Chapter 2 : Intermediate SQL

Aviral Janveja

1 Aggregate Functions

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.

1.1 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". Hence, it is 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. 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

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. 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

This is an aggregation 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 non-numerical value as close alphabetically to "A" as possible.

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
```

1.4 MAX

This is an aggregation 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 non-numerical value as close alphabetically to "Z" as possible.

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
```

1.5 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
```

2 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
```

3 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.

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 :

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

```
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

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 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.

5.1 Using 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. Performance optimization of SQL queries will be discussed in chapter 3.

6 Joins

Up to this point, we have been working with only 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 :

```
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.

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.

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

During the join, SQL looks up each row of 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 for that particular row, resulting in a combined fifteen column table.

Once you have generated this new table using 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 join eliminates 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.

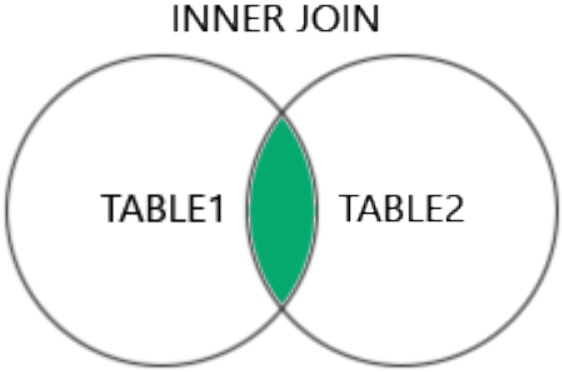

Figure 3: Inner Join

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.

> **Note** : JOIN and INNER JOIN will return the same result. INNER is the default type of JOIN.

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, depending on the type of outer join.

Going forward, we will work with data from Crunchbase that holds information about companies, founders, investors and so on. 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.

8.1 LEFT JOIN

LEFT JOIN tells the database to return all rows from the table in the FROM clause (Table 1 on the left) and only matching rows from the table in the LEFT JOIN clause (Table 2 on the right).

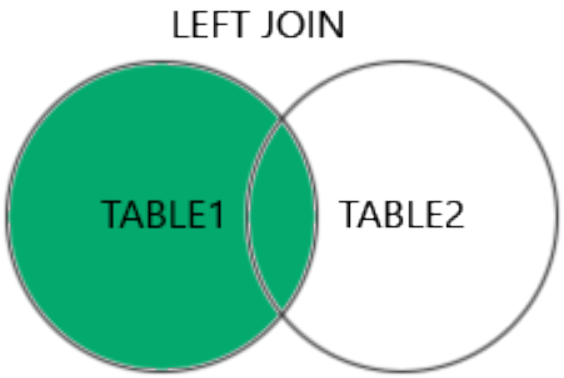

Figure 4: Left Join

Note that you can use the WHERE clause to filter after the join and the ON clause to filter one or both of the tables before joining them. For example :

```
SELECT companies.state_code,
    COUNT(DISTINCT companies.permalink) AS
    unique_companies,
    COUNT(DISTINCT acquisitions.company_permalink) AS
    unique_companies_acquired
    FROM tutorial.crunchbase_companies companies
    LEFT JOIN tutorial.crunchbase_acquisitions acquisitions
    ON companies.permalink = acquisitions.company_permalink
    WHERE companies.state_code IS NOT NULL
    GROUP BY 1
    ORDER BY 3 DESC
```

8.2 RIGHT JOIN

RIGHT JOIN is similar to left join except that they return all rows from the table in the RIGHT JOIN clause (Table 2 on the right) and only matching rows from the table in the FROM clause (Table 1 on the left).

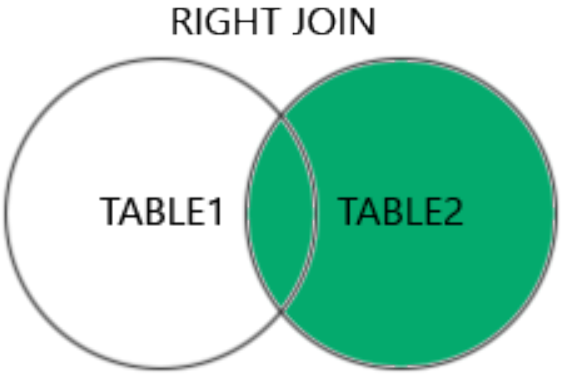

Figure 5: Right Join

RIGHT JOIN is rarely used because you can achieve the same result by simply switching the table names in a LEFT JOIN. It is worth noting that LEFT JOIN and RIGHT JOIN can be written as LEFT OUTER JOIN and RIGHT OUTER JOIN as well.

8.3 FULL JOIN

LEFT JOIN and RIGHT JOIN each return unmatched rows from one of the tables. FULL JOIN returns unmatched rows from both the tables. It is commonly used in conjunction with aggregate functions to understand the amount of overlap between two given tables.

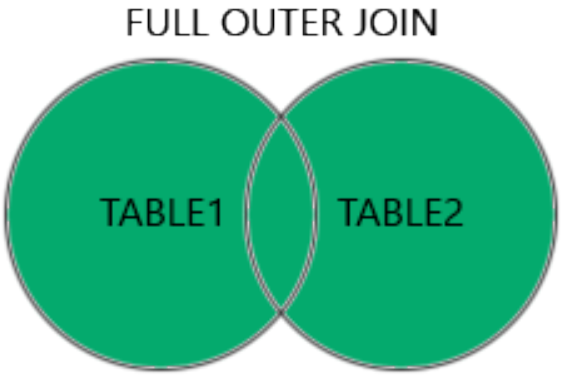

Figure 6: Full Join

9 Self Join

A self join is a regular join, but the table is joined with itself. Sometimes it could be useful to join a table onto itself.

Here is an example to illustrate. Consider a table named **Employees_Table** with columns **employee_id**, **employee_name** and **manager_id**. The manage_id column contains the employee_id of the manager. Now, suppose you want to retrieve the names of employees along with the names of their respective managers.

Each employee in the company, which includes the managers, is assigned an employee_id. Hence, using the manager_id column, we can identify their names as follows :

```
SELECT e.employee_name AS Employee_Name,
m.employee_name AS Manager_Name
    FROM Employees e
    JOIN Employees m
    ON e.manager_id = m.employee_id
```

Note how the same table can be referenced multiple times using different aliases. In this case, **e** and **m**.

In the above query, **e.manager_id = m.employee_id** specifies the condition for the join, where we match each manager with their respective employee_id, resulting in a table where each row contains the name of the employee alongside the name of their manager.

10 UNION

While JOIN allows you to combine two datasets side-by-side, UNION allows you to stack one dataset on top of the other. Put differently, UNION allows you to write two separate SELECT statements and to have the results of both statements display in the same table. For example, the following query will display all results from the first and the second portion of the query in the same table :

```
SELECT *
    FROM tutorial.crunchbase_investments_part1

UNION

SELECT *
    FROM tutorial.crunchbase_investments_part2
```

Note that UNION only appends distinct values. More specifically, when you use UNION, the dataset is appended and any rows in the appended table that are identical to rows in the first table are dropped. If you would like to append all the values from the second table, use **UNION ALL** :

```
SELECT *
    FROM tutorial.crunchbase_investments_part1

UNION ALL

SELECT *
    FROM tutorial.crunchbase_investments_part2
```

SQL has some strict rules for appending data :

1. Both tables must have the same number of columns.
2. The columns must have the same data types and in the same order.

While the column names don't necessarily have to be the same, you will find that they typically are. This is because most of the instances in which you would want to use UNION involve stitching together different parts of the same dataset, as in the example shown above.

Since you are writing two separate SELECT statements, you can treat them differently before appending. For example, you can filter them differently using different WHERE clauses.

11 Joining on Multiple Keys

There are two major reasons you might want to join tables on multiple keys. The first has to do with accuracy. The second reason has to do with performance. This will be covered in greater detail in the next chapter.

For now, all you need to know is that it can occasionally make your query run faster, even when it does not add to the accuracy of the query. For example, the results of the following query will be the same with or without the last line. However, the query runs more quickly with the last line included :

```
SELECT companies.permalink,
       companies.name,
       investments.company_name,
       investments.company_permalink
    FROM tutorial.crunchbase_companies companies
    LEFT JOIN tutorial.crunchbase_investments_part1
    investments
    ON companies.permalink = investments.company_permalink
    AND companies.name = investments.company_name
```

It is worth noting that this will have relatively little effect for small datasets.