Table of Contents

SQL Subqueries

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

Objectives

- Use SQL subqueries to nest queries
- Identify common SQL dialects and tools
- Query data from web databases

SQL Subqueries

Like you might nest one function within another in Python, you can nest queries in SQL. We can use a **subquery** within another query to succinctly implement queries that have multiple query steps.

In [ ]:
```
conn = sqlite3.connect('data/flights.db')
```

Subqueries in `FROM`

You can use a subquery in the `FROM` clause - this is useful, for example, if you want to apply multiple aggregation functions.

Let say we want to get the average of the number of routes departing from all airports. First we'd need to get the total number of routes departing from all airports, then take the average.

```
In [ ]:
pd.read_sql('''
SELECT
    source AS depart_airport
    , COUNT() AS number_of_departures
FROM
    routes
GROUP BY
    source
''', conn)
```

We can use this query as a subquery, and take the average of the new `number_of_departures` column.

```
In [ ]:
pd.read_sql('''
SELECT
    AVG(number_of_departures)
FROM (
    SELECT
        source AS depart_airport
        ,COUNT() AS number_of_departures
    FROM
        routes
    GROUP BY
        source
)
''', conn)
```

Note: Subqueries are Like New Tables!

If you squint, you'll notice that the subquery is taking the place of where we might put a table!

For example, checkout the SQL we wrote in our first subquery example:

```
SELECT
    AVG(number_of_departures)
FROM (
    SELECT
        source AS depart_airport
        ,COUNT() AS number_of_departures
    FROM
        routes
    GROUP BY
        source
)
```

We could imagine that some new table that returned by the subquery existed (let's call it `airport_departures`) and be placed in place of the subquery:

```
SELECT
    AVG(number_of_departures)
FROM (
    airport_departures -- Replacing subquery with this hypothetical table
)
```

You can actually use syntax close to this with **Common Table Expressions (CTEs)** found in the Level Up section below.

Subqueries in `WHERE`

You can use a subquery in the `WHERE` clause - this is useful, for example, if you want to filter a query based on results from another query.

Let's say that we want to get a table with all of the departures and destinations for the flight routes, but I only want to include flights departing from the five countries with the most airports.

To do this, we'd first need to identify the five countries that have the most airports.

In [ ]:
```
pd.read_sql('''
SELECT
    country
    ,COUNT() AS number_of_airports_in_country
FROM
    airports
GROUP BY
    country
ORDER BY
    number_of_airports_in_country DESC
LIMIT 5
''', conn)
```

I could enter these results into a new query of the routes table to get the data I want.

In [ ]:
```
pd.read_sql('''
SELECT
    rt.source AS depart_airport
    ,rt.dest AS destination_airport
    ,ap.country AS depart_country
FROM
    routes AS rt
    LEFT JOIN airports AS ap
        ON rt.source_id = ap.id
WHERE
    ap.country IN (
        "United States",
        "Canada",
        "Germany",
        "Australia",
        "Russia"
    )
ORDER BY
    depart_country
''', conn)
```

This approach works but has a few limitations:

- We have to manually enter the countries to filter them
- The list of countries won't update with our data, so we'd have to monitor and manually change them in the future
- We have to look at two separate queries to understand what our code is supposed to do
- We have to run two separate queries, which might take longer than one combined query

A better solution uses a subquery to get the list of 5 countries and feed it into our WHERE clause.

In [ ]:
```
pd.read_sql('''
SELECT
    rt.source AS depart_airport
    ,rt.dest AS destination_airport
    ,ap.country AS depart_country
FROM
    routes AS rt
    LEFT JOIN airports AS ap
        ON rt.source_id = ap.id
WHERE ap.country IN (
-- Subquery to get the 5 countries with the most airports
    SELECT
        country
    FROM
        airports
    GROUP BY
        country
    ORDER BY
        COUNT() DESC
    LIMIT 20
)

ORDER BY
    depart_country
''', conn)
```

Level Up: Common Table Expressions

Common Table Expressions (CTEs) are a more readable way to implement subqueries, using `WITH` and `AS` .

In [ ]:
```
pd.read_sql('''
WITH top_5_countries AS (
    SELECT
        country
    FROM
        airports
    GROUP BY
        country
    ORDER BY
        COUNT() DESC
    LIMIT 5
)

SELECT
    rt.source AS depart_airport
    ,rt.dest AS destination_airport
    ,ap.country AS depart_country
FROM
    routes AS rt
    LEFT JOIN airports AS ap
        ON rt.source_id = ap.id
WHERE
    ap.country IN top_5_countries
ORDER BY
    depart_country
''', conn)
```

Exercise

Create a table listing all airlines that serve the three airports with the most outbound routes.

In [ ]:

```
## Your work here

```
SELECT ....
FROM ....
WHERE ....
ORDER BY ...
```
```

▶ **Click Here for Answer Code**

SQL Versions

There is no one version of SQL - there are many versions out there! What you're learning about SQL with SQLite will apply to all of them. Just keep in mind when you apply for jobs that you may see any of these listed in any given job posting, and they are all just different versions of what you know.

SQL Dialects

As with dialects of spoken languages, SQL dialects have many commonalities but some differences in syntax and functionality. Here are a few of the major players:

- SQLite (we've already seen this!)
- PostgreSQL (free and open-source!)
- Oracle SQL
- MySQL (half open-souce, half Oracle)
- Microsoft SQL Server
- Transact-SQL (extends MS SQL)

SQLite Pros & Cons

We use SQLite in this course, but it has some limitations.

Pros

- Easy to set up
- Easy to share database files
- Uses little memory

Cons

- Limited functionality for managing users and access permissions
- Not "thread safe": two edits at the same time can mess up your data

Extra Resources: SQL Versions

What Is a SQL Dialect, and Which one Should You Learn?

SQLite vs MySQL vs PostgreSQL

SQL Dialect Reference