Relational Databases and SQL: An Introduction

Sean Sall

May 11th, 2016

Objectives

Today's objectives:

- Explain why we use relational database management systems (RDBMS) in conjunction with SQL
- Explain what it means for SQL to be a *declarative* language, how this is different from Python (which is *imperative*), and why this matters
- Write simple queries on a single table using **SELECT**, **FROM**, **WHERE**, **ORDER BY**, **GROUP BY**, and **CASE** clauses as well as aggregation functions (**MIN**, **MAX**, **AVG**, etc.)
- Write complex queries including **JOINS** and subqueries

Agenda

- Why relational databases?
- Why SQL?
- Basic *clauses*:
  - **SELECT**
  - **FROM**
  - **WHERE**
- More advanced *clauses*:
  - **GROUP BY**
  - **ORDER BY**
  - **CASE**
- SQL **JOINS** and Subqueries
- SQL Order of Operations

Note: All code snippets in this lecture are in `Postgres` syntax

Why does this matter?

- Relational Databases
  - Large amounts of data are stored in relational databases, and having a basic understanding of how they work can help when working with that data
- SQL
  - Structured Query Language (SQL) is the language we use to interact with relational databases

Relational Database Management Systems (RDBMS)

- What are they?
  - One way of storing *persistent* data, i.e. data that:
    - ★ Survives after the process in which it was created has ended
    - ★ Is written to non-volatile storage
    - ★ Is infrequently accessed and unlikely to be changed
  - A *significant portion* of business data is stored in RDBMS
    - ★ They are effectively the *de facto* standard for storing data (although it could be argued that this is recently starting to change due to the volumes of data we are storing - e.g. "Big Data")
    - ★ Popular RDBMS - Oracle, MySQL, SQL Server, SQLLite, PostgreSQL (and more)

Why RDBMS?

- An RDBMS provides the ability to :
  - model relations in the data (via an Object-Relational-Model, or ORM)
  - query data and their relations efficiently
  - maintain data consistency and integrity
  - share data easily between programming languages

RDBMS Data Model

- A RDBMS is composed of a number of user-defined **tables**, each with **columns (fields)** and **rows (records)**
  - each column is of a certain **data type** (integer, string, date)
  - each row is an entry in the table (an observation) that holds values for each one of the columns
  - tables are specified by a **schema** that defines the structure of the data

RDBMS Distributions

- Popular versions of RDBMS include:
  - SQLite
  - PostgreSQL
  - MySQL
  - Oracle Database
  - SQL Server

PostgreSQL cheat sheet - Part 1

- To log in from the terminal, you can log into your default database (usually your username) by just typing in `psql`, or a particular database by specifying its name after the `psql`:

```
psql # logs into default database
psql my_playground # logs into 'my_playground' database
```

- From within `psql`, you can create a new database with:

```
CREATE DATABASE <name>;
```

Note: We have to start the PostgreSQL server process before doing anything else!

PostgreSQL cheat sheet - Part 2

- Useful commands at the `psql` prompt:

`\l` - list all the tables in the database
`\d <table name>` - describe a table's schema
`\h <clause>` - HELP! (for SQL clause help)
`\?` - HELP! (for psql commands like these)
`\q` - quit

Sample Schema

A **schema** defines what **columns** a table will have, and what the **data type** of each column will be:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name VARCHAR(255),
    nickname VARCHAR(255),
    fav_disney_char INTEGER,
    tswift_rating INTEGER
);
```

Note: *VARCHAR* is the `Postgres` equivalent of a `str`

Primary/Foreign Keys - The Basics

- A **primary** key is a column or combination of columns that uniquely identify an entry in the table
  - We have to be explicit about what column(s) we want to be our primary key (it doesn't just happen by default)
  - Each entry in the table **must** have a value for the primary key
  - We can't insert a second entry that has the same primary key as another
- A **foreign** key is a column or combination of columns that uniquely identify an entry of another table.
  - We also have to be explicit about what column(s) we want to be our foreign key

Primary/Foreign Keys - A Sample Schema

```
CREATE TABLE disney_chars (
    id INTEGER PRIMARY KEY,
    name VARCHAR(255),
    princess BOOLEAN,
    prince BOOLEAN,
    mulan BOOLEAN
);

CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name VARCHAR(255),
    nickname VARCHAR(255),
    fav_disney_char INTEGER references disney_chars(id),
    tswift_rating INTEGER
);
```

Note: Primary/Foreign keys don't have to be an `INTEGER`.

Foreign Keys and Relationships

- Foreign keys inherently model a couple of different kinds of relationships between observations in different tables:
  - `one` to `one` (users and personal cell #'s - foreign key goes on *either* side)
  - `one` to `many` (users and purchases - foreign key goes on `many` side)
  - `many` to `many` (items and purchases - *each* table gets a foreign key that references a separate, third table in the middle)

Schema Normalization

- Minimize redundancy. For example:
  - Details about a user (address, age, etc. . .) are only stored **once** (in a *users* table)
  - Any other table (*purchases*, for example) where this data might be relevant only references the *user_id* from the *users* table
- This adds in reliability and data integrity guarantees, i.e. a piece of data only has to be changed in one place, meaning that we can't end up with contradicting pieces of information

Structured Query Language (SQL)

- As a data scientist, your main interaction with RDBMS will be to *extract* information that already exists in a database (you won't be building them often, if at all)
- SQL is the language used to query relational databases
  - All RDBMS use SQL and the syntax and keywords are for the most part the same across systems
  - Even non-relational databases like Hadoop usually have a SQL-like interface available
- SQL allows you to create tables, alter tables, insert records, update records, delete records, and query records within and across tables

SQL - Basic Clauses

All SQL queries have three main ingredients:

SELECT *__What__* data (columns) do you want?
FROM *__Where__* do you want to get the data from?
WHERE *__Under__* what conditions?

An important characteristic of SQL is that it is a *declarative* language, rather than *imperative*. This means that you tell the machine (database) what you want (via the query), and it decides how to actually do it.

Basic Clauses - Examples

- Grab the *unique* names of the Taylor Swift fanatics:

```
SELECT DISTINCT name FROM users WHERE tswift_rating = 10;
```

- Grab all information about the character Mulan:

```
SELECT * FROM disney_chars WHERE name = 'Mulan';
```

Note: SQL queries always return a table.
Note: You can include multiple conditions in the **WHERE**, separated by an **AND**/**OR**.

SQL More Advanced Clauses

Once we know what we want, we can use **ORDER BY** or **GROUP BY** to sort the data or calculate some aggregate statistic.

- Order all Taylor Swift fans by their rating (that way I can steer clear of those people who aren't as big of a fan as me):

```
SELECT name FROM users ORDER BY tswift_rating DESC;
```

- Find the average `tswift_rating` by different `fav_disney_char`:

```
SELECT AVG(tswift_rating) FROM users GROUP BY
fav_disney_char;
```

- Other **aggregate** functions include `MIN`, `MAX`, `COUNT`, `SUM`, `STD`, etc...
- **Note**: Anything in the **SELECT** statement that is not an aggregator has to be in the **GROUP BY**

SQL Case Statements

Case Statements are basically a SQL version of an `if-then-else`.

- Count the number of Taylor Swift fans, based off their `tswift_rating` (higher than 5 is a fan, otherwise not a fan):

```
SELECT SUM(CASE WHEN tswift_rating > 5
    THEN 1 ELSE 0 END) AS fan_count
FROM users;
```

Aliasing: Uses the **AS** to allow you to rename a column/field in a query

SQL Joins

- The **JOIN** clause enables querying based on relations in the data by making use of foreign keys.
- Every **JOIN** clause has two segments:
  1. specifying the tables to join
  2. specifying the columns to match up (occurs in the **ON** clause)

SQL Join Types

The various **JOIN** types specify how to deal with different circumstances regarding the primary and foreign key matchings (e.g. does each observation from one table line up with one, many, or no observations from the other table, and vice versa):

- **INNER JOIN** discards any entries that do not have a match between the keys specified in the **ON** clause
- **LEFT OUTER JOIN** keeps all entries in the left table, regardless of whether a match is found in the right table
- **RIGHT OUTER JOIN** is the same, except it keeps the entries in the right table instead of the left
- **FULL OUTER JOIN** will keep the rows of both tables no matter what

SQL Inner Join - Example

- What if I wanted to see if the `fav_disney_char` of most Taylor Swift fans is a Disney princess?

```
SELECT u.name, u.fav_disney_char, dc.princess
FROM users AS u
INNER JOIN disney_chars AS dc
    ON u.fav_disney_char = dc.id;
```

Note: You can **JOIN** more than two tables by adding in another **JOIN** statement right after the previous one.

Visualizing Joins - Inner Joins

INNER JOIN discards any entries that do not have a match between the keys specified in the **ON** clause

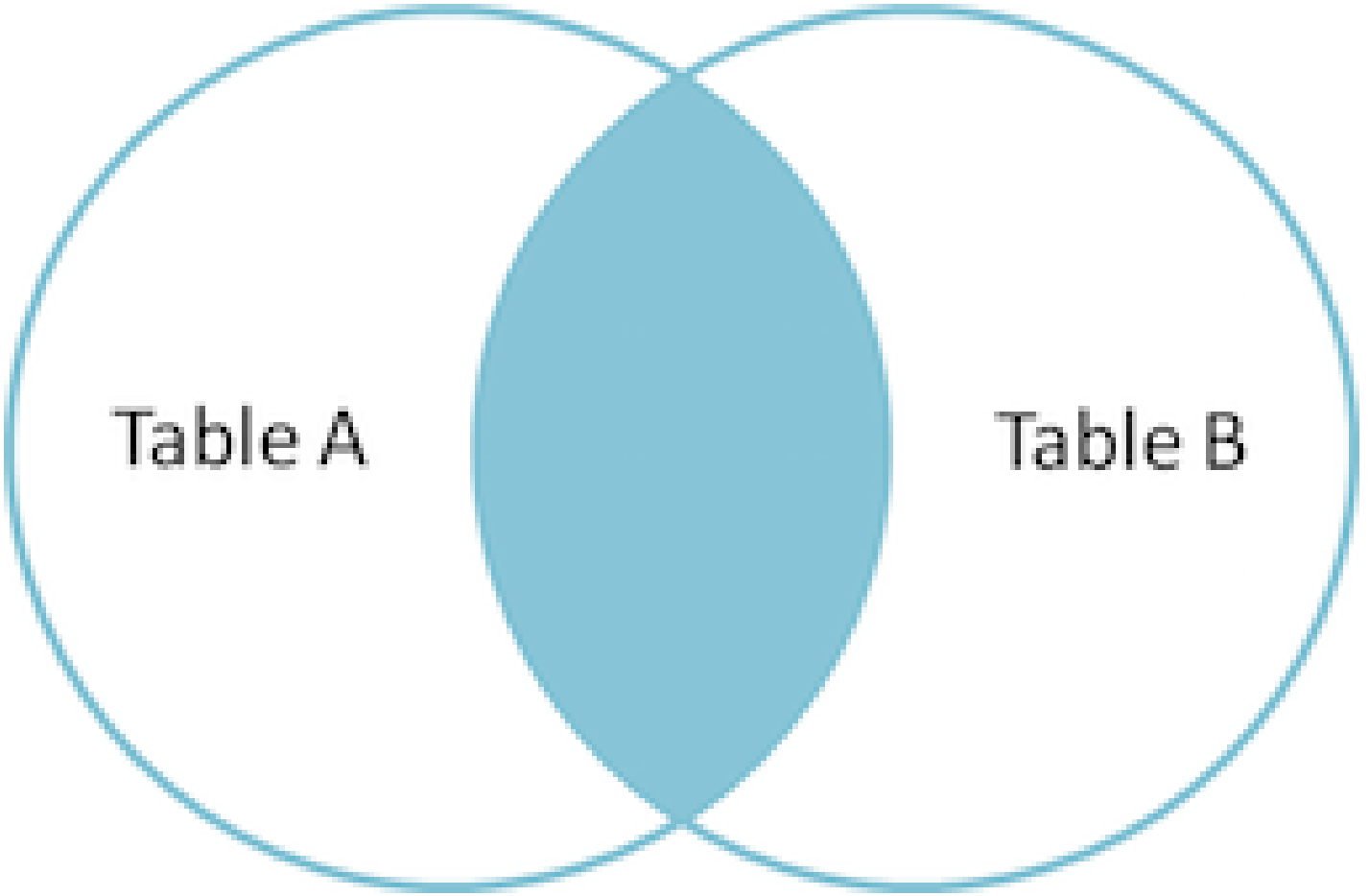

Figure 1:Inner Joins

Visualizing Joins - Left Outer Joins

LEFT OUTER JOIN keeps all entries in the left table, regardless of whether a match is found in the right table

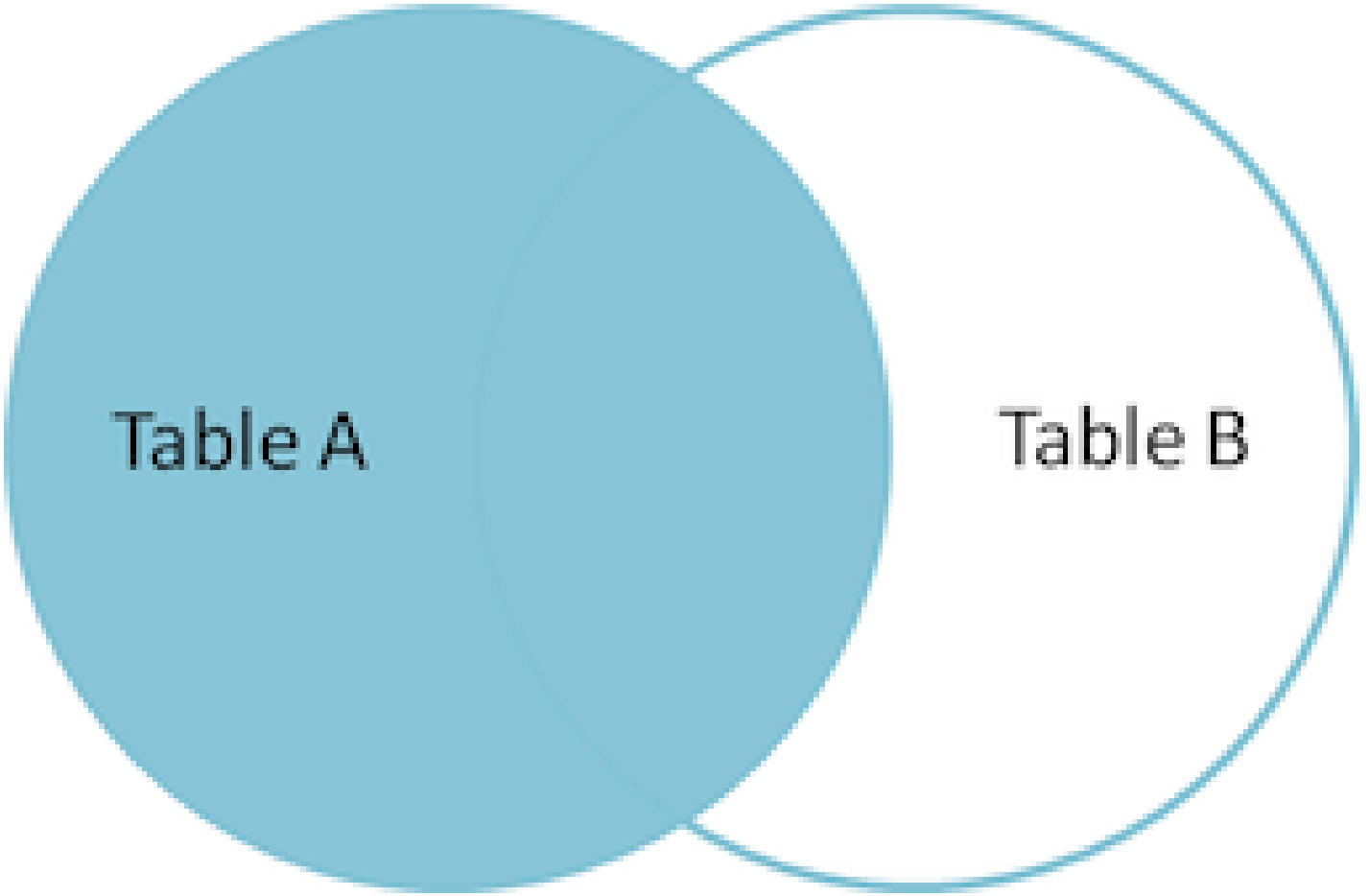

Figure 2:Left Outer Joins

Visualizing Joins - Full Outer Joins

FULL OUTER JOIN will keep the rows of both tables no matter what

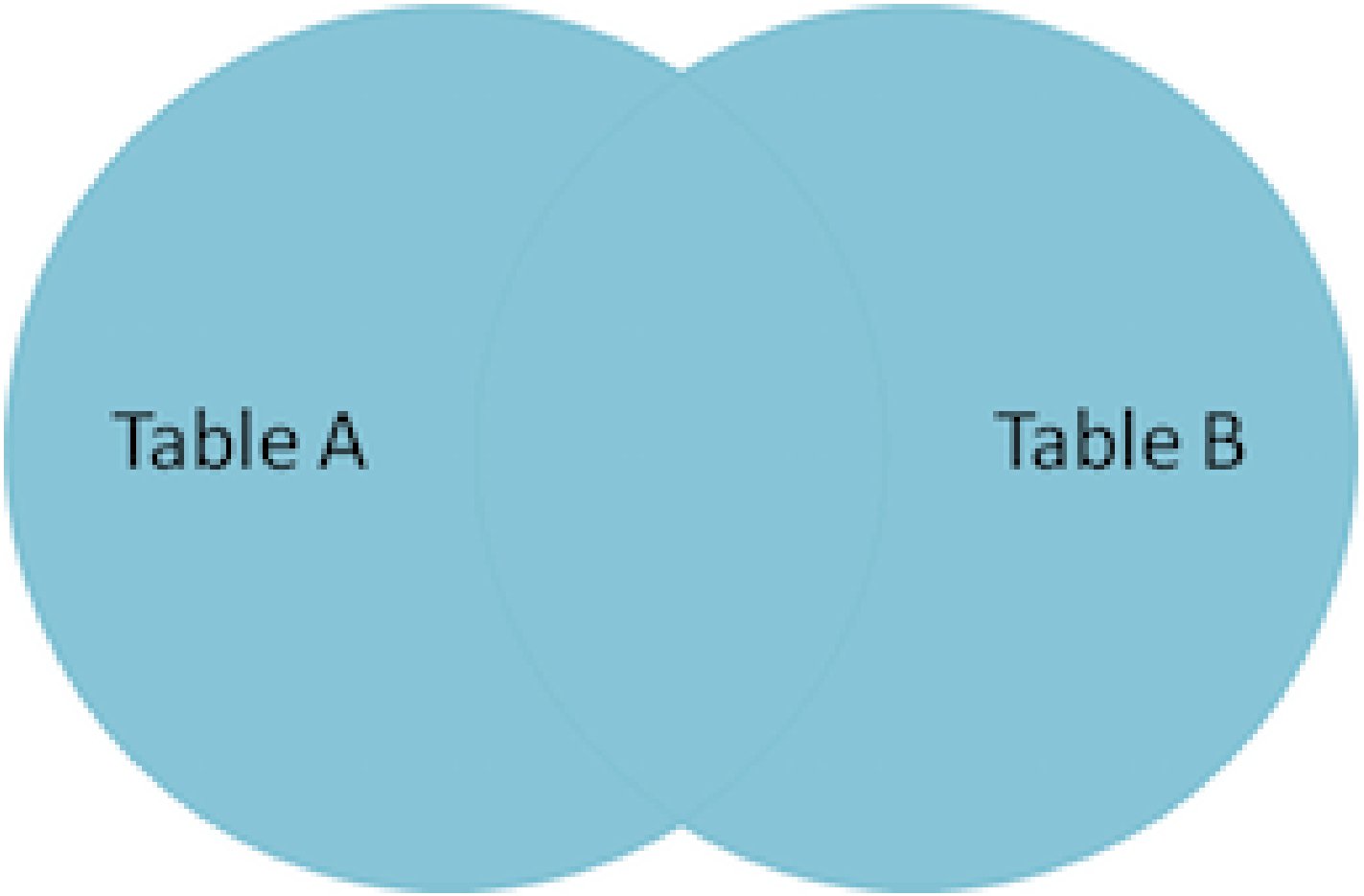

Figure 3:Full Outer Joins

Subqueries

- In general, you can replace any table name with a `SELECT` statement:
  - `SELECT ... FROM (SELECT...)`
- If a query returns a **single value**, you can treat it as such:
  - `... WHERE var1 = (SELECT ...)`
- If a query returns a **single column**, you can treat it sort of like a vector:
  - `... WHERE var1 IN (SELECT...)`

SQL Conceptual Order of Operations

1. **FROM** + **JOIN**: first the product of all tables is formed.
2. **WHERE**: the where clause is used to filter rows that do not satisfy the search condition
3. **GROUP BY** + (**COUNT**, **SUM**, etc.): the rows are grouped using the columns in the group by clause and the aggregation functions are applied on the grouping
4. **HAVING**: Like the **WHERE** clause, but can be applied after aggregation
5. **SELECT**: the targeted list of columns are evaluated and returned

Note: Because of this order of operations, aliases you create in the **SELECT** clause may not be available in the **WHERE** clause (SQL dialect dependent).