MySQL Notes

Iñaki Lakunza

June 11, 2024

Contents

1 Introduction

A **Database** is a collection of data stored in a format that can easily be accessed.

In order to access the database we use a software application named **Database Management System (DBMS)**. We connect tot the DBMS and give it instructions for querying or modifying data. The DBMS will execute our isntructions and send the results back.

There are two main DBMS categories: **Relational** and **NonRelational (NoSQL)**.

In **Relational** Databases we store data in tables that are linked to each other using relationships. Each table stores data about a specific type of object.

SQL (or Sequel) is the language we use to work with these relational DBMS.

In **NonRelational** databases we do not have tables or relationships.

2 The Select Statement

The first step to query data from a database is to select a database.

USE is a keyword to select the database we will be working. We can also select it by double clicking it on the workbench.

With the **SELECT** keyword we will select which columns we want to retrieve. We can then specify which columns we want, or we can select all using *.

Then we have to choose from which table we want to retrieve the columns, using the **FROM** keyword.

Whenever we have multiple SQL statements we have to terminate eacy statement using a semi-column

```
USE sql_store;

SELECT *
FROM customers
```

And, if we want to run the script we can go to query and press to *execute*. Or, in Windows, we can press *ctrl+mayus+enter*.

If we want to pick rows which have some specific attributes we can use the **WHERE** clause.

For instance, if we execute the following code we will just retrieve the row with *customer_id* equal to 1.

```
USE sql_store;

SELECT *
FROM customers
WHERE customer_id = 1
```

We can also sort the data using **ORDER BY**. We have to specify the column which will be used to sort the data.

By default it will be sorted by **ascending order**, but we can do it in descending order using **DESC**.

(**For clarity, we should always specify how we want to sort it, using ASC or DESC**).

We can use – for writting comments, and if we want to write comments which are larger than one row we can use * and */.

```
USE sql_store;

SELECT *
FROM customers
-- WHERE customer_id = 1
ORDER BY first_name DESC
```

3 The SELECT Clause

Apart from the original columns, we can retrieve different combinations of them.

We can combine the tables between them or we can use different values for that.

For instance, with the following code we will retrieve the column points and another column with the name *points + 10*, which will have the values of the table *points* summed by 10:

```
SELECT points, points + 10
FROM customers
```

We have addition, substraction, multiplication (*), division (/), modulo (%).

If our line is getting to long, we can divide it in different rows. And, we have to take into account that if not specified, the name of the column will be the line used to get it, but we can change it using **AS**:

```
SELECT
    last_name,
    first_name,
    points,
    (points + 10) * 100 AS 'discount␣factor'
FROM customers
```

We may have repeated elements. If we want to get just UNIQUE elements, we can use the **DISTINCT** keyword:

```
SELECT DISTINCT state
FROM customers
```

4 The WHERE clause

We can use the **WHERE** clause to filter data.

These are the operators that we have:

$>, >=, <, <=, =, <>$ $(or! =)$.

For text, we have to enclose it in single quotes (') or double quotes (").

When working with DATES, even though they are not text, we have to use quotes to compare them:

```
SELECT *
FROM customers

WHERE birth_date > '1990-01-01'
```

5 The AND, OR and NOT Operators

For combining conditions:

```
SELECT *
FROM customers

WHERE birth_date > '1990-01-01' AND points > 1000
```

And the **OR** clause works similarly.

We have to remember that the AND operator is always evaluated faster than the OR operator (similarly to + and *) But we can always use parenthesis to better define the wanted order.

The NOT operator is used to negate a condition.

6 The IN Operator

If we want to use a conditions with multiple ORs, we could write it the following way:

```
SELECT *
FROM Customers
WHERE state = 'VA' OR state = 'GA' OR state = 'FL'
```

But this code is not clean and is quite tedious.

Instead, we can use the **IN** operator:

```
SELECT *
FROM Customers
WHERE state IN ('VA', 'FL', 'GA')
```

And we can also use the NOT operator, using **NOT IN**.

7 The BETWEEN Operator

Instead of:

```
WHERE points >= 1000 AND points <= 3000
```

We can write:

```
WHERE points BETWEEN 1000 AND 3000
```

8 The LIKE Operator

We can use the **LIKE** operator to retrieve elements which follow a concrete pattern.

For that purpose we use the % character, it will define a pattern, it does not matter if it is uppercase of lowercase.

The % symbol does not have to be at the end of the text, it can be placed before or/and after.

On the other hand, we can use _ for just character, in that position there will just be able to place a single character (we can use multiple _).

```
WHERE last_name LIKE '\%a\%'
WHERE last_name LIKE '___'
```

9 The REGEXP Operator

The following 2 lines are identical:

```
WHERE last_name LIKE '\%field\%'
WHERE last_name REGEXP 'field'
```

If we want for instance to start with a chosen word, we can use the ˆ symbol, so **we represent the beginning of a string**.

And the same way, we can use a dollar sign $ to represent the end of a string:

```
WHERE last_name REGEXP '^field' -- The last name must begin with 'field'
WHERE last_name REGEXP 'field$' -- The last name must end with 'field '
$
```

And if we want to use different patterns we can use |:

```
WHERE last_name REGEXP '^field|mac|rose'
```

So, it should start with 'field', or have 'mac' or 'rose'.

On the other hand, we can do different options, for instance, if we want to find last names which contain 'ge' or 'ie' or 'me':

```
WHERE last_name REGEXP '[gim]e'
WHERE last_name REGEXP '[a-h]e' -- Instead of [abc...h]
```

10 The IS NULL Operator

We will have some missing elements, these will be 'filled' with a 'NULL'.
NULL means the absence of a value.

For instance, if we want to get the customers which do not have a phone number associated:

```
SELECT *
FROM customers
WHERE phone IS NULL
```

And obviously we can use **IS NOT NULL** the same way.

11 The ORDER BY Operator

In RELATIONAL databases every table should have a primary key column, and the values in that column should UNIQUELY identify (no repeated values) the records in that table. And by default the elements will be sorted using these values.

If we want to sort the element using other values we can use the **ORDER BY** Operator.

And as we know by default the sorting will be done in ascending order.
And we use **ASC** (optional) and **DESC** to sort the elements the way we want.

We can use more than one field to sort the elements, this will be useful if we have repeated elements in the first field, so those will be sorted using the second field.

And we can use different sorting orders:

```
ORDER BY state DESC, first_name ASC
```

We can also sort the elements using the indixes in which we retrieve them:

```
SELECT first_name, last_name
FROM customers
ORDERY BY 1, 2
```

This approach works, **but it is not recommended, it is always better to use the specific names**, since the code may change and we may retrieve different elements in different orders, so by using the names we avoid problems.

12 The LIMIT Operator

The **LIMIT** clause is used to limit how many elements we want to retrieve as a maximum (if we have less then just those will be retrieved).

Then, we can use an offset to pick the elements we want:

```
LIMIT 3    -- Retrieve first 3 elements
LIMIT 6, 3 -- Jump first 6 and retrieve next 3 elements
```

13 Inner Joins

So far we have just selected elements from a single table. But in the real world most probably we will want to work with multiple tables.

If we want to combine elements from different tables we use the **JOIN** keyword (we can also tyoe **INNER JOIN**).
There also exist another type of join: **OUTER JOIN**.
And we can optionally type **INNER JOIN**.

Here is an example:

```
SELECT *
FROM orders
JOIN customers
    ON orders.customer_id = customers.customer_id
```

The first fields that will be retrieved will belong to the *orders* table, and then we will have the elements from the *customers* table..
If we simplify it:

```
SELECT order_id, first_name, last_name
FROM orders
JOIN customers
    ON orders.customer_id = customers.customer_id
```

We have the field *customer_id* in both tables, so we can not select this field directly, we get an ambiguity error. WE HAVE TO USE A PREFIXING WITH A TABLE NAME:

```
SELECT order_id,
    orders.customer_id,
    first_name,
    last_name
FROM orders
JOIN customers
    ON orders.customer_id = customers.customer_id
```

(In this example it does not matter from which table we retrieve the *customer_id* field, since we have chosen the condition to the join to be equal in both tables.)

We can make simpler our code by avoiding the repetitions of the table names, by using **aliases**:

```
SELECT order_id,
    o.customer_id,
    first_name,
    last_name
FROM orders o     -- alias: o
JOIN customers c -- alias: c
    ON o.customer_id = c.customer_id
```

We always have to use ON, since we have to tell MySQL in which order to join the elements, so, in the tables we use a field to mark the correspondence with the elements from other tables, in this example it has been *customer_id*.

14 Joining Across Databases

We will have to use a prefix for the other database we will be working with:

```
SELECT *
FROM order_items oi -- alias: oi
JOIN sql_inventory.products -- using database sql_inventory
    ON oi.product_id = p.product_id
```

If we would do this by using the *sql_inventory* database as the main database we would do the following:

```
USE sql_inventory;

SELECT *
FROM sql_inventory.order_items oi
JOIN products p
    ON oi.product_id = p.product_id
```

15 Self Joins

We can also join a table with itself. In order to do this we have to use **JOIN**, **but repeating the table using another alias**:

```
USE sql_hr;

SELECT *
FROM employees e -- employees
JOIN employees m -- managers
    ON e.reports_to = m.employee_id
```

In this example, the original table has the employee_id and a field known reports_to. So, all the employees in this simple example reported to the same person (id: 37270), and the person with id 37270 had Null in the field reports_to.

After joining the table with itself this way, the output has all the fields of the original table, and then it has again the fields, but in all cases it tells the values for the person which report to (in this example to the same person (37270)).

If we want to simplify even more the output:

```
USE sql_hr;

SELECT
    e.employee_id,
    e.first_name,
    m.first_name AS manager
FROM employees e
JOIN employees m
    ON e.reports_to = m.employee_id
```

So now we would just get the employee_id, the employee's id and the first name of the person who they have to report to.

So, joining a table with itself is almost the same as doing it with another table, **but we have to use different aliases**.

16 Joining Multiple Tables

Obviously, if a table has a connection with more than 1 table, it will have more than 1 field which relates the elements from our table with other tables. One relating field for one connection.

```
USE sql_store;

SELECT
    o.order_id,
    o.order_date,
    c.first_name,
    c.last_name,
    os.name AS status
FROM orders o
JOIN customers c
    ON o.customer_id = c.customer_id
JOIN order_statuses os
    ON o.status = os.order_status_id
```

In the real world we will end up joining a large number of tables.

17 Compound Join Conditions

In all the examples we have seen so far we use a single column to uniquely identify the rows in a given table. For instance the *customer_id*.

But there are times where we won't be able to use a single column to uniquely identify elements in a given table.

So, we will use more than one condition to join our tables:

```
SELECT *
FROM order_items oi
JOIN order_item_notes oin
    ON oi.order_id = oin.order_id
    AND oi.product_id = oin.product_id
```

18 Outer Joins

So far we have just seen Inner Joins (the one that it is done by default, so if we just write JOIN then we will do Inner Join).

For instance, if we use the following code **we will just see customers WHICH HAVE AN ORDER, the rest will not appear**:

```
SELEECT
    c.customer_id,
    c.first_name,
    o.order_id
FROM customers c
JOIN orders o
    ON c.customer_id = o.customer_id
ORDER BY c.customer_id
```

The reason because we have just seen customers that have an order is the ON condition.
By joining those two tables we are only returning records that match this condition.
To solve this problem we use an **OUTER JOIN**.

In QSL two types of Outer Joins can be made: LEFT JOIN and RIGHT JOIN.

When we use a LEFT JOIN, all the records from the left table (the one we choose in FROM) are returned whether the condition imposed in ON is true or not.
In the following example all the elements in the *customers* table will appear, and if they do not have a *order_id* then a NULL value will appear:

```
SELECT
    c.customer_id,
    c.firts_name,
    o.order_id
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id
ORDER BY c.customer_id
```

On the other hand, if we use a **RIGHT JOIN**, the elements in the second table (the one chosen with JOIN) will appear, whether they fulfill the condition or not:

```
SELECT
    c.customer_id,
    c.firts_name,
    o.order_id
FROM customers c
RIGHT JOIN orders o
    ON c.customer_id = o.customer_id
ORDER BY c.customer_id
```

19 The USING Clause

When our MySQL code gets larger and larger it gets more and more tedious to write and read lines such as:

```
    ON o.customer_id = c.customer_id
```

So, in this cases, where in both tables the name of the field is the same, we can replace the ON clause with the **USING** clause:

```
SELECT
    o.order_id,
    c.first_name,
    sh.name AS shipper
FROM orders o
JOIN customers c
    -- ON o.customer_id = c.customer_id
    USING (customer_id)
LEFT JOIN shippers sh
    -- ON o.shipper_id = sh.shipper_id
    USING (shipper_id)
```

We have to remember that in order to be able to use **USING**, the field name in both tables must be the same.

And, if we want to use more than one condition:

```
SELECT *
FROM order_items oi
JOIN order_item_notes oin
    USING (order_id, product_id)
```

We can also use something called **NATURAL JOIN**, when using it MySQL will find the field name which is used in both tables and use it for joining them.
But this should be avoided, since it may produce unpredictable results, it is always better to write things explicitly.

20 Cross Joins

We use Cross Joins to join or combine every record from the first (left) table with every record in the second table (right).

So, for example, if we have 10 elements in the first table and 10 in the second, the output will have 100 elements, since every element in the first table will be joined with every element from the second table.

Since we are joining all the elements, we do not have to specify a condition with ON.

21 UNIONS

With JOINs we can combine elements from multiple tables, but in SQL we can also **combine rows with multiple tables**.

For instance, we may have a field which defines the order_date, and we want to add a label: if the order is placed in the current year, the table is going to be active and if the order is placed in previous years, then we want the label to be archived:

```
SELECT
    order_id,
    order_date,
    'Active' -- All the elements will have
             -- the same attribute in this new field:
             -- It is the name of the field, so in this case Active
```

But, if we want to have *Active* by default in the attributes, but change the name of the field, we can do the following:

```
SELECT
    order_id,
    order_date,
    'Active' AS status
FROM orders
WHERE order_date >= '2019-01-01'
```

(**The date is hard-coded, we will see later how to be able to change this year after year automatically**).

So, if we want to get all, the active ones and the archived ones:

```
SELECT
    order_id,
    order_date,
    'Active' AS status
FROM orders
WHERE order_date >= '2019-01-01'
UNION
```

```
SELECT
    order_id,
    order_date,
    'Archived' AS status
FROM orders
WHERE order_date < '2019-01-01'
```

22 Column Attributes

We can click on the following icon to have a deeper look at the table:

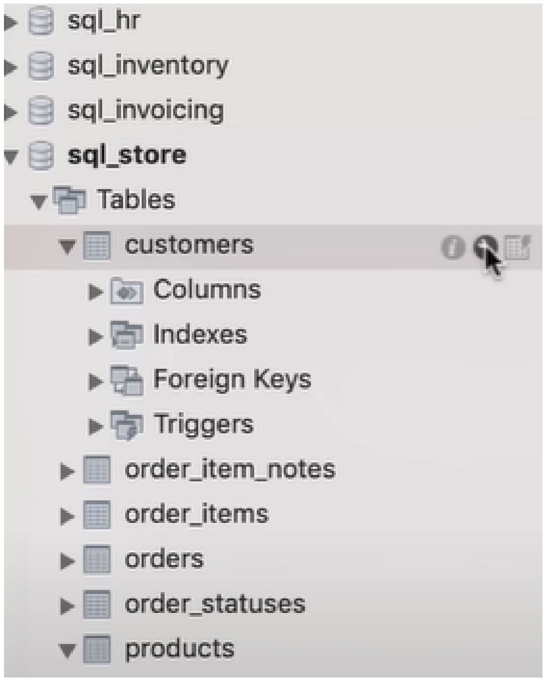

Figure 1:

We can see the datatype accepted by the column in there.
The parenthesis in the datatypes indicate the length of the attributes that can be placed there.

The best practice is to use VARCHARs to store strings or textual values.

PK is short for primary key, the values in that field uniquely identify the elements in the table.
NN is short for Not Null, it determines if the column can accept null values or not.
AI is short for Auto Increment, and it is normally used for primary key columns, every time we insert a new record in the table, we let MySQL insert the value in the column.
The last column specifies the default value, the one which will be placed if we do not specify a value.

23 Inserting a Single Row

In order to insert a record in the table we will use the **INSERT INTO** quote. Then we specify in which table we will insert it.

Then we have to use the **VALUES** clause. Here we have to supply values for all the fields (the key field is optional, and the ones that have default values do not have to be specified too).

For the primary key, it is recommended to use the **DEFAULT** clause, instead of an explicit value.

For our example:

```
INSERT INTO customers
VALUES (
    DEFAULT,
    'John',
    'Smith',
    '1990-01-01' -- or DEFAULT,
    NULL -- or DEFAULT,
    'address',
    'city',
    'CA',
    DEFAULT
    )
```

As said, the DEFAULT keyword can only be used for field that have a default value associated, and the NULL value in field which accept NULL values.

But we can write this statement in another form:

```
INSERT INTO customers (
    first_name,
    last_name,
    birth_date,
    address,
    city,
    state)
-- Those are the ones we will insert,
-- we do not insert the others since
-- they have default values, so we
-- do not have to write them

VALUES (
    'John',
    'Smith',
    '1990-01-01' -- or DEFAULT,
    'address',
    'city',
    'CA'
)
```

And we can insert them in any order we want, as long as we respect the order specified in INSERT INTO.

24 Inserting Multiple Rows

If we want to insert multiple rows at the same time, we just have to separate them with commas and parenthesis:

```
INSERT INTO shippers (name)
VALUES ('Shipper1'),
       ('Shipper2'),
       ('Shipper3')
```

25 Inserting Hierarchical Rows

So far we have just inserted the data into a single table.

But most of the times we will want to insert elements into multiple tables, since different tables will be related.

There exist two types of tables: parent tables and children tables, meaning that children tables may be associated to different parent values.

This is how it is done:

```
INSERT INTO orders (
    customer_id, order_state, status
)
VALUES (1, '2019-01-02', 1);

-- LAST_ITEM_ID is used to get the ID automatically generated
INSERT INTO order_items
VALUES
    (LAST_INSERT_ID(), 1, 1, 2.95),
    (LAST_INSERT_ID(), 1, 1, 2.95)
```

26 Creating a Copy of a Table

We will use the **CREATE TABLE** clause:

```
CREATE TABLE orders_archived AS
SELECT * FROM orders
```

But if we do it this way, the new table will not have a primary key associated (and so it is not marked as a Auto Increment column (AI)). **So, we need to do it by hand**.

If we want to copy just some elements we have to use the WHERE clause:

```
INSERT INTO orders_archived
SELECT *
FROM orders
WHERE order_date < '2019-01-01'
```

27 Updating a Single Row

We do updates the following way:

```
UPDATE invoiced
SET
    payment_total = 10,
    payment_date = '2019-03-01'
WHERE invoice_id = 1
```

28 Updating Multiple Rows

```
UPDATE invoices
SET
    payment_total = invoice_total * 0.5
    pament_date = due_date
WHERE client_id IN (3, 4)
```

29 Using Subqueries in Updates

It may happen that we do not have the id of a client, for instance, we may just have its name, we want to be able to find its id with the name and then update that row:

```
UPDATE invoices
SET
    payment_total = invoice_total * 0.5
    payment_date = due_date
WHERE client_id =
            (SELECT client_id
            FROM clients
            WHERE name = 'Myworks')
```

But, **a single query may return multiple elements, since it does not have to be unique**.
When multiple elements are retrieved, we have to use the IN clause, and **the update will be done to all elements which fulfill the condition**:

```
UPDATE invoices
SET
    payment_total = invoice_totak * 0.5,
    payment_date = due_date
```

```
WHERE client_id IN
            (SELECT client_id
            FROM clients
            WHERE state IN ('CA', 'NY'))
```

(**The best practice is to first just run the code part which retrieves the elements that fulfill the condition, to see if they are the correct ones**)

30 Deleting Rows

We have to use the **DELETE FROM** statement:

```
DELETE FROM invoices
-- if we do not write the WHERE clause all the table will be deleted
WHERE client_id = (
    SELECT *
    FROM clients
    WHERE name = 'Myworks'
)
```