Chapter 3 : Joins

Aviral Janveja

1 Introduction

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 multiple tables in a database. Join is the way to combine data from two or more tables based on a common 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 itself, such as the conference the school is in. That information is present in a separate table on college football teams.

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

1.1 Table Names

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.

1.2 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 with 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 from the example above that both these tables contain a common 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.

2 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 two tables.

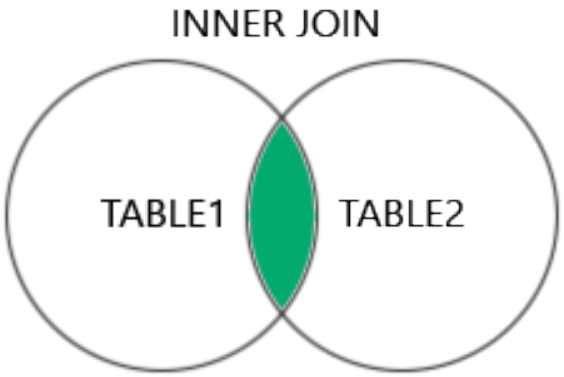

Figure 1: Inner Join

Therefore, if a player from the players table goes to a school that is not included in the teams table, that player will not be included in the result. Similarly, if there are schools in the teams table that do not match with schools in the players table, those rows will not be included in the result either.

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

3 Outer Joins

When performing an inner join, only matching rows from both tables are returned. In an outer join, In addition to the matched rows, unmatched rows from one or both of the tables are also 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.

3.1 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).

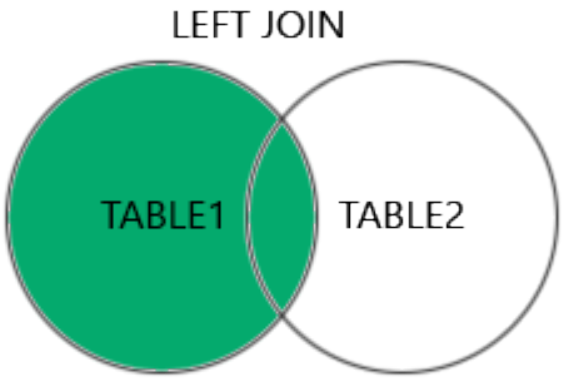

Figure 2: Left Join

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
```

3.2 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).

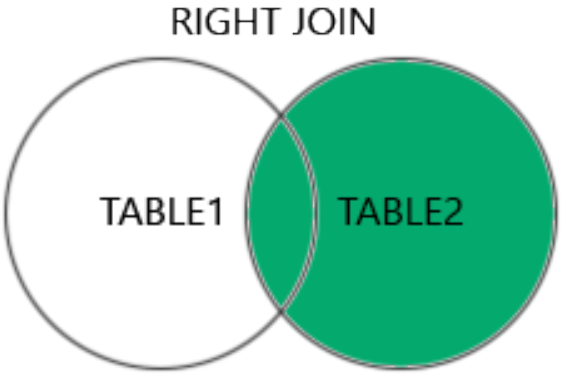

Figure 3: Right Join

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.

3.3 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.

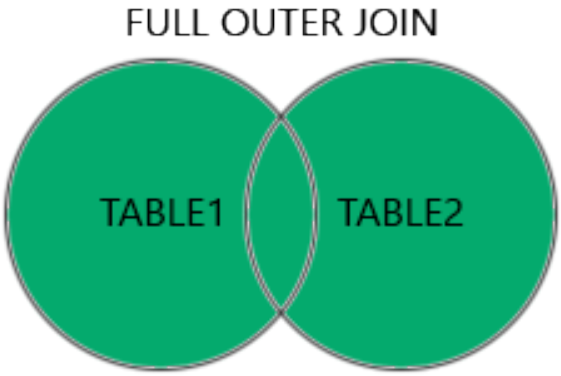

Figure 4: Full Join

4 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.

5 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
```

5.1 UNION ALL

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.

6 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.