Chapter 2 : Aggregation

Aviral Janveja

Aggregate functions perform calculations on a set of values and return a single value as result. For example, you can find the sum total of all values in a column using an aggregate function. We cover them one by one in the following sections.

1 COUNT

This is an aggregate function for counting the number of rows in a particular column. For example, 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". It is thereby recommended to name your columns so that they make a little more sense. As mentioned in an earlier lesson, it is best to use lower case letters and underscores :

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

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

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

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

2 SUM

This is an aggregate function that returns the sum of values in a numerical column. The following query selects 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. To perform calculations across rows, use arithmetic functions as described in chapter one. 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.

3 MIN

This is an aggregate function that returns the smallest value in a particular column. Depending on the column type, MIN will return the smallest number, the earliest date or the first value alphabetically. For example, the following query selects the MIN value from the "volume" column in the Apple stock prices dataset :

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

4 MAX

This is an aggregate function that returns the largest value in a particular column. Depending on the column type, MAX will return the largest number, the latest date or the last value alphabetically. For example, the following query selects the MAX value from the "volume" column in the Apple stock prices dataset :

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

5 AVG

This is an aggregate function that calculates the average of values in a numerical column. It ignores nulls completely. For example :

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

6 GROUP BY

GROUP BY enables us to separate data into groups, which can then be aggregated independently. For example :

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

The above query gives us the aggregate count, grouped year wise. You can also 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 along with GROUP BY :

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

7 HAVING

The HAVING clause enables us to filter on aggregate columns, something which the WHERE clause cannot accomplish. For example :

```
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
```

The above query groups and aggregates the data by year and month, while also filtering for months where high was greater than 400.

7.1 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 :

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

7.2 Internal Execution Order

The order of execution of a query is different from the way it is written. Here is the logical order in which SQL processes the query internally :

1. FROM table : Load the data.
2. GROUP BY year, month : Group the data into buckets for each year-month pair.
3. AGGREGATE MAX(high) : For each group, find the maximum high price.
4. HAVING MAX(high) $>$ 400 : Filter out groups where the maximum high is $\leq$ 400.
5. SELECT year, month, MAX(high) AS month_high : Project the required columns.
6. ORDER BY year, month : Sort the final result.

8 CASE

The CASE statement is SQL's way of handling **if-else** 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, Case will stop reading and return the result. If no conditions are true, it will return the value in the ELSE clause. If there is no ELSE part and no conditions are true, it returns NULL.

> **Note** : The CASE statement always goes in the SELECT clause, just like the aggregate functions.

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 as you would 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
```

8.1 CASE with Aggregate Functions

Let us say that you only want to **count** those rows that fulfill a certain **condition**. Here is an example :

```
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 aggregate functions 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 aggregate function and the GROUP BY clause, hence producing the following result :

	year_group	count
1	First Two	15546
2	Last Two	10752

Figure 1: CASE with COUNT

8.2 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

9 DISTINCT

You will occasionally want to look at only unique values in a particular column. You can do this using the 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 don't need to add it for each column name.

9.1 DISTINCT inside Aggregate Functions

You can use DISTINCT when performing an aggregation as well. 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 with aggregate functions, can slowdown your queries quite a bit.