Chapter 2 : Intermediate SQL

Aviral Janveja

1 Aggregate Functions

Chapter 1 on Basic SQL pointed out that arithmetic operators only perform operations across rows. Aggregate functions are used to perform operations across entire columns, which could include millions of rows of data or more. We will cover each one of them individually.

1.1 COUNT

COUNT is a SQL aggregate function for counting the number of rows in a particular column. For example, the following gives the count of all the rows in the dataset :

```
SELECT COUNT(*)
    FROM tutorial.aapl_historical_stock_price
```

The following code will provide the number of rows in the column named "high", where values are not null.

```
SELECT COUNT(high)
    FROM tutorial.aapl_historical_stock_price
```

You can also use it on non-numerical columns :

```
SELECT COUNT(date)
    FROM tutorial.aapl_historical_stock_price
```

Notice that the column header in the results just reads "count". We recommend naming your columns so that they make a little more sense to anyone else who views your work. As mentioned in an earlier lesson, it's best to use lower case letters and underscores. You can add column names using AS :

```
SELECT COUNT(date) AS count_of_date
    FROM tutorial.aapl_historical_stock_price
```

If you must use spaces, you will need to use double quotes.

```
SELECT COUNT(date) AS "Count Of Date"
    FROM tutorial.aapl_historical_stock_price
```

Note that this is really the only place where you will use double quotes in SQL. Single quotes for everything else.

1.2 SUM

SUM is a SQL aggregate function that totals the values in a given column. Unlike COUNT, you can only use SUM on columns containing numerical values. The query below selects the sum of the volume column from the Apple stock prices dataset :

```
SELECT SUM(volume)
    FROM tutorial.aapl_historical_stock_price
```

An important thing to remember : aggregators only aggregate vertically. If you want to perform a calculation across rows, you would do this with simple arithmetic functions. You don't need to worry as much about the presence of nulls with SUM as you would with COUNT, as SUM treats nulls as 0.

1.3 MIN / MAX

MIN and MAX are SQL aggregation functions that return the lowest and highest values in a particular column. Depending on the column type, MIN will return the lowest number, earliest date or non-numerical value as close alphabetically to "A" as possible. As you might suspect, MAX does the opposite, it returns the highest number, the latest date or the non-numerical value closest alphabetically to "Z".

For example, the following query selects the MIN and the MAX from the "volume" column in the Apple stock prices dataset :

```
SELECT MIN(volume) AS min_volume,
       MAX(volume) AS max_volume
    FROM tutorial.aapl_historical_stock_price
```

1.4 AVG

AVG is a SQL aggregate function that calculates the average of a selected group of values. It is very useful, but has some limitations. First, it can only be used on numerical columns. Second, it ignores nulls completely. You can see this by comparing these two queries of the Apple stock prices dataset :

```
SELECT AVG(high)
    FROM tutorial.aapl_historical_stock_price
    WHERE high IS NOT NULL
```

The above query produces the same result as the following query :

```
SELECT AVG(high)
    FROM tutorial.aapl_historical_stock_price
```

2 GROUP BY

SQL aggregate function like COUNT, AVG, and SUM have something in common, they all aggregate across the entire table. But what if you want to aggregate only

part of the table? For example, you might want to count the number of entries for each month individually.

In such situations, you need to use the GROUP BY clause. GROUP BY allows you to separate data into groups, which can be aggregated independently of one another. Here is an example :

```
SELECT year, COUNT(*) AS count
    FROM tutorial.aapl_historical_stock_price
    GROUP BY year
```

You can group by **multiple columns**, as follows :

```
SELECT year, month, COUNT(*) AS count
    FROM tutorial.aapl_historical_stock_price
    GROUP BY year, month
```

If you want to **order** the aggregations by month or year, use ORDER BY alongside GROUP BY :

```
SELECT year, month, COUNT(*) AS count
    FROM tutorial.aapl_historical_stock_price
    GROUP BY year, month
    ORDER BY year, month
```

3 HAVING

In the previous lesson, we learned how to use the GROUP BY clause to aggregate stats by month and year. However, you will often encounter datasets where GROUP BY isn't enough to get what you are looking for.

Let us say that, it is not enough, just to know the aggregated stats by month. After all, there are a lot of months in this dataset. Instead, you might want to find all the months where stock price went over 400. The WHERE clause won't work here because it doesn't allow you to **filter on aggregate columns**. That is where the HAVING clause comes in :

```
SELECT year, month, MAX(high) AS month_high
    FROM tutorial.aapl_historical_stock_price
    GROUP BY year, month
    HAVING MAX(high) > 400
    ORDER BY year, month
```

3.1 Query Order

As mentioned in prior lessons, the order in which you write the clauses is important. Here is the order for everything you have learned so far :

SELECT $\rightarrow$ FROM $\rightarrow$ WHERE $\rightarrow$
GROUP BY $\rightarrow$ HAVING $\rightarrow$ ORDER BY

4 CASE

The CASE statement is SQL's way of handling **conditional** logic. The CASE statement is followed by at least one pair of WHEN and THEN statements and must end with the END statement. The ELSE statement is optional and provides a way to capture values not specified in the WHEN-THEN statements. CASE is easiest to understand in the context of an example :

```
SELECT player_name, year,
    CASE
        WHEN year = 'SR' THEN 'yes'
        ELSE 'no'
    END AS is_a_senior
    FROM benn.college_football_players
```

The WHEN-THEN conditions are evaluated in order. Once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause. If there is no ELSE part and no conditions are true, it returns NULL.

The CASE statement always goes in the SELECT clause. You can include **multiple** WHEN-THEN statements as well as an ELSE statement to deal with any unaddressed conditions. You can also string together multiple conditional statements using AND-OR the same way you might in a WHERE clause.

```
SELECT player_name, weight,
    CASE
        WHEN weight > 250 THEN 'Super_Heavyweight'
        WHEN weight > 200 AND weight <= 250 THEN
        'Heavyweight'
        WHEN weight > 175 AND weight <= 200 THEN
        'Middleweight'
        ELSE 'Lightweight'
    END AS weight_class
    FROM benn.college_football_players
```

4.1 Using CASE with Aggregate Functions

Let us say you want to only **count** rows that fulfill a certain **condition**. Here is an example of counting multiple conditions in one query :

```
SELECT CASE
    WHEN year = 'FR' OR year = 'SO' THEN 'First Two'
    WHEN year = 'JR' OR year = 'SR' THEN 'Last Two'
    ELSE 'No Year Data'
END AS year_group,
COUNT(*) AS count
FROM benn.college_football_players
GROUP BY year_group
```

Combining CASE statements with aggregations can be tricky at first. It is often helpful to write a query containing the CASE statement first and run it on its

own. From there, you can add the aggregation function and the GROUP BY clause, producing the following result :

	year_group	count
1	First Two	15546
2	Last Two	10752

Figure 1: CASE with COUNT

4.2 Using CASE inside Aggregate Functions

In the previous example, data was displayed vertically, but in some instances, you might want to show data **horizontally**. This is known as **pivoting**. Let us take the above query and re-orient it horizontally :

```
SELECT
    COUNT(CASE WHEN year = 'FR' OR year = 'SO'
    THEN 1 ELSE NULL END) AS "First Two",
    COUNT(CASE WHEN year = 'JR' OR year = 'SR'
    THEN 1 ELSE NULL END) AS "Last Two"
    FROM benn.college_football_players
```

The output now looks as follows :

	First Two	Last Two
1	15546	10752

Figure 2: Horizontal Pivot

5 DISTINCT

You will occasionally want to look at only the unique values in a particular column. You can do this using SELECT DISTINCT syntax.

```
SELECT DISTINCT month
    FROM tutorial.aapl_historical_stock_price
```

If you include two or more columns in a SELECT DISTINCT clause, your results will contain all of the unique pairs of those two columns :

```
SELECT DISTINCT year, month
    FROM tutorial.aapl_historical_stock_price
```

Note : You only need to include DISTINCT once in your SELECT clause—you do not need to add it for each column name.

5.1 Using DISTINCT in Aggregations

You can use DISTINCT when performing an aggregation. You will use it most commonly with the COUNT function.

```
SELECT COUNT(DISTINCT month) AS unique_months
    FROM tutorial.aapl_historical_stock_price
```

Notice that DISTINCT goes inside the aggregate function rather than at the beginning of the SELECT clause. It is also worth **noting** that using DISTINCT, particularly in aggregations, can **slowdown** your queries quite a bit. Performance of SQL queries will be discussed in chapter 3.

6 Joins

Up to this point, we have only been working with one table at a time. The real power of SQL however, comes from working with data from multiple tables at once. Join is a way to combine data from two or more tables based on a related column between them.

We have been working with the table on college football players. This table includes data on players, including each player's weight and the school that they play for. However, it does not contain much information about the school, such as the conference the school is in. That information is present in a separate table on college football teams.

So, let us say we want to figure out which conference has the highest average weight. Since, this information is spread across two separate tables, let us construct the required join query step by step.

6.1 Aliases

First things first, when performing joins, typing in the long table names every time is pretty annoying, instead a shorter alias is much better. You can assign an alias by adding a space after the table name and typing in the intended alias, as shown below :

```
SELECT AVG(players.weight) AS average_weight
   FROM benn.college_football_players players
```

Once you have assigned an alias, you can refer to columns from that table in the SELECT clause using the alias name. As shown above, the weight column is selected as "players.weight".

6.2 JOIN and ON

After the FROM statement, we have two new statements, JOIN which is followed by a table name and ON, which is followed by column names separated by an equals sign.

```
SELECT teams.conference AS conference,
AVG(players.weight) AS average_weight
    FROM benn.college_football_players players
    JOIN benn.college_football_teams teams
    ON teams.school_name = players.school_name
```

ON indicates how the two tables, the one after the FROM and the one after the JOIN, relate to each other. You can see in the example above that both tables contain the column called **school_name**.

During the join, SQL looks up the school_name column in the teams and players tables. If there is a match in row values, SQL takes all five columns from the teams table and joins them onto the ten columns of players table. The new result is a combined fifteen column table.

Once you have generated this new table using the join, you can use the aggregate function AVG on player weights and group by the conference column from the teams table.

```
SELECT teams.conference AS conference,
AVG(players.weight) AS average_weight
    FROM benn.college_football_players players
    JOIN benn.college_football_teams teams
    ON teams.school_name = players.school_name
    GROUP BY teams.conference
    ORDER BY AVG(players.weight) DESC
```

Hence, providing the average weight for each conference and solving the problem statement we started with.

7 Inner Join

Inner joins eliminate rows from both tables that do not satisfy the join condition set forth in the ON statement. In mathematical terms, an inner join is the intersection (AND) of the two tables.

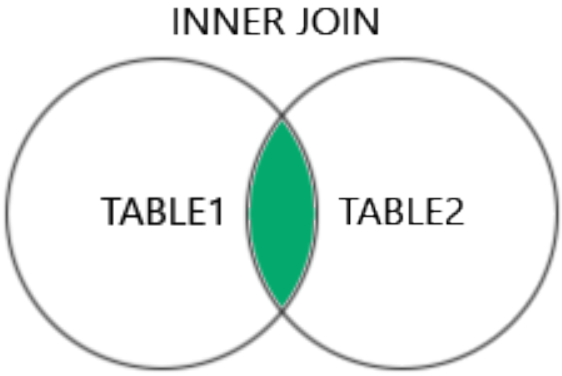

Figure 3: Inner Join

Therefore, if a player goes to a school that isn't in the teams table, that player won't be included in the result from an inner join. Similarly, if there are schools in the teams table that don't match to any schools in the players table, those rows won't be included in the results either.

7.1 Identical Column Names

When you join two tables, it might be the case that both tables have columns with identical names. In the below example, both tables have columns called school_name :

```
SELECT players.*, teams.*
    FROM benn.college_football_players players
    JOIN benn.college_football_teams teams
    ON teams.school_name = players.school_name
```

The results can only support one column with a given name. When you include two columns of the same name, the results will simply show the exact same result set for both columns even if the two columns should contain different data. You can avoid this by naming the columns individually. It so happens, that these two columns will actually contain the same data because they are used for the **join key** (The column used to join the two tables), but the following query technically allows these columns to be independent :

```
SELECT players.school_name AS players_school_name,
teams.school_name AS teams_school_name
    FROM benn.college_football_players players
    JOIN benn.college_football_teams teams
    ON teams.school_name = players.school_name
```

8 Outer Joins

When performing an inner join, only rows from either table that are matched in the other table are returned. In an outer join, In addition to the matched rows, unmatched rows from one or both the tables can also be returned.

Going forward, we will work with data from Crunchbase on that holds information about companies, startups, founders, investors and related stuff. There are two tables, one with data on the companies and second with data on the acquisition details of the companies.

There is a **permalink** column in the **companies** table, which is the unique identifier for each row. This column maps to the **company_permalink** and **acquirer_permalink** columns in the **acquisitions** table. The foreign key you use to join these two tables will depend entirely on whether you are looking to add information about the acquiring company or the company that was acquired.

9 Left Join