Introduction to SQL Using Python: Using JOIN Statements to Merge Multiple Tables

E

Erika D Follow

Nov 10, 2019 · 10 min read

This blog is a tutorial on how to pull data from multiple tables in SQL. In my previous blog tutorials, data was only queried from a single table at a time. This tutorial will introduce **JOIN** statements. With **JOIN** statements, data from multiple tables can be returned in a single query.

This blog will discuss the following:

- **LIMIT Statement**
- **Returning Multiple Tables without a JOIN Statement**
- **JOIN Statement**
- **Return Specific Columns with a JOIN Statement**
- **Filter & Aggregate with Multiple Tables**
- **INNER JOIN**
- **LEFT JOIN**
- **RIGHT JOIN**
- **FULL OUTER JOIN**
- UNION Statement

We will be using the database used in my previous tutorials which can be downloaded here. If you have not seen my previous tutorials, now would be a good time to preview the data tables available in the Football Delphi database. You can read my first two SQL tutorials, at the below links:

- Introduction to SQL Using Python: Filtering Data with the WHERE Statement
- Introduction to SQL Using Python: Computing Statistics & Aggregating Data

To begin, we will download the necessary libraries, **sqlite3** and **pandas**.

```
# Import necessary libraries

import sqlite3
import pandas as pd
```

Import necessary libraries

Next, you will need to connect to the database and create a cursor object.

```
# Connect to database
conn = sqlite3.connect('''database.sqlite''')

# Create cursor object
cur = conn.cursor()
```

Connect to database & create cursor object

The following is format we will be using to run our SQL queries in Python.

Format for SQL queries in Python

LIMIT Statement

The **LIMIT** statement will limit the number of rows returned in a SQL query. It is written at the end of a SQL query and the last statement to be executed in the SQL order of operations. To see how the **LIMIT** statement is used, preview the query below:


```
SELECT * FROM Teams_in_Matches LIMIT 10
```

The above query returns the first ten rows from the **Teams_in_Matches** table. To practice using the **LIMIT** statement, write a query that returns the first ten rows from the **Unique_Teams** dataset and compare your query to the one below:

```
SELECT * FROM Unique_Teams LIMIT 10
```

Returning Multiple Tables Without a JOIN Statement

The **Teams_in_Matches** dataset has two columns, **Match_ID** and **Unique_Team_ID**. Since we are not familiar with the **Unique_Team_ID** for each team, it would be easier to identify each team if we could see the name of the team along side the columns in the **Team_in_Matches** data table. The **Unique_Teams** data table has two columns, **Team_Name** and **Unique_Team_ID**. If we returned both tables together we could see the name of each team next to the **Match_ID** and **Unique_Team_ID** columns from the **Teams_in_Matches** dataset. The below query returns the data from both tables:

SELECT * FROM Teams_in_Matches, Unique_Teams

The query returned 6,290,944 rows although the **Teams_in_Matches** table only has 49,148 rows and the **Unique_Teams** table only has 128 rows. You may have thought that the query would return at most 49,148 rows. What happened was that SQL took every single row from the **Teams_in_Matches** table and matched it with every single row from the **Unique_Teams** table, so the query returned 49,148 * 128 = 6,290,944 rows. The returned data we have now is not useful and way too large. If we want SQL to return the **Match_ID** and **Unique_Team_ID** with the corresponding **Team_Name** we have to tell SQL how to return the results. Since **Unique_Team_ID** is a column name in both tables we can use the query below to get the correct results:

SELECT * FROM Teams_in_Matches, Unique_Teams WHERE Teams_in_Matches.Unique_Team_ID = Unique_Teams.Unique_Team_ID

We added a **WHERE** statement and told SQL **how/what column** to join together the two tables, **WHERE Teams_in_Matches.Unique_Team_ID = Unique_Teams.Unique_Team_ID**. The returned results show only the rows where the **Unique_Team_ID** from the **Teams_in_Matches** table matched the value of the **Unique_Team_ID** from the **Unique_Teams** data table. When working with multiple tables you have to clarify what table each column is coming from, especially when the column names have the same name. In the **WHERE** statement we wrote the name of the table followed by a period and then the column name, i.e. **Teams_in_Matches.Unique_Team_ID**.

JOIN Statement

We will run the same query we just used but instead of using a **WHERE** statement, we will use a **JOIN** statement. To see how this looks, look at the query below:

SELECT * FROM Teams_in_Matches JOIN Unique_Teams ON Teams_in_Matches.Unique_Team_ID = Unique_Teams.Unique_Team_ID

Instead of listing the tables we want to use in the **FROM** statement we first list the table, **Teams_in_Matches** and then added a **JOIN** statement. We are telling SQL that we want the data from the **Teams_in_Matches** table joined together with the data from the **Unique_Teams**. We also included an **ON** statement, this tells SQL HOW to join the data together.

The **Teams** table does not include the **Unique_Team_ID** for each team. Practice using the **JOIN** statement by writing a query that joins together the **Unique_Teams** data table and the **Teams** table, only return the first 10 rows. Compare your query to the one below:

```
SELECT * FROM Unique_Teams JOIN Teams ON Unique_Teams.TeamName = Teams.TeamName LIMIT 10;
```

Return Specific Columns with a JOIN Statement

If we go back and look at our query where we joined together the **Teams_in_Matches** table and the **Unique_Teams** table, you will see the the **Unique_Team_ID** column is listed twice. By specifying the columns we want returned in the **SELECT** statement, we can get rid of this redundancy.

```
SELECT Teams_in_Matches.Match_ID, Teams_in_Matches.Unique_Team_ID, Unique_Teams.TeamName
FROM Teams_in_Matches JOIN Unique_Teams ON Teams_in_Matches.Unique_Team_ID =
Unique_Teams.Unique_Team_ID LIMIT 5
```

Again, because we are querying data from multiple tables, we specify what table each column is coming from by writing the table name followed by a period and then the

column name (**Unique_Teams.TeamName**). Practice selecting specific columns when working with multiple tables by writing a query that shows the **Unique_Team_ID** and **TeamName** from the **Unique_Teams** table and **AvgAgeHome**, **Season** and **ForeignPlayersHome** from the **Teams** table. Only return the first five rows. Compare your query to the one below:

```
SELECT Unique_Teams.Unique_Team_ID, Unique_Teams.TeamName, Teams.AvgAgeHome, Teams.Season,
Teams.ForeignPlayersHome FROM Unique_Teams JOIN Teams ON Unique_Teams.TeamName =
Teams.TeamName LIMIT 5
```

Although it is good practice to write the table name before the column name when querying multiple tables, it is only necessary to do so when there are duplicate column names. The following query will produce the same results as the query above:

```
SELECT Unique_Team_ID, Unique_Teams.TeamName, AvgAgeHome, Season, ForeignPlayersHome FROM
Unique_Teams JOIN Teams ON Unique_Teams.TeamName = Teams.TeamName LIMIT 5;
```

Filter & Aggregate with Multiple Tables

We can still use the same filtering methods and aggregation methods discussed in my previous blogs when working with multiple tables. In the last query we see data for Bayern Munich is returned for every season. The following query groups together data by each team and displays the average team player age and max number foreign players for each team from all the seasons after the 2000 season.

```
SELECT Unique_Team_ID, Unique_Teams.TeamName, AVG(AvgAgeHome) AS Average_Age,
MAX(ForeignPlayersHome) AS Max_Foreign_Players FROM Unique_Teams JOIN Teams ON
Unique_Teams.TeamName = Teams.TeamName WHERE Season > 2000 GROUP BY Unique_Team_ID,
Unique_Teams.TeamName LIMIT 5;
```

The above query is a bit complicated but it's good to keep in mind that all the techniques discussed from my previous blogs and this blog can be used together to create queries. For now write a query that shows the highest **Match_ID** for each team that ends in a "y" or a "r". Along with the maximum **Match_ID**, display the **Unique_Team_ID** from the **Teams_in_Matches** table and the **TeamName** from the **Unique_Teams** table. Compare your query to the one below:

```
SELECT MAX(Match_ID), Teams_in_Matches.Unique_Team_ID, TeamName FROM Teams_in_Matches JOIN Unique_Teams ON Teams_in_Matches.Unique_Team_ID = Unique_Teams.Unique_Team_ID WHERE (TeamName LIKE '%y') OR ( TeamName LIKE '%r') GROUP BY Teams_in_Matches.Unique_Team_ID, TeamName;
```

INNER JOIN

The **JOIN** statement we have been using so far is also known as an **INNER JOIN**. An **INNER JOIN** is typically the default type of **JOIN** statement. We have seen the below query earlier but this time, instead of writing **JOIN**, we write **INNER JOIN**. The results are the same for both queries.


```
SELECT Teams_in_Matches.Match_ID, Teams_in_Matches.Unique_Team_ID, Unique_Teams.TeamName FROM Teams_in_Matches INNER JOIN Unique_Teams ON Teams_in_Matches.Unique_Team_ID = Unique_Teams.Unique_Team_ID LIMIT 5
```

An inner join takes all the rows from two tables where ever there is a match between two specified columns. If a **Match_ID** is not assigned a **Unique_Team_ID**, then the **Match_ID** will not show up in the results and vice versa, if a **TeamName** is not assigned a **Unique_Team_ID** then that **TeamName** will not appear in the results.

LEFT JOIN

To make the next example easier to understand, we will modify the **Unique_Teams** data table. Run the query below:

```
DELETE FROM Unique_Teams WHERE TeamName LIKE 'O%' OR TeamName LIKE 'F%'
```

There are various types of **JOINs** in addition to the **INNER JOIN**. We will now discuss the **LEFT JOIN**. A **LEFT_JOIN** will return all the rows from the left table/the first table listed in the query, and any matching rows from the right table/the second table listed in the query. If there is no match, then the data from the left table will still be displayed and the data from the right table will show up as **NaN**. To see how a **LEFT JOIN** works, check out the query below:


```
SELECT HomeTeam, FTHG AS HomeGoals, FTAG AS AwayGoals, FTR AS Outcome, Unique_Team_ID FROM
Matches LEFT JOIN Unique_Teams ON Matches.HomeTeam = Unique_Teams.TeamName LIMIT 10
```

The first table, **Matches**, is the left table so all rows from that table are included. The second table, **Unique_Team**, is the right table and only when there is a matching row is data included from that table. For example, there is no corresponding row from the **Unique_Teams** data table where **TeamName** is **Oberhausen** or **Frankfurt FSV**, so the **Unique_Team_ID** shows up as **NaN** in those rows. Run the following code before the next example to help make **LEFT JOINs** easier to understand.

```
DELETE FROM Teams_in_Matches WHERE Unique_Team_ID BETWEEN 20 AND 30;
```

To practice using a **LEFT JOIN**, return all the rows from the **Matches** table, if the Home Team in a match has a **Unique_Team_ID** show the data from the **Teams_in_Matches** table. Only return the first 10 rows. Compare your query with the one below:


```
SELECT * FROM Matches LEFT JOIN Teams_in_Matches ON Matches.Match_ID =
Teams_in_Matches.Match_ID LIMIT 10;
```

This time, we see that the rows where the **Match_ID** is 7, 8 and 9, the second **Match_ID** column and **Unique_Team_ID** columns are **NaN**. This means there was no rows in

Match_ID from the **Teams_in_Matches** data table that were 7, 8 or 9.

RIGHT JOIN

Before moving on, run the following code:

```
DELETE FROM Matches WHERE HomeTeam LIKE 'H%' OR HomeTeam LIKE 'W%';
```

The **RIGHT JOIN** is similar to the **LEFT JOIN**. A **RIGHT JOIN** returns all the rows from the right table/the second table listed, and all the matching rows from the left table/the first table listed. To see how this works look at the query below:


```
SELECT TeamName, Unique_Team_ID, Match_ID, Date, FTHG AS HomeGoals FROM Matches RIGHT JOIN
Unique_Teams ON Unique_Teams.TeamName = Matches.HomeTeam WHERE Season = 2016 LIMIT 10;
```

The query above won't run if you are using Jupyter Notebook because **RIGHT JOINS** are not currently supported. You can go to https://www.w3schools.com/sql/sql_join_right.asp to read more about **RIGHT JOINS** and practice. The above query should return all the rows from the **Unique_Teams** table and only the matches rows from the Matches table. This is basically the opposite of the **LEFT JOIN** and **LEFT JOINS** are used more commonly used than **RIGHT JOINS**.

FULL OUTER JOIN

The **FULL OUTER JOIN** returns all the rows from both the left and right tables and combines the rows when there is a match. It will also return rows when there is no

match. It's kind of like a combination of the **LEFT JOIN** and the **RIGHT JOIN**. The query below should return all the rows from the **Matches** table and the **Unique_Teams** table.


```
SELECT * FROM Matches FULL OUTER JOIN Unique_Teams ON Unique_Teams.TeamName =
Matches.HomeTeam LIMIT 10;
```

The above query will not run if you are using Jupyter Notebook but you can read more about **FULL OUTER JOINS** here, https://www.w3schools.com/sql/sql_join_full.asp.

The below image is a visual representation of the **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN** and the **FULL OUTER JOIN**.

Image via: https://www.w3schools.com/sql/sql_join.asp

UNION Statement

The **UNION** statement is not a **JOIN** statement but it is a way to combine multiple tables. The **UNION** statement combines the results from two tables. The two tables must have the same number of columns and be the same data type. At this point we have deleted some teams from the **Matches** data table and some teams from the **Unique_Teams** table. We can use a **UNION** statement to get a list of all the teams.

SELECT HomeTeam FROM Matches UNION SELECT TeamName FROM Unique_Teams;

The above results combine the two results from **SELECT HomeTeam FROM Matches** and **SELECT TeamName FROM Unique_Teams**. Duplicates are not included but if you wished for duplicates to be included, you could write **UNION ALL** instead of **UNION**.

You have reached the end of this tutorial. The following items were discussed:

- LIMIT Statement
- Returning Multiple Tables without a JOIN Statement
- JOIN Statement
- Return Specific Columns with a JOIN Statement
- Filter & Aggregate with Multiple Tables
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL OUTER JOIN
- UNION Statement

I encourage you to keep practicing using the above methods to gain a deeper understanding of how they work.

Sql Python Data Science Tutorial