Table of Contents

1 Introduction to SQL Queries

2 Objectives

- Describe relational databases
- Connect to a SQLite database and get schema information
- Use SQL SELECT and `pd.read_sql()` to query databases
- Use WHERE, ORDER BY, and LIMIT to modify queries

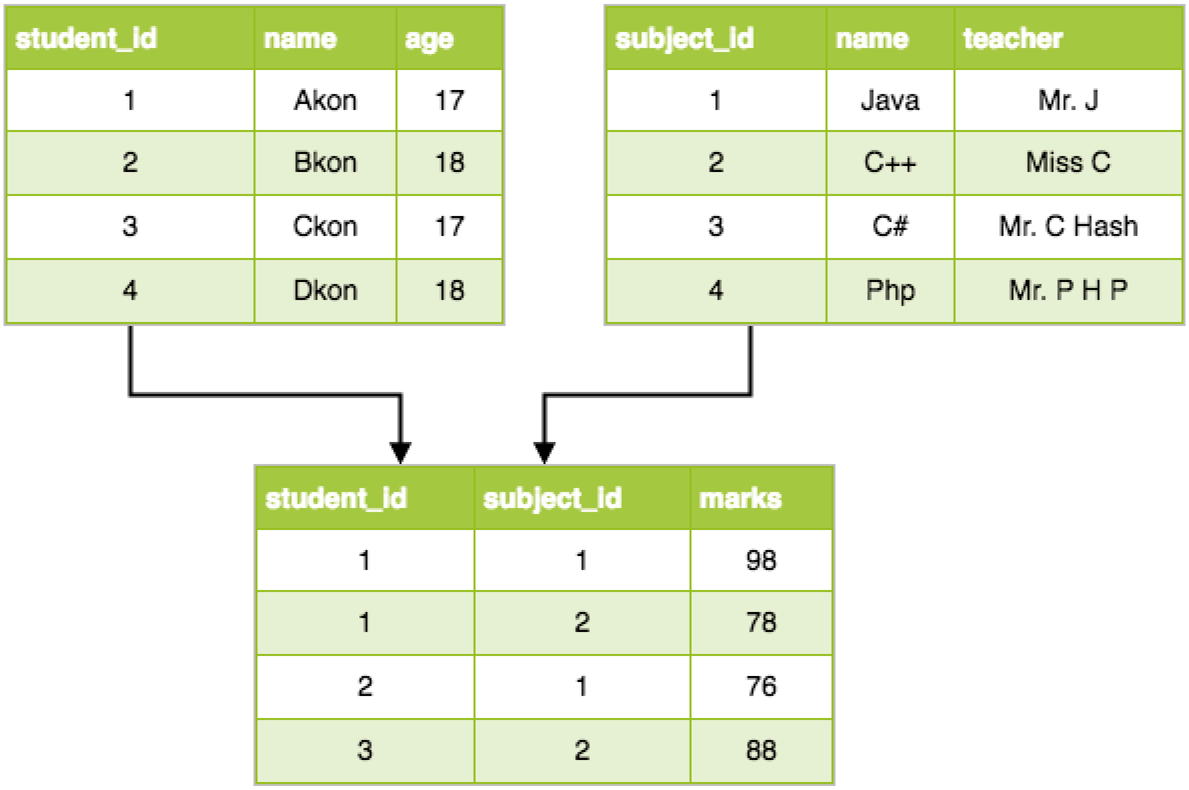
3 Motivation

Most data aren't stored in static files like CSVs or JSONs. Rather, data are typically stored in **databases** that make it easy for many users to store, update, share, and access data in real time. CSVs and JSONs are just extracts of some data from those databases.

Structured Query Language (SQL) is a common language for interacting with databases, and will be invaluable for you in almost any data role. You will use it often to get the data that you need for your analyses.

4 Relational Databases

Relational databases typically have multiple **tables** containing data, and the tables have defined relationships.



student_id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

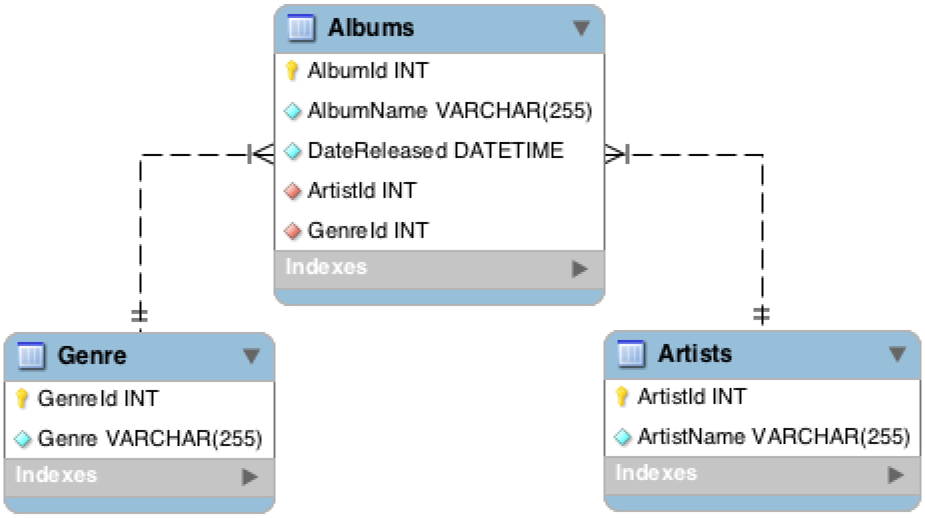
subject_id	name	teacher
1	Java	Mr. J
2	C++	Miss C
3	C#	Mr. C Hash
4	Php	Mr. P H P

student_id	subject_id	marks
1	1	98
1	2	78
2	1	76
3	2	88

reference for image (https://www.studytonight.com/dbms/database-model.php)

4.1 Database Schema

Each database has a **schema** that defines the structure of the database, including the tables and relationships between tables.



source of image (https://database.guide/what-is-a-database-schema/)

4.2 Columns

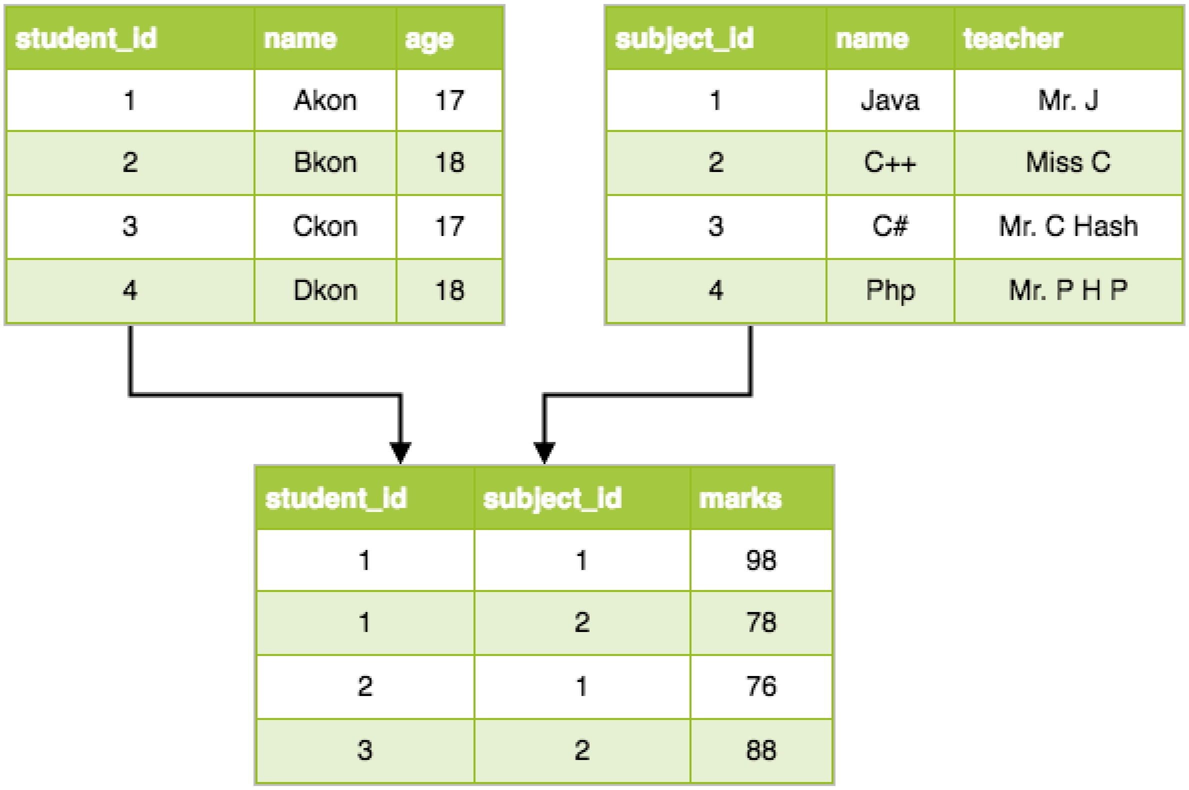
Similar to how DataFrames can have multiple Series, tables can have multiple **columns** (aka "fields"). Each column has a datatype, but the datatypes available for SQL table columns differ from the datatypes in `pandas` .

Books	SQL Data Type
ID (book)	integer
Title	Char (200)
Author	Char (200)
Genre	Char (100)
Year	Date
Language	Char (50)

4.3 Keys

A **primary key** uniquely identifies each row in a table. This is often a unique ID number. A **foreign key** is used in one table to refer to the primary key from another table.

We **join** tables using these keys to get data from multiple tables at once - we will cover this in a future lesson on SQL Joins.



student_Id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

subject_Id	name	teacher
1	Java	Mr. J
2	C++	Miss C
3	C#	Mr. C Hash
4	Php	Mr. P H P

student_Id	subject_Id	marks
1	1	98
1	2	78
2	1	76
3	2	88

reference for image (https://www.studytonight.com/dbms/database-model.php)

5 SQLite

SQLite is one of many tools that exist to create databases. We use it here because it is easy to integrate into a Jupyter Notebook using the `sqlite3` package.

There are many other database tools out there, and they all work somewhat differently with their own SQL dialects. Just know that the specific methods or syntax you see here will differ slightly in other database implementations.

5.1 *Sidebar: More About SQLite*

"SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle." - sqlite documentation (https://docs.python.org/2/library/sqlite3.html)

5.2 Load a SQLite DB

Import the `sqlite3` package, which will allow us to load a SQLite database.

```
In [ ]: import sqlite3
```

```
executed in 3ms, finished 10:25:50 2021-10-20
```

In this repository is a `flights.db` file that we can open as a SQLite database. This database contains tables with information about airlines, airports, and flight routes.

In [ ]:
```
!ls
```

executed in 142ms, finished 10:26:31 2021-10-20

First, we'll use the `sqlite3` package to create a connection to the database, which is currently just stored in that file on our hard drive.

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

executed in 4ms, finished 10:29:03 2021-10-20

Next, we'll create a cursor to interact with the database. Like the cursor for your mouse interacts with pixels on your screen, this cursor will allow us to interact with the elements of the database.

In [ ]:
```
cursor = con.cursor()
```

executed in 3ms, finished 10:29:40 2021-10-20

5.3 Query the `airports` Table

We will write a simple query using the SQL SELECT statement, which returns data from the database. We write the query as a string, which will then get parsed via the `sqlite3` package.

In this case, we say `SELECT *` to specify that we want data from all columns, and we say `FROM airports` to specify that we want data from the `airports` table.

In [ ]:
```
airports_query = \
"""
SELECT *
FROM airports
"""
```

executed in 3ms, finished 10:33:11 2021-10-20

To run this query, we use the `.execute()` method with our cursor.

In [ ]:
```
cursor.execute(airports_query)
```

executed in 13ms, finished 10:33:34 2021-10-20

Note that the `.execute()` method didn't actually return our data. The data is now just available in our cursor object. We'll use the `.fetchall()` method to get all the rows from our query.

In [ ]:
```
cursor.fetchall()
```

executed in 257ms, finished 10:34:51 2021-10-20

Looks like we got some data, but it's not clear what each element represents. We can view the column names in the cursor's `description` attribute.

In [ ]:

```
cursor.description
```

executed in 5ms, finished 10:37:01 2021-10-20

5.4 pd.read_sql()

We can get the data and the column names into a nice, tidy DataFrame using `pd.read_sql()`

In [ ]:

```
import pandas as pd

airports_df = pd.read_sql(airports_query, con)
airports_df.head()
```

executed in 1.68s, finished 10:40:45 2021-10-20

5.5 Explore the Schema

In SQLite, the schema of our database lives in the `sqlite_master` table. More info here (https://www.techonthenet.com/sqlite/sys_tables/index.php).

In [ ]:

```
schema_df = pd.read_sql("""

SELECT *
FROM sqlite_master

""", con)

schema_df
```

executed in 12ms, finished 10:44:40 2021-10-20

In [ ]:

```
cursor.execute("""
SELECT *
FROM sqlite_master
""").fetchall()
```

It looks like there are three tables in our database: airports, airlines, and routes. Each table also has an **index**, which is used to optimize queries for large databases.

The column names and datatypes for each table are defined in the schema in the `sql` column.

In [ ]:

```
# Airports table info

print(schema_df['sql'].iloc[0])
```

executed in 3ms, finished 10:51:58 2021-10-20

5.6 Exercise

Get the columns and datatypes for the airlines table. You can do this with either `pd.read_sql()` or the schema table.

Click Here for Answer Code

In [ ]:

```
# Your work here
```

executed in 3ms, finished 10:54:45 2021-10-20

6 Writing SQL Queries

In this section we will build SQL queries using the SELECT statement, showing off a bunch of different clauses and options available.

6.1 SELECT Statement

SELECT statements can have multiple **clauses**, which must be included in a specific order (more info here (https://sqlite.org/lang_select.html)). Only SELECT and FROM are required.

Let's explore the following clauses and structure:

```
SELECT columns
FROM table
WHERE condition
ORDER BY columns
LIMIT number
```

6.2 SELECT: Picking Columns

Add the names of the columns that you want after the word `SELECT`, or use `*` to get all columns.

In [ ]:

```
pd.read_sql("""

SELECT city, country
FROM airports

""", con)
```

executed in 23ms, finished 10:57:38 2021-10-20

6.2.1 DISTINCT

Use `DISTINCT` to drop duplicates.

In [ ]:

```
pd.read_sql("""

SELECT DISTINCT city, country
FROM airports

""", con)
```

executed in 30ms, finished 10:59:05 2021-10-20

6.2.2 AS

Use `AS` to rename columns.

In [ ]:

```
pd.read_sql("""

SELECT city AS "Airport City",
       country AS "Airport Country"
FROM airports

""", con)
```

executed in 23ms, finished 11:00:34 2021-10-20

6.2.3 Functions

There are dozens of functions that you can use in SELECT statements to modify results - you can see some examples here (https://sqlite.org/lang_corefunc.html).

In [ ]:

```
pd.read_sql("""

SELECT UPPER(name) AS "AIRPORT NAMES IN CAPS",
       LENGTH(name) AS "Airport Name Length"

FROM airports

""", con)
```

executed in 27ms, finished 11:01:33 2021-10-20

6.2.3.1 Aggregation

Some functions will aggregate your data and return a table with one row.

In [ ]:

```
pd.read_sql("""

SELECT COUNT() AS "Number of Airports"
FROM airports

""", con)
```

executed in 11ms, finished 11:03:56 2021-10-20

6.2.3.2 Datatype Compatibility

Make sure that your column is the right datatype for the function to avoid unexpected results.

In [ ]:

```
pd.read_sql("""

SELECT name AS "Airport Name",
       MAX(altitude) AS "Altitude (ft)"

FROM airports

""", con)
```

executed in 9ms, finished 11:05:13 2021-10-20

6.2.3.3 `CAST()`

You could fix this using the `CAST()` function.

In [ ]:

```
pd.read_sql("""

SELECT name AS "Airport Name",
       MAX(CAST(altitude as int)) AS "Altitude (ft)"

FROM airports

""", con)
```

executed in 10ms, finished 11:07:47 2021-10-20

6.2.4 Exercise

Which country has the northern-most airport?

> *Hint: Look for the highest latitude*

Click Here for Answer Code

In [ ]:

```
# Your work here
```

6.3 FROM: Picking Tables

The `FROM` clause specifies the tables you get data from. You can use aliases here with `AS` - this will be useful for more complex queries involving multiple tables.

In [ ]:

```
pd.read_sql("""

SELECT ap.name AS "Airport Name",
       MAX(CAST(ap.altitude AS int)) AS "Altitude (ft)"

FROM airports AS ap

""", con)
```

6.4 WHERE: Picking Rows

The `WHERE` clause filters results from your query. This uses conditional logic and operators similar to Python's - you can find more here (https://sqlite.org/lang_expr.html).

In [ ]:

```
pd.read_sql("""

SELECT name AS "Airport Name",
       CAST(latitude AS int) AS "Airport Latitude",
       CAST(altitude AS int) AS "Altitude (ft)"

FROM airports

WHERE "Altitude (ft)" >= 10000 AND
      "Airport Latitude" BETWEEN 20 AND 50

""", con)
```

6.4.1 `IS`

The `IS` operator is useful when working with `NULL` values - other operators will not work as expected.

In [ ]:

```
pd.read_sql("""

SELECT name AS "Airport Name",
       code AS "Airport Code"

FROM airports

WHERE "Airport Code" IS NOT NULL

""", con)
```

6.5 ORDER BY: Sorting Results

Use `ORDER BY` to identify the column(s) you want to sort on. Specify `ASC` for ascending order, `DESC` for descending order.

In [ ]:

```
pd.read_sql("""

SELECT name AS "Airport Name",
       CAST(latitude AS int) AS "Airport Latitude",
       CAST(altitude AS int) AS "Altitude (ft)"

FROM airports

WHERE "Altitude (ft)" >= 10000

ORDER BY "Airport Latitude" DESC,
         "Altitude (ft)" DESC

""", con)
```

6.6 LIMIT: Number of Results

Specify the maximum number of results you want

In [ ]:

```
pd.read_sql("""

SELECT name AS "Airport Name",
       CAST(latitude AS int) AS "Airport Latitude",
       CAST(altitude AS int) AS "Altitude (ft)"

FROM airports

WHERE "Altitude (ft)" >= 10000

ORDER BY "Airport Latitude" DESC,
         "Altitude (ft)" DESC

LIMIT 10

""", con)
```

7 Exercises

7.1 Country List

Create a list of countries with airports and order them alphabetically A-Z.

> *Hint: You will need to remove duplicates.*

Click Here for Answer Code

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

7.2 Southern Airports

Get the name, country and latitude of the 10 southern-most airports.

> *Hint: Look for the smallest latitude.*

Click Here for Answer Code

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

7.3 Active UK Airlines

Create a list of active airlines in the United Kingdom from the airlines table.

> *Hint: You will need to explore the airlines table to figure out how to do this.*

Click Here for Answer Code

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

7.4 Explore Routes

Get the column names from the routes table and inspect some raw data. Which columns might be keys that connect this table to the other two tables?

Click Here for Answer Code

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

8 Level Up: `CASE`

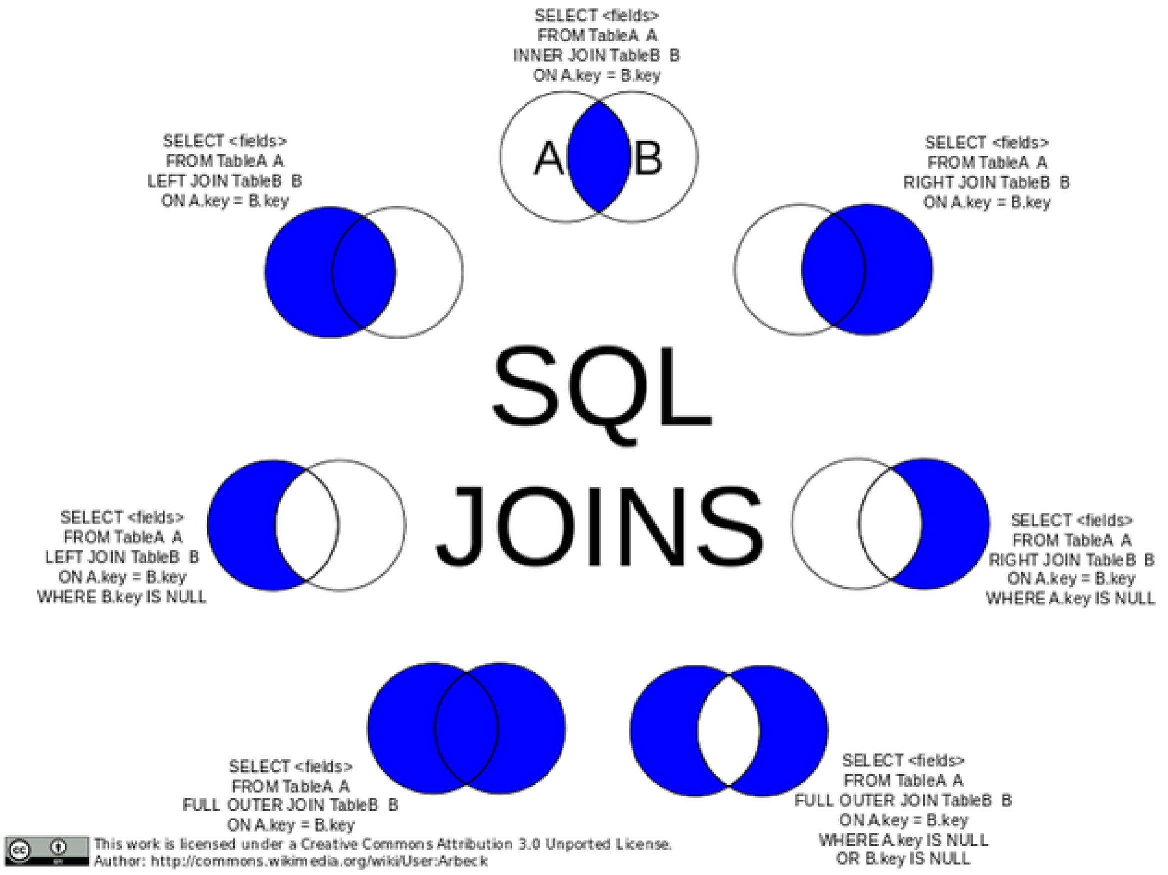
Use `CASE` to create new columns using conditional logic.

In [ ]:

```
pd.read_sql("""

SELECT name AS "Airport Name",
       CAST(altitude AS int) AS "Altitude (ft)",
       CASE
            WHEN CAST(altitude AS int) > 1000 THEN "High"
            WHEN CAST(altitude AS int) < 100 THEN "Low"
            ELSE "Moderate"
       END AS "Altitude Category"

FROM airports

LIMIT 20

""", con)
```

9 Level Up: SQL Joins

SQL joins can be used to both **add** data to a table and **remove** data from a table.



```
SELECT <fields>
FROM TableA A
INNER JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
WHERE B.key IS NULL
```

```
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
```

```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

How are these different joins possible?

Notice that I choose a column from each table "on" which to effect the join. This is the means by which I pair up the records from one table with the records of another.

Look back up at the sample diagram under "What is a Relational Database?". We might use the "student_id" column to match up names in the names table with grades in the grades table. But what if there are values in one table's version of "student_id" that don't appear in the other table's version? In that case we need to let the software know whether or not we want to have *all* of the records, regardless of whether they have corresponding entries in all the tables we are joining. This makes for the variety depicted above.

- If I select records from "A INNER JOIN B", then a record will be displayed *only if it exists in both tables*.
- If I select records from "A LEFT JOIN B", then *all relevant records from A will be displayed*, regardless of whether they have representation in B. Records from B with no representation in

In [ ]:

```
pd.read_sql("""

    SELECT p.name, l.name, p.country
    FROM airports p
    LEFT JOIN airlines l
    ON p.country=l.country
    ORDER BY l.name
    LIMIT 5

""", con)
```

In [ ]:

```
pd.read_sql("""

    SELECT p.name, l.name, p.country
    FROM airports p
    INNER JOIN airlines l
    ON p.country=l.country
    ORDER BY l.name
    LIMIT 5

""", con)
```