Table of Contents

- 1 SQL Aggregation and Join
- 2 Objectives
- 3 Aggregating Functions
  - 3.1 Example Simple Aggregations
- 4 Grouping in SQL
  - 4.1 Example `GROUP BY` Statements
    - 4.1.1 Without `GROUP BY`
    - 4.1.2 With `GROUP BY`
  - 4.2 Group Task
  - 4.3 Exercise: Grouping
- 5 Filtering Groups with `HAVING`
  - 5.1 Examples of Using `HAVING`
    - 5.1.1 Simple Filtering - Number of Airports in a Country
  - 5.2 Filtering Different Aggregations - Airport Altitudes
    - 5.2.1 Looking at the `airports` Table
    - 5.2.2 Looking at the Highest Airport
    - 5.2.3 Looking at the Number of Airports Too
    - 5.2.4 Filtering on Aggregations
- 6 Joins
  - 6.1 `INNER JOIN`
    - 6.1.1 Code Example for Inner Joins
      - 6.1.1.1 Inner Join Routes & Airline Data
      - 6.1.1.2 Note: Losing Data with Inner Joins
  - 6.2 `LEFT JOIN`
    - 6.2.1 Code Example for Left Join
  - 6.3 Exercise: Joins
- 7 Level Up: Execution Order

1 SQL Aggregation and Join

In [ ]:
```
import pandas as pd
import sqlite3

conn = sqlite3.connect("data/flights.db")
cur = conn.cursor()
```

2 Objectives

- Use SQL aggregation functions with GROUP BY
- Use HAVING for group filtering
- Use SQL JOIN to combine tables using keys

3 Aggregating Functions

> A SQL **aggregating function** takes in many values and returns one value.

We have already seen some SQL aggregating functions like `COUNT()`. There are also others, like SUM(), AVG(), MIN(), and MAX().

3.1 Example Simple Aggregations

In [ ]:
```
# Max value for longitude
pd.read_sql('''
    SELECT
        -- Note we have to cast to a numerical value first
        MAX(CAST(longitude AS REAL))
    FROM airports
''', conn)
```

In [ ]:
```
# Max value for id in table
pd.read_sql('''
SELECT
    MAX(CAST(id AS integer))
FROM
    airports
''', conn)
```

In [ ]:
```
# Effectively counts all the inactive airlines
pd.read_sql('''
    SELECT COUNT()
    FROM airlines
    WHERE active='N'
''', conn)
```

We can also give aliases to our aggregations:

In [ ]:
```
# Effectively counts all the active airlines
pd.read_sql('''
    SELECT
        COUNT() AS number_of_active_airlines
    FROM
        airlines
    WHERE
        active='Y'
''', conn)
```

4 Grouping in SQL

We can go deeper and use aggregation functions on *groups* using the `GROUP BY` clause.

The `GROUP BY` clause will group one or more columns together with the same values as one group to perform aggregation functions on.

4.1 Example `GROUP BY` Statements

Let's say we want to know how many active and non-active airlines there are.

4.1.1 Without `GROUP BY`

Let's first start with just seeing how many airlines there are:

In [ ]:
```
df_results = pd.read_sql('''
    SELECT
        -- Reminder that this counts the number of rows before the SELECT
        COUNT() AS number_of_airlines
    FROM
        airlines
''', conn)

df_results
```

One way for us to get the counts for each is to create two queries that will filter each kind of airline

In [ ]:
```
df_active = pd.read_sql('''
    SELECT
        COUNT() AS number_of_active_airlines
    FROM
        airlines
    WHERE
        active='Y'
''', conn)

df_not_active = pd.read_sql('''
    SELECT
        COUNT() AS number_of_not_active_airlines
    FROM
        airlines
    WHERE
        active='N'
''', conn)

display(df_active)
display(df_not_active)
```

This works but it's inefficient.

4.1.2 With `GROUP BY`

Instead, we can tell the SQL server to do the work for us by grouping values we care about for us!

In [ ]:
```
df_results = pd.read_sql('''
    SELECT
        COUNT() AS number_of_airlines
    FROM
        airlines
    GROUP BY
        active
''', conn)

df_results
```

This is great! And if you look closely, you can observe we have *three* different groups instead of our expected two!

Let's also print out the `airlines.active` value for each group/aggregation so we know what we're looking at:

In [ ]:
```
df_results = pd.read_sql('''
    SELECT
        airlines.active,
        COUNT() AS number_of_airlines
    FROM
        airlines
    GROUP BY
        airlines.active
''', conn)

df_results
```

4.2 Group Task

- Which countries have the highest numbers of active airlines? Return the top 10.

In [ ]:
```
pd.read_sql('''
    SELECT
        *
    FROM
        airlines
''', conn)
```

Possible Solution

> Note that the `GROUP BY` clause is considered *before* the `ORDER BY` and `LIMIT` clauses

4.3 Exercise: Grouping

- Run a query that will return the number of airports by time zone. Each row should have a number of airports and a time zone.

In [ ]:
```
# Your code here
```

Possible Solution

5 Filtering Groups with `HAVING`

We showed that you can filter tables with `WHERE`. We can similarly filter *groups/aggregations* using `HAVING` clauses.

5.1 Examples of Using `HAVING`

5.1.1 Simple Filtering - Number of Airports in a Country

Let's come back to the aggregation of active airports:

In [ ]:
```
pd.read_sql('''
    SELECT
        COUNT() AS num,
        country
    FROM
        airlines
    WHERE
        active='Y'
    GROUP BY
        country
    ORDER BY
        num DESC
''', conn)
```

We can see we have a lot of results. But maybe we only want to keep the countries that have more than 30 active airlines:

In [ ]:
```
pd.read_sql('''
    SELECT
        country,
        COUNT() AS num
    FROM
        airlines
    WHERE
        active='Y'
    GROUP BY
        country
    HAVING
        num > 30
    ORDER BY
        num DESC
''', conn)
```

5.2 Filtering Different Aggregations - Airport Altitudes

We can also filter on other aggregations. For example, let's say we want to investigate the `airports` table.

Specifically, we want to know the height of the *highest airport* in a country given that it has *at least* 100 *airports*.

5.2.1 Looking at the `airports` Table

In [ ]:
```
df_airports = pd.read_sql('''
    SELECT
        *
    FROM
        airports
''', conn)

df_airports.head()
```

5.2.2 Looking at the Highest Airport

Let's first get the highest altitude for each airport:

In [ ]:
```
pd.read_sql('''
    SELECT
        airports.country
        ,MAX(
            CAST(airports.altitude AS REAL)
        ) AS highest_airport_in_country
    FROM
        airports
    GROUP BY
        airports.country
    ORDER BY
        airports.country
''', conn)
```

5.2.3 Looking at the Number of Airports Too

We can also get the number of airports for each country.

In [ ]:
```
pd.read_sql('''
    SELECT
        airports.country
        ,MAX(
            CAST(airports.altitude AS REAL)
        ) AS highest_airport_in_country
        ,COUNT() AS number_of_airports_in_country
    FROM
        airports
    GROUP BY
        airports.country
    ORDER BY
        airports.country
''', conn)
```

5.2.4 Filtering on Aggregations

> Recall:
>
> We want to know the height of the *highest airport* in a country given that it has *at least* 100 *airports*.

In [ ]:
```
pd.read_sql('''
    SELECT
        airports.country
        ,MAX(
            CAST(airports.altitude AS REAL)
        ) AS highest_airport_in_country
        -- Note we don't have to include this in our SELECT
        ,COUNT() AS number_of_airports_in_country
    FROM
        airports
    GROUP BY
        airports.country
    HAVING
        COUNT() >= 100
    ORDER BY
        airports.country
''', conn)
```

6 Joins

The biggest advantage in using a relational database (like we've been with SQL) is that you can create **joins**.

> By using `JOIN` in our query, we can connect different tables using their *relationships* to other tables.
>
> Usually we use a key (*foreign key*) to tell us how the two tables are related.

There are different types of joins and each has their different use case.

6.1 INNER JOIN

> An **inner join** will join two tables together and only keep rows if the *key is in both tables*

Example of an inner join:

```
SELECT
    table1.column_name,
    table2.different_column_name
FROM
    table1
    INNER JOIN table2
        ON table1.shared_column_name = table2.shared_column_name
```

6.1.1 Code Example for Inner Joins

Let's say we want to look at the different airplane routes

In [ ]:
```
pd.read_sql('''
    SELECT
        *
    FROM
        routes
''', conn)
```

This is great but notice the `airline_id` column. It'd be nice to have some more information about the airlines associated with these routes.

We can do an **inner join** to get this information!

6.1.1.1 Inner Join Routes & Airline Data

In [ ]:
```
pd.read_sql('''
    SELECT
        *
    FROM
        routes
        INNER JOIN airlines
            ON routes.airline_id = airlines.id
''', conn)
```

We can also specify that we want to retain only certain columns in the `SELECT` clause:

In [ ]:
```
pd.read_sql('''
    SELECT
        routes.source AS departing
        ,routes.dest AS destination
        ,routes.stops AS stops_before_destination
        ,airlines.name AS airline
    FROM
        routes
        INNER JOIN airlines
            ON routes.airline_id = airlines.id
''', conn)
```

6.1.1.2 Note: Losing Data with Inner Joins

Since data rows are kept only if *both* tables have the key, some data can be lost

In [ ]:
```
df_all_routes = pd.read_sql('''
    SELECT
        *
    FROM
        routes
''', conn)

df_routes_after_join = pd.read_sql('''
    SELECT
        *
    FROM
        routes
        INNER JOIN airlines
            ON routes.airline_id = airlines.id
''', conn)
```

In [ ]:
```
# Look at how the number of rows are different
df_all_routes.shape, df_routes_after_join.shape
```

If you want to keep your data from at least one of your tables, you should use a left join instead of an inner join.

6.2 `LEFT JOIN`

> A **left join** will join two tables together and but will keep all data from the first (left) table using the key provided.

Example of a left and right join:

```
SELECT
    table1.column_name,
    table2.different_column_name
FROM
    table1
    LEFT JOIN table2
        ON table1.shared_column_name = table2.shared_column_name
```

6.2.1 Code Example for Left Join

Recall our example using an inner join and how it lost some data since the key wasn't in both the `routes` *and* `airlines` tables.

In [ ]:
```
df_all_routes = pd.read_sql('''
    SELECT
        *
    FROM
        routes
''', conn)

# This will lose some data (some routes not included)
df_routes_after_inner_join = pd.read_sql('''
    SELECT
        *
    FROM
        routes
        INNER JOIN airlines
            ON routes.airline_id = airlines.id
''', conn)

# The number of rows are different
df_all_routes.shape, df_routes_after_inner_join.shape
```

If wanted to ensure we always had every route even if the key in `airlines` was not found, we could replace our `INNER JOIN` with a `LEFT JOIN`:

In [ ]:
```
# This will include all the data from routes
df_routes_after_left_join = pd.read_sql('''
    SELECT
        *
    FROM
        routes
        LEFT JOIN airlines
            ON routes.airline_id = airlines.id
''', conn)

df_routes_after_left_join.shape
```

6.3 Exercise: Joins

Which airline has the most routes listed in our database?

In [ ]:
```
# Your code here
```

Possible Solution

7 Level Up: Execution Order

```
SELECT
    COUNT(table2.col2) AS my_new_count
    ,table1.col2
FROM
    table1
    JOIN table2
        ON table1.col1 = table2.col2
WHERE
    table1.col1 > 0
GROUP BY
    table2.col1
```

1. From
2. Where
3. Group By
4. Having
5. Select
6. Order By
7. Limit