Further SQL Queries

1. Introduction

In this tutorial, we will expand on the basic `SELECT - FROM - WHERE` SQL statements. Specifically, we will learn to

- do a whole lot more with the `WHERE` clause
- Use `CASE - WHEN` to make new columns based on existing columns
- Define “Aliases” to make code more descriptive or more concise
- Use `JOIN` to integrate information across tables

We’ll be using 2 databases today, `rexon_metals.db` and a new database, `weather_stations.db`

- `rexon_metals.db` - 3 tables: CUSTOMER. CUSTOMER_ORDER, PRODUCT
- weather_stations.db - A single table: STATION_DATA

2. More WHERE Clause Use

WHERE on text data

The following should make sense to you (run it anyway, just to confirm).

```
SELECT *
FROM customer
WHERE state == "TX" ;
```

No surprise there. But what if we wanted all the rows in which part of a string variable matched a pattern? To do this, we can use the `LIKE` keyword with a text string containing one or both of the wildcards “%” or “_”. The percent sign is a wildcard that matches any number of characters, and the underscore matches any single character. So

- “”%“” - matches, for example, “hello”, “h”, “World”, or “yada123blah456yada”
- “_” - matches, for example, “a”, “K”, “1” but **not** “apple”, “King”, or “11”

Perhaps the name of the company we want to look at is on the tip of our tongue, but we’re pretty sure it starts with “R”. Select records where the customer name starts with “R”. We could make a query like this

```
SELECT *
FROM customer
WHERE name LIKE "R%";
```

Or perhaps we want to search in a fairly tight geographical region. You could filter on the first three digits of the ZIP code.

```
SELECT *
FROM customer
WHERE CAST(zip AS TEXT) LIKE '750%';
```

Finally, if we wanted all the weather records in which the 2nd letter of the `report_code` was “F”, we could do this.

```
SELECT *
FROM STATION_DATA
WHERE report_code LIKE '_F%';
```

WHERE Clauses with Booleans and AND

We can, of course, do filtering with boolean data, regarding weather conditions for example.

```
SELECT *
FROM station_data
WHERE precipitation > 0 AND fog = 1
```

You're probably thinking something like "Wait, why do we have to test for. Since it's Boolean, can't we just do something like this? "

```
SELECT *
FROM station_data
WHERE precipitation > 0 AND fog
```

Try it! It works on some versions of SQL but not others.

WHERE Clauses using BETWEEN, AND, OR, and IN

These work exactly like we'd expect. So, for example, getting products within a certain price range goes like this

```
SELECT * FROM product
WHERE price BETWEEN 100 AND 200;
```

And grabbing the data from April and May when there was rain or hail.

```
SELECT * FROM station_data
WHERE month IN (4, 5) AND (rain = 1 OR hail = 1);
```

Note the first set of parentheses is necessary for the `IN`, and the second set insures the correct order of operations for the comparisons.

The BYs for WHERE: GROUP_BY and ORDER BY

`GROUP BY`

`GROUP BY` is used when you need to *aggregate* data, like count the number of observations or compute the average within each group specified for `GROUP_BY`. For example, let's group by year and calculate average temperature.

```
SELECT year, AVG(temperature)
FROM station_data
GROUP BY year;
```

In mosT SQL implementations, there are only 5 aggregate functions, MIN, MAX, SUM, COUNT, and AVG. It's a strange set, because AVG is redundant (it could be computed with the sums and the counts), and there is no standard deviation or variance. Ah, well…

ORDER BY

We can sort data by any column in ascending (`ASC`) or decending (`DESC`) order. To sort the station data by temperature in descending order, do

```
SELECT * FROM station_data
ORDER BY temperature DESC;
```

3. CASE

SQL has a `CASE - WHEN - ELSE` construct that is similar to the `match - case` in Python. So we can make a new column to categorize days based on temperature like this

```sql
SELECT station_number, temperature,
CASE
    WHEN temperature > 80 THEN 'Hot'
    WHEN temperature BETWEEN 50 AND 80 THEN 'Mild'
    ELSE 'Cold'
END AS weather_condition
FROM station_data;
```

4. ALIASES

Table Aliases

A table alias is a temporary name assigned to a table in a SQL query. The alias is used for the duration of the query and does not affect the table's name outside of that query.

Why Use Table Aliases?

1. **Simplification**: When table names are long or complex, aliases can simplify their references in the query.
2. **Clarification**: In queries involving multiple tables, aliases help clarify which columns are coming from which tables.
3. **Necessity**: In self-joins, where a table joins to itself, aliases are necessary to distinguish the different instances of the table.

Syntax

The syntax to define a table alias is straightforward. After stating the table name, you place the keyword `AS` followed by the alias name. The `AS` keyword is optional but can make the query clearer. Here's the general syntax:

```sql
SELECT column1, column2
FROM table_name AS alias_name
WHERE alias_name.column1 = condition;
```

Note the dot syntax to reference a column using the alias.

Using Column Aliases

You can also create aliases for columns to either shorten their names or to give more descriptive names to columns in the result set.

Syntax

The syntax for column aliases follows a similar pattern. After the column name, you place the `AS` keyword followed by the alias name for the column. Here's how you might do it:

```
1  SELECT column_name AS alias_name FROM table_name;
```

Example

Assign aliases to some columns in the `rexon_metals.db` to make the view more descriptive:

```
1  SELECT description AS product_material,
2  price AS product_price
3  FROM product;
```

In this example, `product_material` and `product_price` are aliases for the `description` and `price` columns from the `product` table, making the output more understandable.

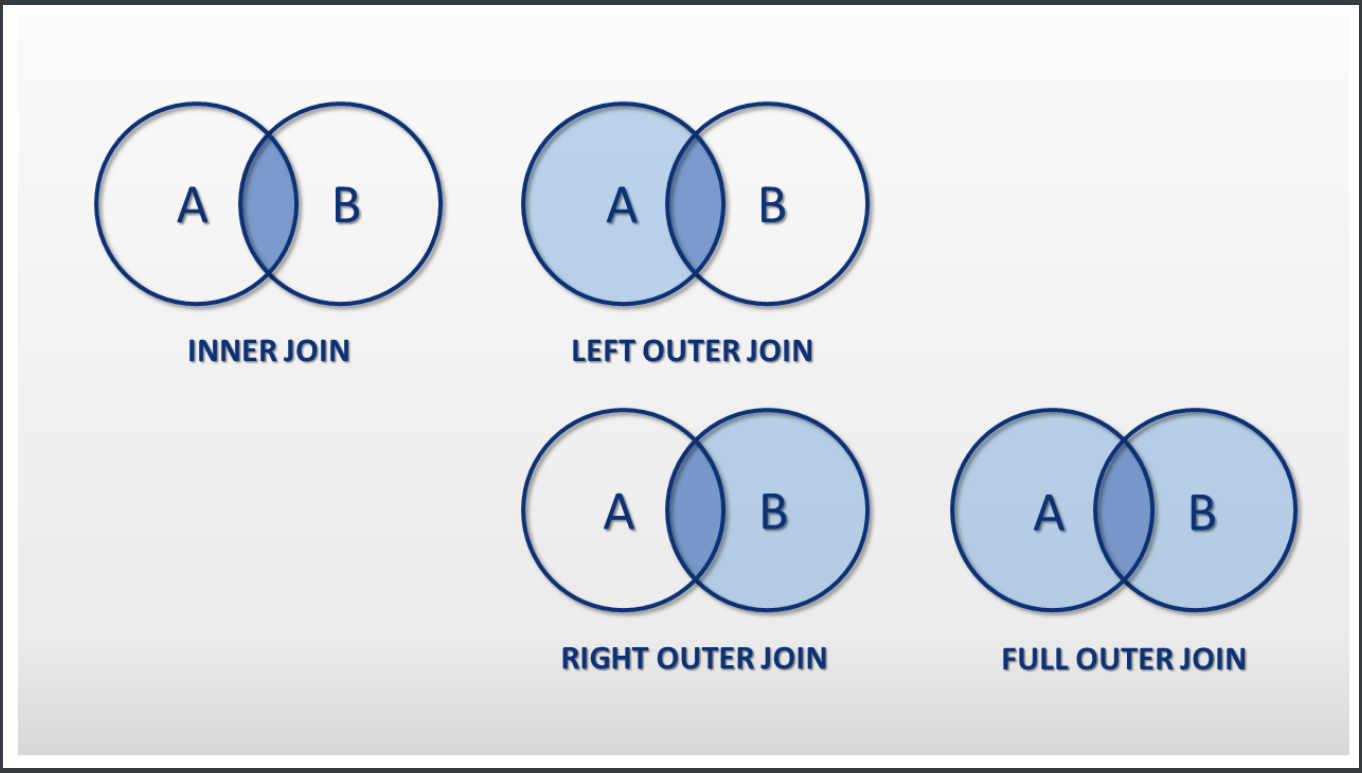
5. JOINs

Joining information from multiple tables is a fundamental aspect of SQL. After all, why have a relational data base with multiple logically-organized tables if there is no way to integrate the data across tables. There are four basic join types, but we mostly use just two of them, the "inner join" and "left join".

Explanation of JOIN Types

- **INNER JOIN**: Returns rows when there is a match in both tables.
- **LEFT JOIN**: Returns all rows from the left table, and the matched rows from the right table.
- **RIGHT JOIN** and **FULL JOIN**: These types of joins are not supported directly by SQLite but can be emulated using specific queries.

Here is a diagram showing the four types of joins. A represents the left table and B represents the right table.



Most often, we want to do an inner join; in SQL, you can specify this with `INNER JOIN`, but this is also the default if you just use `JOIN`.

INNER JOIN

Let's say we want to look at just the customers who have actually ordered something. To do this, we "join" the tables and get the names from the `customer` table who's `customer_id`'s appear in the `customer_order` table:

```
SELECT c.name
FROM customer_order AS co
INNER JOIN customer AS c ON co.customer_id = c.customer_id
```

Note the use of aliases so shorten the actual comparison.

Since `JOIN` alone defaults to `INNER JOIN` and the `AS`'s are optional in this context, this also works:

```
SELECT c.name
FROM customer_order co
JOIN customer c ON co.customer_id = c.customer_id
```

Now let's add another join to get some product information for the companies that actually ordered something. To find all orders along with their corresponding product descriptions and customer information, we could do this:

```
SELECT c.name, p.description, p.price
FROM customer_order AS co
INNER JOIN customer AS c ON co.customer_id = c.customer_id
INNER JOIN product p ON co.product_id = p.product_id;
```

LEFT JOIN

List all customers and their orders, *including those who have not placed any orders*, we can use a left join:

```
SELECT c.name, p.description, p.price
FROM customer c
LEFT JOIN customer_order co ON c.customer_id = co.customer_id
LEFT JOIN product p ON co.product_id = p.product_id;
```

This will produce a view with “NULL” values for the description and price if a company has yet to order anything.

Aggregating Data with JOINs

We can `GROUP_BY` in order to compute aggregations after the join. We can also use `ORDER BY` to sort the output. Here, we compute the total amount of money spent by each of our customers, and sort the output from highest to lowest, so we can see who our big spenders are!

```
SELECT c.name, SUM(p.price * co.ORDER_QTY) AS total_spent
FROM customer c
JOIN customer_order co ON c.customer_id = co.customer_id
JOIN product p ON co.product_id = p.product_id
GROUP BY c.name
ORDER BY total_spent DESC;
```

So Marsh Lane Metal Works is our biggest customer by money spent. They only ordered one time, but it was a fair amount of steel, one of the more expensive products.

Using Aliases with JOIN

Aliases are completely optional. They can make columns in a view more descriptive, as we saw above, or you can use them to make the SELECT clause and the comparisons following a `JOIN` a bit shorter.

Consider a query on the `rexon_metals.db` database where we need to join the `customer`, `product`, and `customer_order` tables. Without aliases, it looks like this:

```
SELECT customer.customer_name, product.description, product.price
FROM customer_order
JOIN customer ON customer_order.customer_id = customer.customer_id
JOIN product ON customer_order.product_id = product.product_id;
```

Here's the same query using aliases to make it more concise:

```
SELECT c.customer_name, p.description, p.price
FROM customer_order co
JOIN customer c ON co.customer_id = c.customer_id
JOIN product p ON co.product_id = p.product_id;
```

In this example, `co` is an alias for `customer_order`, `c` for `customer`, and `p` for `product`. These are used to reference the respective tables in the `JOIN` conditions and when selecting columns, to make everything more compact.

Whether you use aliases or not is totally up to you (or your boss or team leader or whatever…).

6. Exercises

A. Make a View that has columns 1) station_number, 2) days_with_precipitation (which will be an aggregation of the `GROUP BY` following a `WHERE` filter), and 3) a station_type indicating whether the station was a "Wet Station" or a "Dry Station" depending whether the station got more than 6 days of rain or not. Order the View by days_with_precipitation.Hint: Since the weather station data is a single table, we won't need to use a join, but we will to use aggregations and a `CASE - WHEN - ELSE`.Hint: Since the weather station data is a single table, we won't need to use a join, but we will to use aggregations and a `CASE - WHEN - ELSE`.

B. Use JOIN to create a view from the rexon_metals.db showing customer name (customer.name) and price paid (customer_order.order_qty * product.price) for each order.

> Turn in a pdf with screenshots of SQLiteStudio showing your query pane and the results pane for the above two exercises.

---

7. Summary

7. Summary

We have learned quit a bit in this tutorial. Specifically, we have learned to

- do more with `WHERE` filtering
- make new informative or summary columns with `CASE`
- use aliases for tables or columns if desired
- `JOIN` columns from different tables, therefore integrating information across the database tables

With that, we actually have a fairly complete albeit basic SQL mental toolbox! The rest is just practice practice practice if you need to access a SQL database.