns in the USING clause be C1, C2,…, Cn, all Ci unqualified and identifying a common column of A and B. Then the case becomes identical to a case 1 in which the condition has the form:

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:

select albums.*, artists.* from albums LEFT JOIN artists ON albums.ArtistId = artists.ArtistId



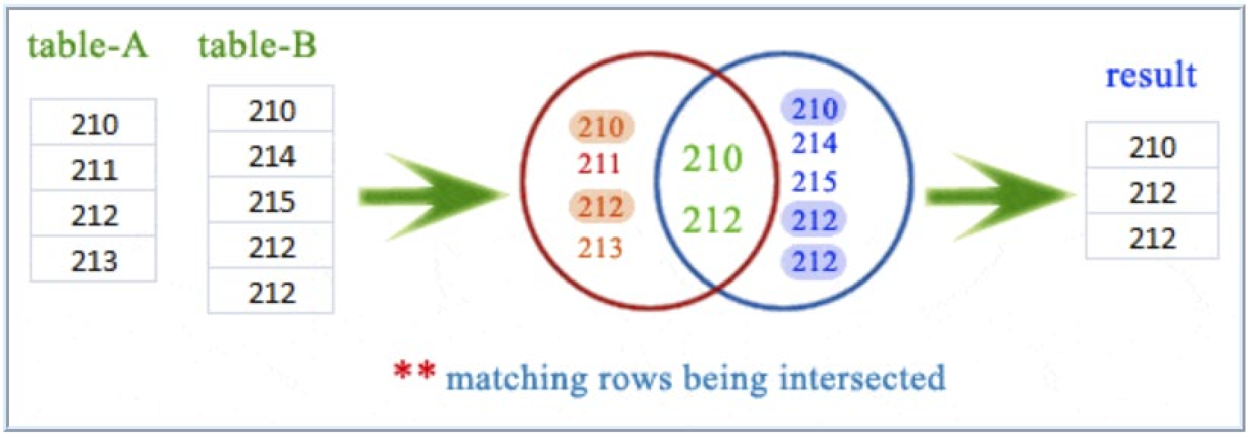
	AlbumId	Title	ArtistId	ArtistId:1	Name
1	1	Title	1	1	AC/DC
2	2	Title second	NULL	NULL	NULL

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

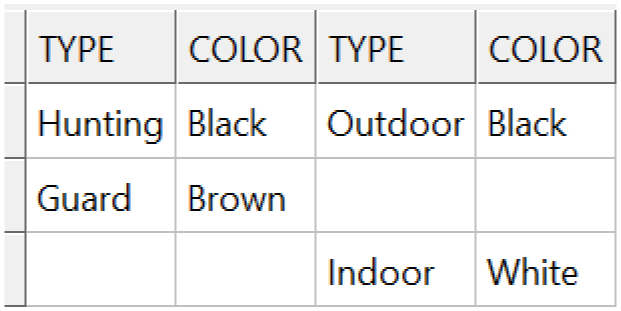
Self join

```
//  SELECT m.FirstName || ' ' || m.LastName,
//      employees.FirstName || ' ' || employees.LastName
//  FROM employees
//  INNER JOIN employees m
//  ON m.ReportsTo = employees.EmployeeId
using als = alias_m<Employee>;
auto firstNames = storage.select(
            columns(c(alias_column<als>(&Employee::firstName)) || " " || c(alias_column<als>(&Employee::lastName)),
                    c(&Employee::firstName) || " " || c(&Employee::lastName)),
            inner_join<als>(on(alias_column<als>(&Employee::reportsTo) == c(&Employee::employeeId))));
```

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

```
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!

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

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

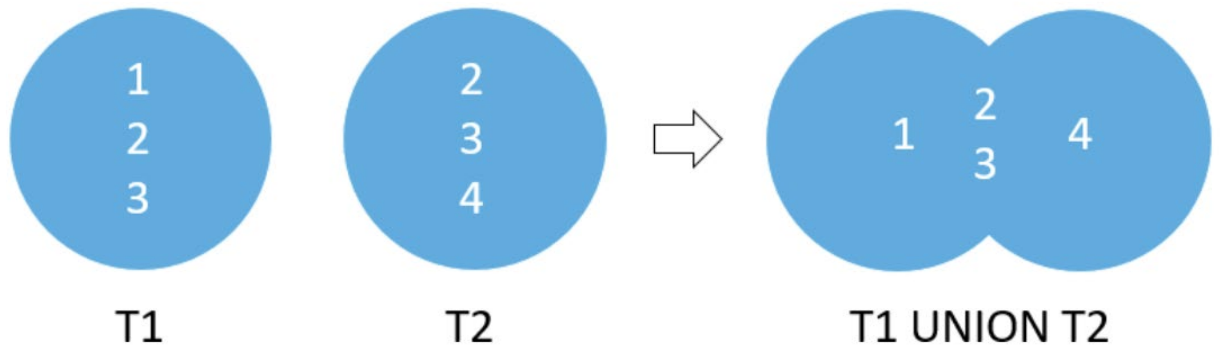
Union

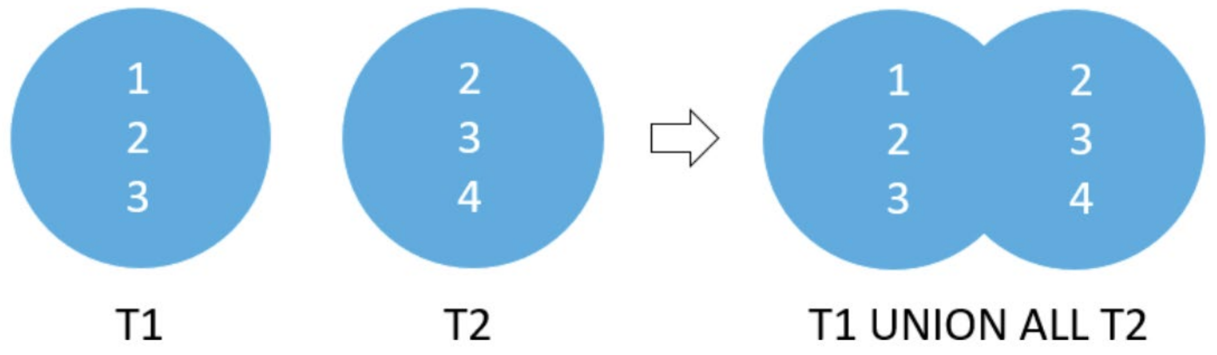
The difference between UNION and JOIN Is that the JOIN clause combines columns from multiple related tables while UNION combines rows from multiple similar tables.

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 atop 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 statement = storage.prepare(
      select(union_all(
             select(columns(as<NamesAlias>(&Department::m_deptname), as_optional(&Department::m_deptno))),
             select(union_all(
                   select(columns(quote("--------------------"), std::optional<int>())),
                   select(columns(as<NamesAlias>(&Employee::m_ename), as_optional(&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

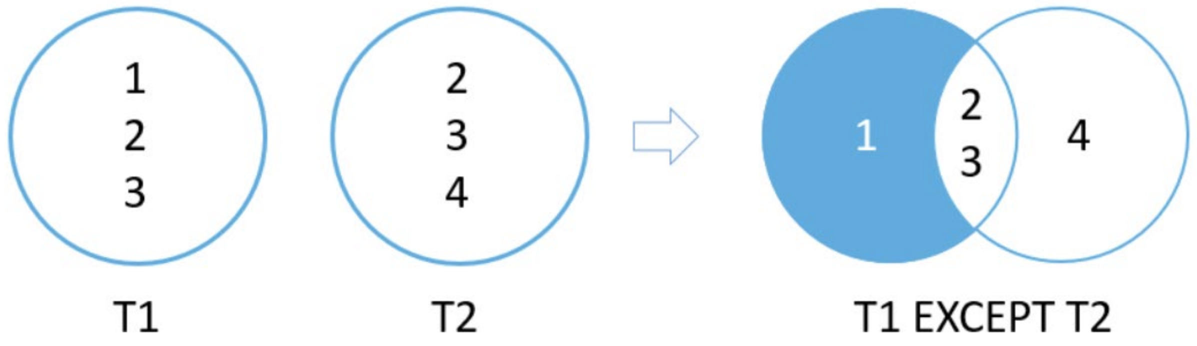
Find all the dept_id in dept_master but not in emp_master:

```
//  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 statement = storage.prepare(select(except(select(&Artist::m_id), select(&Album::m_artist_id))));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```



Intersect

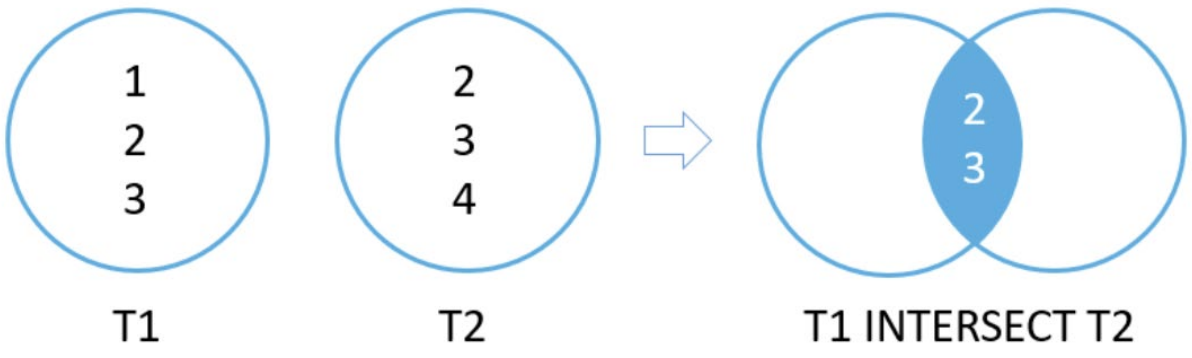
Compares the result sets of 2 queries and returns distinct rows that are output by both queries.

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)));
```

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);

```
//  SELECT first_name, last_name, salary
//  FROM employees
//  WHERE salary >(
//          SELECT salary
//          FROM employees
//          WHERE first_name='Alexander');
auto rows = storage.select(
        columns(&Employee::firstName, &Employee::lastName, &Employee::salary),
        where(greater_than(&Employee::salary,
             select(&Employee::salary,
             where(is_equal(&Employee::firstName, "Alexander"))))));


//  SELECT employee_id,first_name,last_name,salary
//  FROM employees
//  WHERE salary > (SELECT AVG(SALARY) FROM employees);
auto rows = storage.select(columns(
        &Employee::id, &Employee::firstName, &Employee::lastName, &Employee::salary),
        where(greater_than(&Employee::salary,
             select(avg(&Employee::salary)))));


//  SELECT first_name, last_name, department_id
//  FROM employees
//  WHERE department_id IN
//      (SELECT DEPARTMENT_ID FROM departments
//      WHERE location_id=1700);
auto rows = storage.select(
            columns(&Employee::firstName, &Employee::lastName, &Employee::departmentId),
            where(in(&Employee::departmentId, select(&Department::id, where(c(&Department::locationId) == 1700)))));


//  SELECT first_name, last_name, department_id
//  FROM employees
//  WHERE department_id 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)));


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

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


// SELECT 'e'."LAST_NAME", 'e'."SALARY", 'e'."DEPARTMENT_ID" FROM 'employees' 'e'
// WHERE (('e'."SALARY" > (SELECT (AVG("employees"."SALARY")) FROM 'employees',
// 'employees' e WHERE (("employees"."DEPARTMENT_ID" = 'e'."DEPARTMENT_ID")))))

using als = alias_e<Employee>;
auto rows = storage.select(
      columns(alias_column<als>(&Employee::lastName),
      alias_column<als>(&Employee::salary),
      alias_column<als>(&Employee::departmentId)),
      from<als>(),
      where(greater_than(
      alias_column<als>(&Employee::salary),
      select(avg(&Employee::salary),
      where(is_equal(&Employee::departmentId, alias_column<als>(&Employee::departmentId)))))));


//  SELECT first_name, last_name, employee_id, job_id
//  FROM employees
//  WHERE 1 <=
//      (SELECT COUNT(*) FROM Job_history
//      WHERE employee_id = employees.employee_id);
auto statement = storage.prepare(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. Syntax:

EXISTS (subquery)

The subquery is a select statement that returns 0 or more rows.

```
//  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 statement = storage.prepare(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)));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);


//  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 statement = storage.prepare(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)))))));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```

```
/*
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 statement = storage.prepare(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))));

auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```

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 statement = storage.prepare(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))));
auto sql = statement.expanded_sql();
auto rows = storage.execute(statement);
```

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()));
```

Aliases for columns and tables

For tables:

```
//  SELECT C.ID, C.NAME, C.AGE, D.DEPT
//  FROM COMPANY AS C, DEPARTMENT AS D
//  WHERE  C.ID = D.EMP_ID;
using als_c = alias_c<Employee>;
using als_d = alias_d<Department>;
auto rowsWithTableAliases = storage.select(columns(
      alias_column<als_c>(&Employee::id),
      alias_column<als_c>(&Employee::name),
      alias_column<als_c>(&Employee::age),
      alias_column<als_d>(&Department::dept)),
     where(is_equal(alias_column<als_c>(&Employee::id), alias_column<als_d>(&Department::empId))));
```

For columns:

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

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

//  SELECT COMPANY.ID as COMPANY_ID, COMPANY.NAME AS COMPANY_NAME, COMPANY.AGE, DEPARTMENT.DEPT
//  FROM COMPANY, DEPARTMENT
//  WHERE COMPANY_ID = DEPARTMENT.EMP_ID;
auto rowsWithColumnAliases = storage.select(columns(
	as<EmployeeIdAlias>(&Employee::id),
	as<CompanyNameAlias>(&Employee::name),
	&Employee::age,
	&Department::dept),
		where(is_equal(get<EmployeeIdAlias>(), &Department::empId)));
```

For columns and tables:

```
//  SELECT C.ID AS COMPANY_ID, C.NAME AS COMPANY_NAME, C.AGE, D.DEPT
//  FROM COMPANY AS C, DEPARTMENT AS D
//  WHERE  C.ID = D.EMP_ID;
auto rowsWithBothTableAndColumnAliases = storage.select(columns(
	as<EmployeeIdAlias>(alias_column<als_c>(&Employee::id)),
	as<CompanyNameAlias>(alias_column<als_c>(&Employee::name)),
	alias_column<als_c>(&Employee::age),
	alias_column<als_d>(&Department::dept)),
		where(is_equal(alias_column<als_c>(&Employee::id), alias_column<als_d>(&Department::empId))));
```

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 >=80 THEN 'A+'
//      WHEN MARKS >=70 THEN 'A'
//      WHEN MARKS >=60 THEN 'B'
//      WHEN MARKS >=50 THEN 'C'
//      ELSE 'Sorry!! Failed'
//      END as 'Grade'
//      FROM STUDENT;
auto rows = storage.select(columns(
       &Student::id,
       &Student::name,
       &Student::marks,
       as<GradeAlias>(case_<std::string>()
              .when(greater_or_equal(&Student::marks, 80), then("A+"))
              .when(greater_or_equal(&Student::marks, 70), then("A"))
              .when(greater_or_equal(&Student::marks, 60), then("B"))
              .when(greater_or_equal(&Student::marks, 50), then("C"))
              .else_("Sorry!! Failed")
              .end())));
```

Changing data

Insert

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());
```

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)));

// INSERT INTO 'Invoices' ("customerId") VALUES (NULL)
Invoice inv{ -1, 1, std::nullopt };
auto statement = storage.prepare(
      insert(inv, columns(&Invoice::customerId)));
auto sql = statement.expanded_sql();
storage.execute(statement);
```

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'


auto statement = storage.prepare(
      insert(into<User>(),
      select(columns(&UserBackup::id, &UserBackup::name, &UserBackup::hash))));
auto sql = statement.expanded_sql();
storage.execute(statement);
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.

For insert and update commands[4], 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 with SQLITE_CONSTRAINT error and the current SQL statement backs out 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 back out 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 of the UNIQUE or PRIMARY KEY type, 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 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[5] are enabled.

```
auto rows = storage.insert(or_abort(),
        into<User>(),
```

[4] The on conflict clause is used in the create table command with the same semantics

[5] See PRAGMA recursive_triggers

```
        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;

```
//  '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 products
//  SET quantity = 5 WHERE id = 1;
```

```
storage.update_all(set(
      c(&Product::quantity) = 5),
      where(c(&Product::id) == 1));
```

// 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);
}


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


//  UPDATE contacts
//  SET phone = REPLACE[6](phone, '410', '+1-410')
storage.update_all(set(
      c(&Contact::phone) = replace(&Contact::phone, "410", "+1-410")));
```

Delete

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

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

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


//  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:

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.

NOTE: Changes to the database are faster if done within a transaction.

```
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});
        storage.replace(Student{4, "Jagruti", "jagu@gmail.com", 30});
        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);
      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;
```

Data definition

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[7] 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;
    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();
```

[7] As in relational normalization

```
}
```

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.

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

struct UserVisit {
    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()),
                    make_column("first_name", &User::firstName),
                    make_column("last_name", &User::lastName),
                    primary_key(&User::id, &User::firstName)),
             make_table("visits",
                   make_column("user_id", &UserVisit::userId),
                   make_column("user_first_name", &UserVisit::userFirstName),
                   make_column("time", &UserVisit::time),
                   foreign_key(&UserVisit::userId, &UserVisit::userFirstName)
                          .references(&User::id, &User::firstName)));
     storage.sync_schema();
}
```

This defines a compound foreign key and a corresponding compound primary key.

AUTOINCREMENT constraint

```
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, autoincrement(), primary_key()),
                                   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 and UNIQUE 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.

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 invoiceDatw to be the current date at the moment of insertion.

PERSISTENT collections

```
/**
 *  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

```
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();
}
```

This one implements the structure for a self-join.

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[8]] [INSERT|UPDATE|DELETE]
  ON table_name
  [WHEN condition]
```

[8] Only allowed for views

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

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())
```

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.

Sqlite_orm library and DSL downloadable from fnc12/sqlite_orm at dev (github.com). Do the following git command from a Visual Studio 2022 console:

Git clone –branch dev https://github.com/fnc12/sqlite_orm.git sqlite_orm_dev_download_date

Where 'download_date' is the date you download it (e.g. March_1_2022).

Installiing SQLite using vcpkg

Install Microsoft/vcpkg from microsoft/vcpkg: C++ Library Manager for Windows, Linux, and MacOS (github.com) by following the 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: ♡ SQLite ORM light header only library for modern C++ (github.com)

The Future of sqlite_orm

The most important features missing from sqlite_orm currently are support for ***views*** and ***common table expressions***, in particular as represented by the WITH clause (see The WITH Clause (sqlite.org)).

An example of a dynamic from (which exemplify common table expressions) follows:

```
select depno, sum(salary) as total_sal, sum(bonus) as total_bonus from
(
         select e.empno,
         e.ename,
         e.salary,
         e.deptno,
         b.type,
         e.salary * case
         when b,type = 1 then .1
         when b.type = 2 then .2
         else .3
         end as bonus
         from emp e, emp_bonus b
         where e.empno = b.empno
         and e.deptno =20
) y
group by deptno
```

and an example of a WITH clause follows:

```
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;
```