GENERAL ASSEMBLY

# SQL BOOTCAMP

*Mason Gallo*

*Instructor*

COURSE

# PRE-WORK

# IF YOU HAVEN'T ALREADY...

- Read the README:
  - README
- Pay particular attention to the cheatsheet and exercises

## OPENING

# GETTING STARTED

# SQL BOOTCAMP

# WHO AM I

- Data Scientist
- Open Source Contributor
- ML Researcher - Educational Technology
- Data Science Instructor @ GA

Washington University in St. Louis

HAVAS helia. creative with data

Georgia Tech

GA GENERAL ASSEMBLY

HAVAS WORLDWIDE

Omnicom Media Group OMD phd

Duke Clinical Research Institute
From Thought Leadership to Clinical Practice

# MY PHILOSOPHY

- If you're not sure, please ask!
- If you're still stuck, we'll revisit together
- Be considerate of your fellow students
- Participate!!!
- NO ONE knows everything
- Anything worth knowing is hard
- Breaks

# LEARNING OBJECTIVES

- Learn enough SQL to effectively query data from a single table
- Learn enough SQL to effectively query data from multiple tables
- Understand where to go when you encounter new SQL problems
- Build a roadmap for your career next-steps with SQL

# AGENDA

- Installation
- Databases 101
- Basic queries
- Aggregation
- Joins
- Dealing with data
- Next steps for your career

# INSTALLATION

# DOWNLOAD CLASS MATERIALS

- All materials are available*** HERE

Click “Clone or Download” -> “Download Zip”

Slides are available in the main directory with filename sql_bootcamp.pdf

***If you know git, you can always clone the repo

# INSTALLATION

- If you're on Windows, go HERE
- If you're on OSX, go HERE

If for some reason you're not using the recommended installation, please raise your hand!

# GET ACQUAINTED

- Take a few minutes to get acquainted with the client for your OS
- Find the area to enter “queries”
- Type into the query area:

select * from db

- Find the “run” or “execute” button and run the query
- What’s the output?

SQL BOOTCAMP

# DATABASES 101

# YOUR THOUGHTS

**How do you store your data? At work? Personal?**

# MY THOUGHTS

**Storing social performance for many brands over several years**

# WHAT ARE THE OPTIONS

- Storing in a raw text file

- Storing in excel or google sheets

# LIMITATIONS

- Slow and inefficient
- Hard to share with coworkers
- Difficult to use with other applications
- Doesn’t scale (millions? billions?)

# WHAT DO WE WANT

- A methodology that can be used by many people and machines
- Scalability
- Speed
- Safety

# WHAT DO WE WANT

Databases are a **structured** data source optimized for efficient **retrieval** and **storage**

# WHAT DO WE WANT

Databases are a **structured** data source optimized for efficient **retrieval** and **storage**

**Structured:** we must pre-define the structure of our data

# WHAT DO WE WANT

Databases are a **structured** data source optimized for efficient **retrieval** and **storage**

**Structured:** we must pre-define the structure of our data

**Retrieval:** we must have a systematic way to retrieve the structured data

# WHAT DO WE WANT

Databases are a **structured** data source optimized for efficient **retrieval** and **storage**

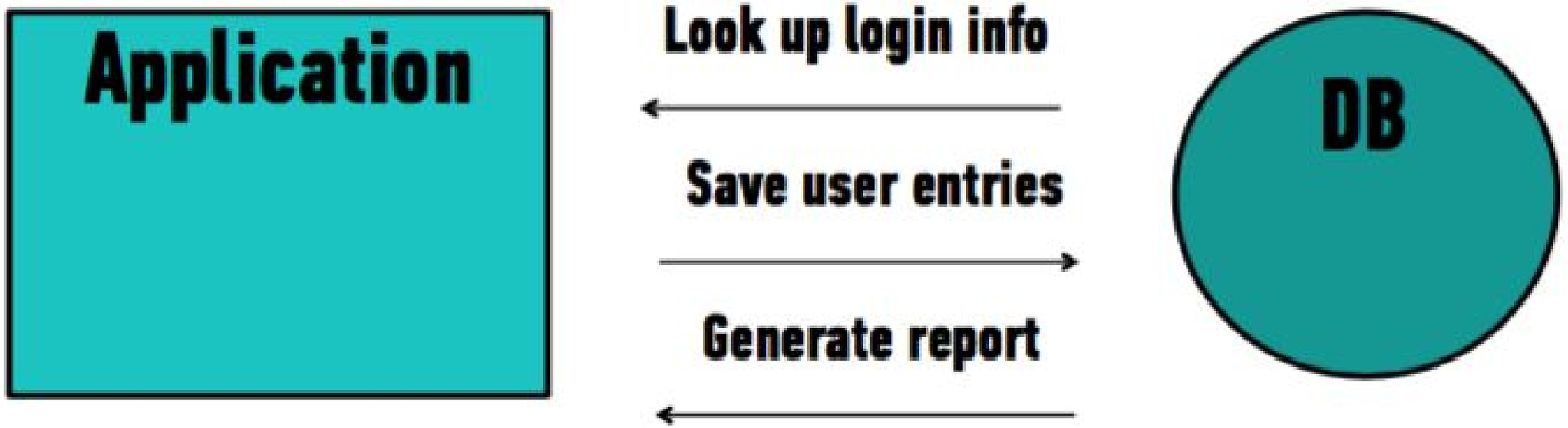
**Structured:** we must pre-define the structure of our data

**Retrieval:** we must have a systematic way to retrieve the structured data

**Storage:** we must have the ability to read and save data

# YEAH YEAH YEAH, SO WHERE DOES SQL FIT IN?

SQL is the language used to perform these actions!



# WHAT IS SQL?

SQL (Structured Query Language) is a query language designed to **E**xtract, **T**ransform, **L**oad data in relational databases

# WHO USES SQL?

Web Developers

Data Analysts

Data Scientists

Product Managers

Statisticians

System Administrators

Backend Developers

...And many more

# THE TERMS NO ONE EXPLAINS

- Database: think of as a directory or folder of tables
- Table: a collection of related structured data, usually as rows / columns
- Record: SQL lingo for “row”
- Field: SQL lingo for “column”
- Localhost: this computer (ie the computer you’re using)
- Root: a user with FULL permission to do virtually anything
- UTF-8: capable of encoding all unicode characters

SQL BOOTCAMP

# BASIC QUERIES

# NOTE BEFORE WE GET STARTED

- Capitalization doesn’t matter (SQL isn’t case-sensitive)
- SQL ignores whitespace (but don’t go crazy)
- Be OCD about your punctuation (it matters!)

# CREATE YOUR FIRST DATABASE

```
CREATE DATABASE ga;

USE ga;
```

Now, use your GUI to import users.sql, selecting UTF-8 if asked

```
mysql -u root -p test ga < users.sql
```

# BASIC QUERIES

```
SELECT *
FROM users
```

*Return all rows from all columns*

# BASIC QUERIES

```
SELECT title, first_name,
last_name
FROM users
```

*Return all rows from specific columns*

# BASIC QUERIES

```
SELECT DISTINCT title,
first_name, last_name
FROM users
```

*Return unique rows from the specified columns*

# BASIC QUERIES

```
SELECT *
FROM users
WHERE <condition>
```

*Return all columns under a specified condition*

# BASIC QUERIES

```
SELECT *
FROM users
WHERE state = “arizona”
```

# BASIC QUERIES

```
SELECT *
FROM users
WHERE state LIKE “arizona”
AND title LIKE “Miss”
```

# ***BONUS***

SELECT *

FROM users

WHERE state LIKE “arizona”

AND title LIKE binary “miss”

Binary will force case sensitivity!

# BASIC QUERIES

```
SELECT *
FROM users
WHERE state IN
(“arizona”, “florida”)
```

# BASIC QUERIES

```
SELECT *
FROM users
WHERE state IN
(“arizona”, “florida”)
```

What happens if we put a NOT in front of IN?

# BASIC QUERIES

```
SELECT *
FROM users
WHERE zip = 10007
```

Notice the use of = here instead of LIKE. Any ideas?

We can also use >, <, !=, >=, <=

# BASIC QUERIES

```
SELECT *
FROM users
WHERE street LIKE “%rd”
```

% matches anything

_ matches a single character

# BASIC QUERIES

```
SELECT *
FROM users
WHERE street LIKE “%rd”
ORDER BY first_name
```

What happens if you put DESC after first_name?

# BASIC QUERIES

```
SELECT *
FROM users
WHERE street LIKE “%rd”
ORDER BY first_name
LIMIT 5
```

What does LIMIT do?

# YOUR TURN

Complete the warm up exercises here

SQL BOOTCAMP

# AGGREGATION

# YOUR TURN

What if we want to compute overall statistics about the data?

Think: we want counts, averages, etc

# AGGREGATION

```
SELECT *
FROM users
```

*Return all rows from columns*

# AGGREGATION

```
SELECT count(*)
FROM users
```

*Return the count of all rows from all columns*

# AGGREGATION

```
SELECT distinct first_name
FROM users
```

*Return the unique first names*

# AGGREGATION

```
SELECT count(distinct first_name)
FROM users
```

*Return the count of unique first names*

# AGGREGATION

```
SELECT avg(zip)
FROM users
```

*Return the average zip code (yes, I know this doesn't make sense)*

# AGGREGATION

```
SELECT avg(zip) as “nonsense”
FROM users
```

*Rename our avg zip to “nonsense”*

# AGGREGATION

```
SELECT gender, avg(zip) as
“nonsense”
FROM users
GROUP BY gender
```

*What if we want to know the avg zip by gender?*

*We can use sum, avg, min, max, count*

# AGGREGATION

```
SELECT state, gender, avg(zip) as
“nonsense”
FROM users
GROUP BY state, gender
```

*What happened here?*

# AGGREGATION

```
SELECT state, gender, avg(zip) as
“nonsense”
FROM users
GROUP BY state, gender
ORDER BY 1
```

*What happened here?*

# AGGREGATION

```
SELECT state, gender, avg(zip) as
“nonsense”
FROM users
GROUP BY state, gender
ORDER BY 1
```

*What happened here?*
*Shorthand?*

# AGGREGATION

```
SELECT state, gender, avg(zip) as
nonsense
FROM users
GROUP BY state, gender
HAVING nonsense > 50000
```

*What happened here? What happens if we use WHERE instead?*

# AGGREGATION

```
SELECT <columns>
FROM <table>
WHERE <condition>
GROUP BY <columns>
HAVING <condition on aggregates>
ORDER BY <columns>
LIMIT <number>
```

*General SQL structure*
*(putting it all together)*

# YOUR TURN

Complete the aggregation exercises here

SQL BOOTCAMP

# JOINS

# JOINS

What if we want to deal with multiple tables?

Think: we want to capture information from 2 or more tables simultaneously

Import flights.sql

# JOINS

A **relational database** is organized in the following manner:

- A database has **tables** which represent individual entities or objects
- Tables have a predefined **schema** - rules that tell it what columns exist and what they look like

# JOINS

## A **relational database** is organized in the following manner:

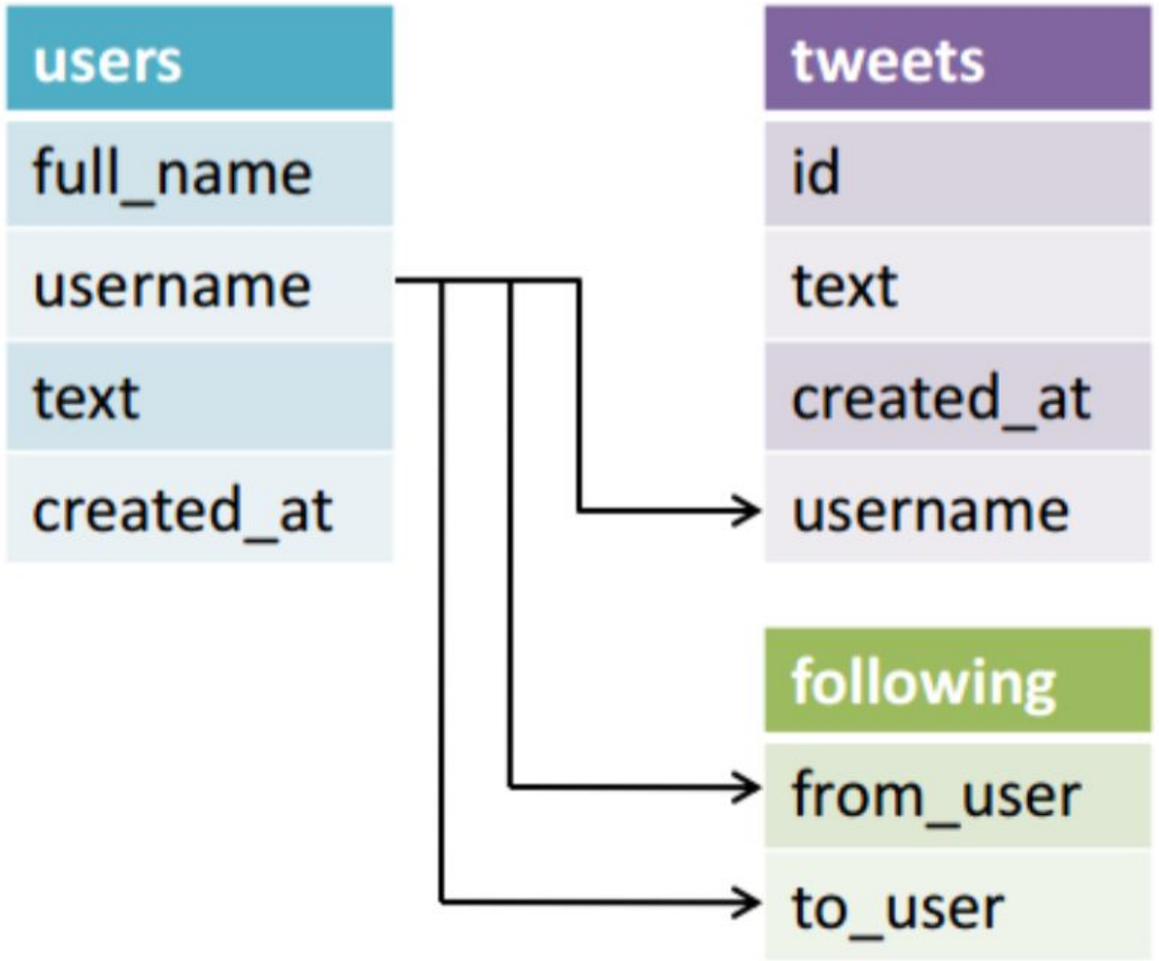
- table

| id | first_name | last_name | date_of_birth |
|---|---|---|---|
| 312 | Joe | Smith | 1980-12-24 |
| 1532 | Michelle | Anderson | 1973-03-12 |

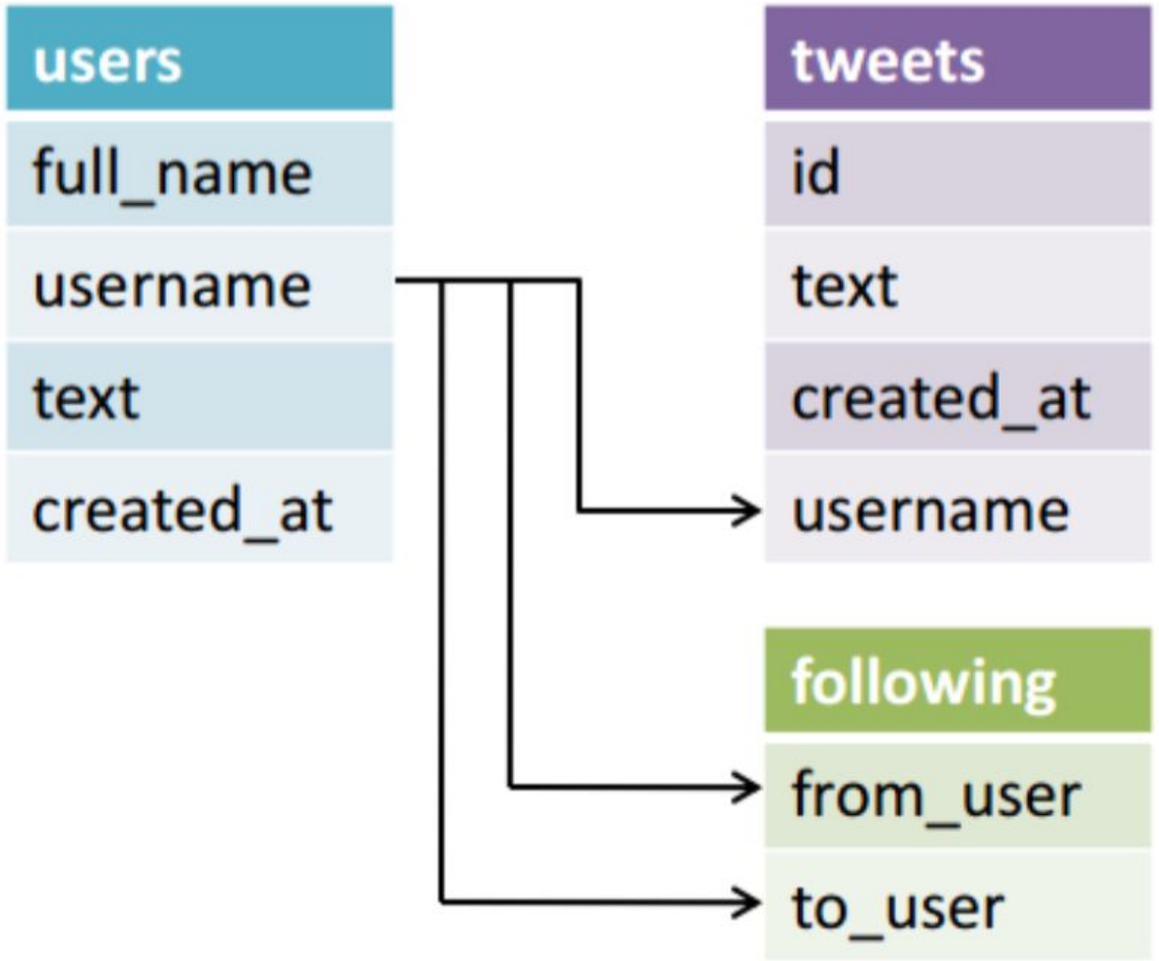
- schema

```
id              bigint
first_name      char(36)
last_name       char(36)
date_of_birth   timestamp
```

# JOINS


users
full_name
username
text
created_at
tweets
id
text
created_at
username
following
from_user
to_user

# JOINS


users
full_name
username
text
created_at
tweets
id
text
created_at
username
following
from_user
to_user

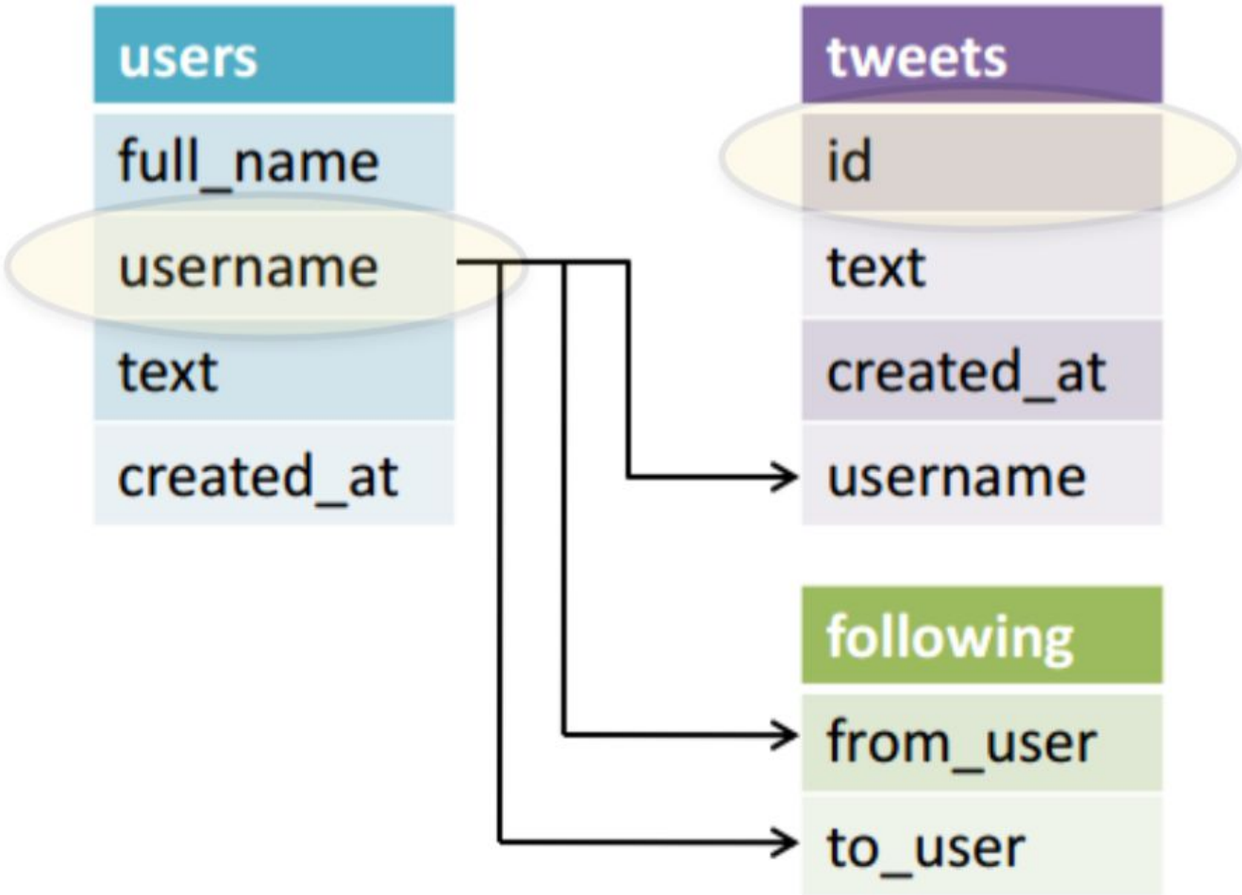
# JOINS

Each table should have a PRIMARY KEY

Think: a unique identifier for each row

Ex: SSN

# JOINS


users
full_name
username
text
created_at
tweets
id
text
created_at
username
following
from_user
to_user

# JOINS

Each table should have a PRIMARY KEY

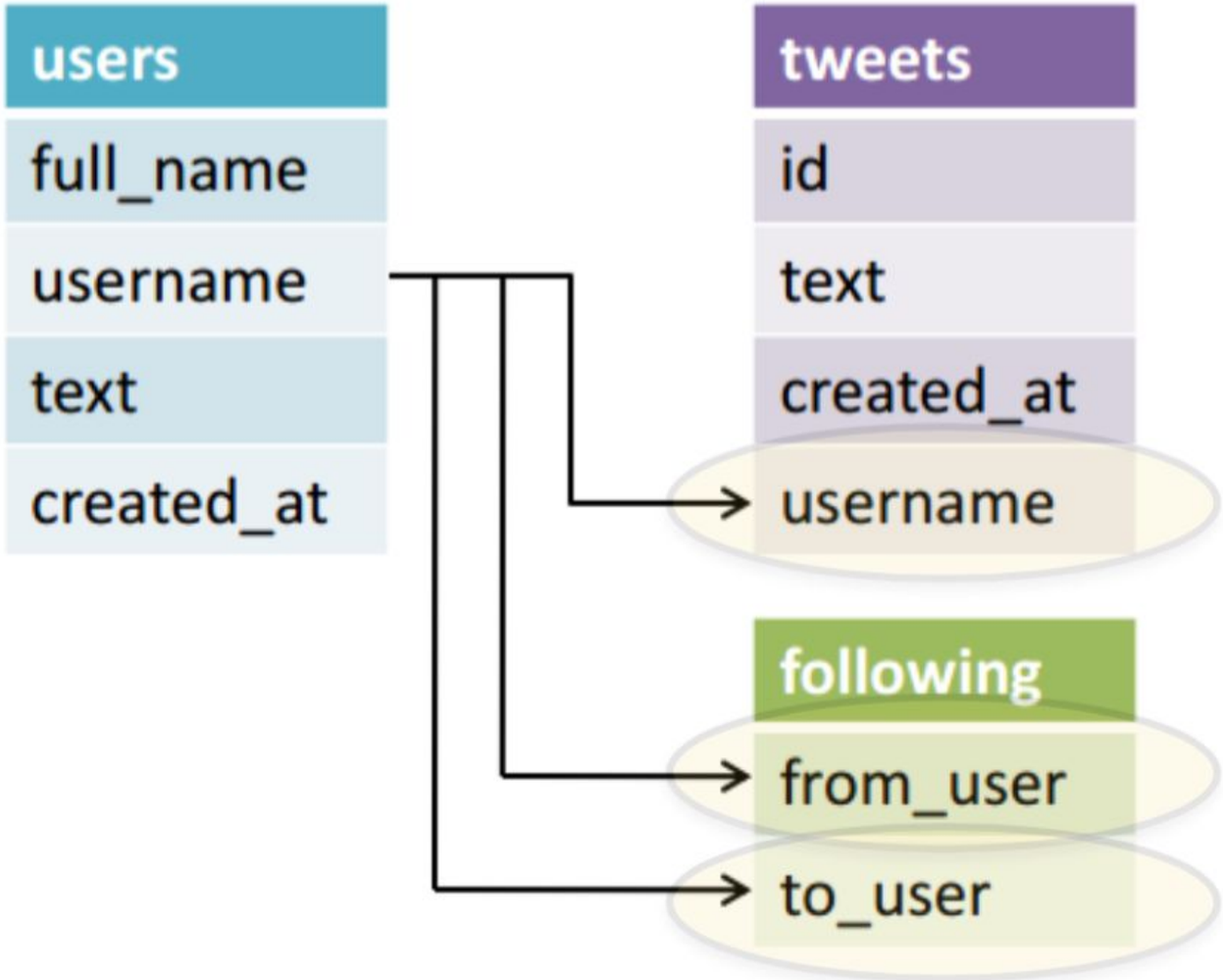
Think: a unique identifier for each row

Ex: SSN

Additionally, each table may have a FOREIGN KEY

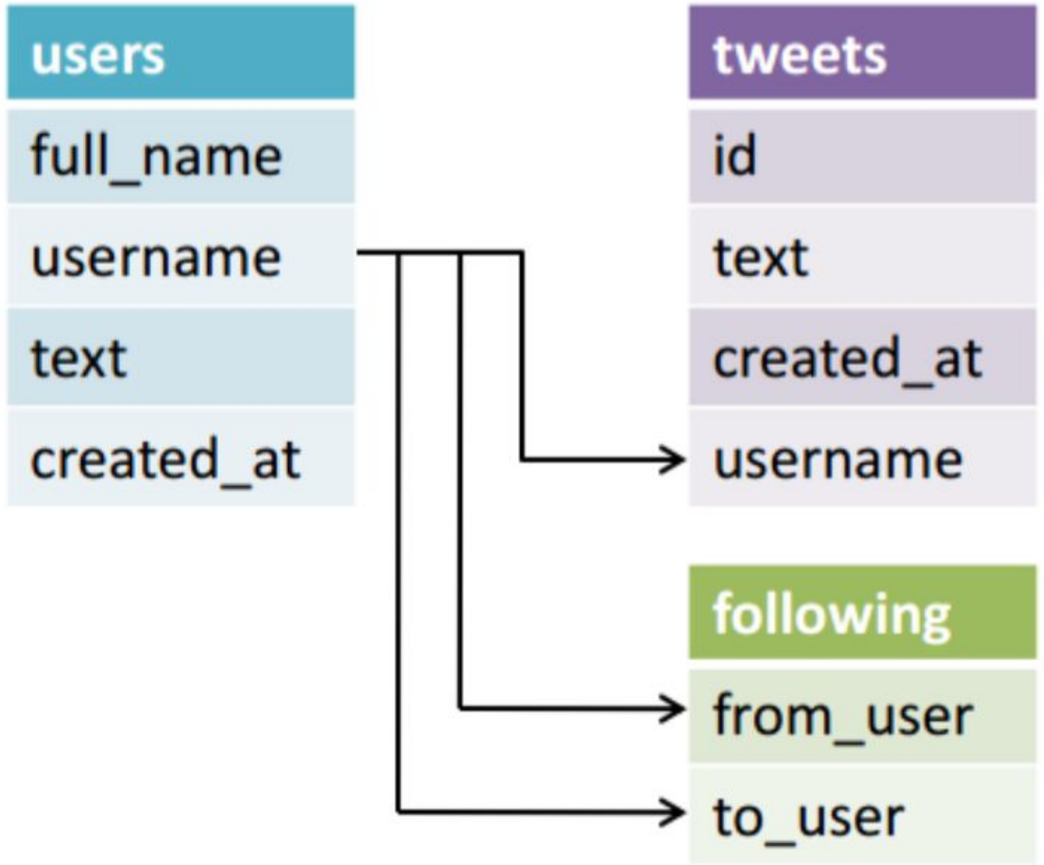
Think: an ID that links one table to another

# JOINS


users
full_name
username
text
created_at
tweets
id
text
created_at
username
following
from_user
to_user

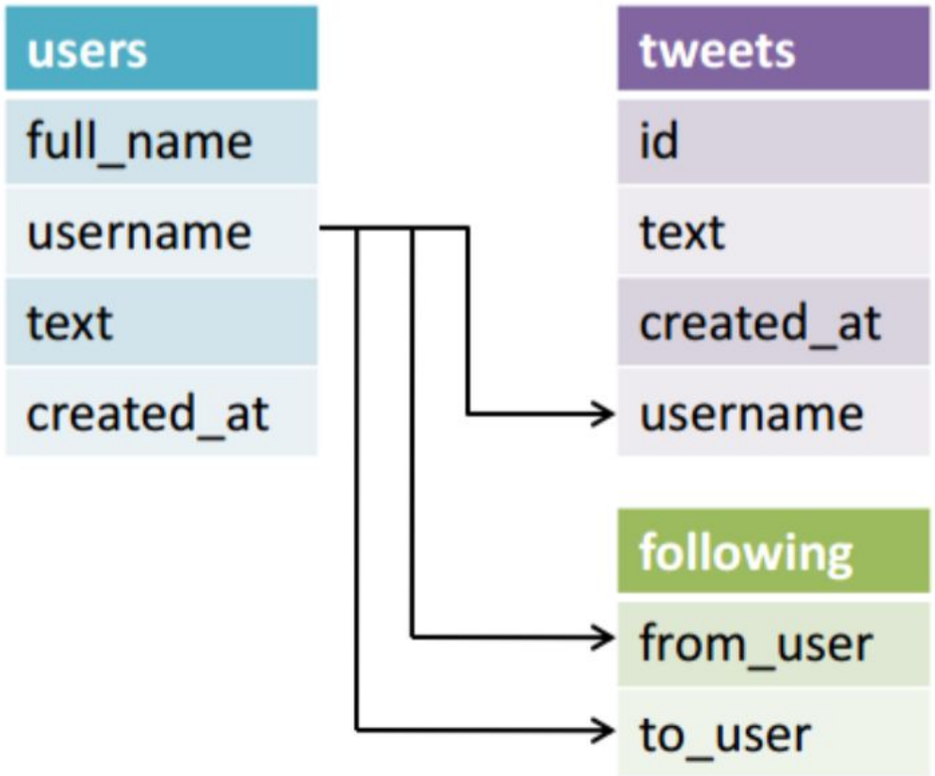
# JOINS

*Create a table with all the users' full names and their tweets*



# JOINS

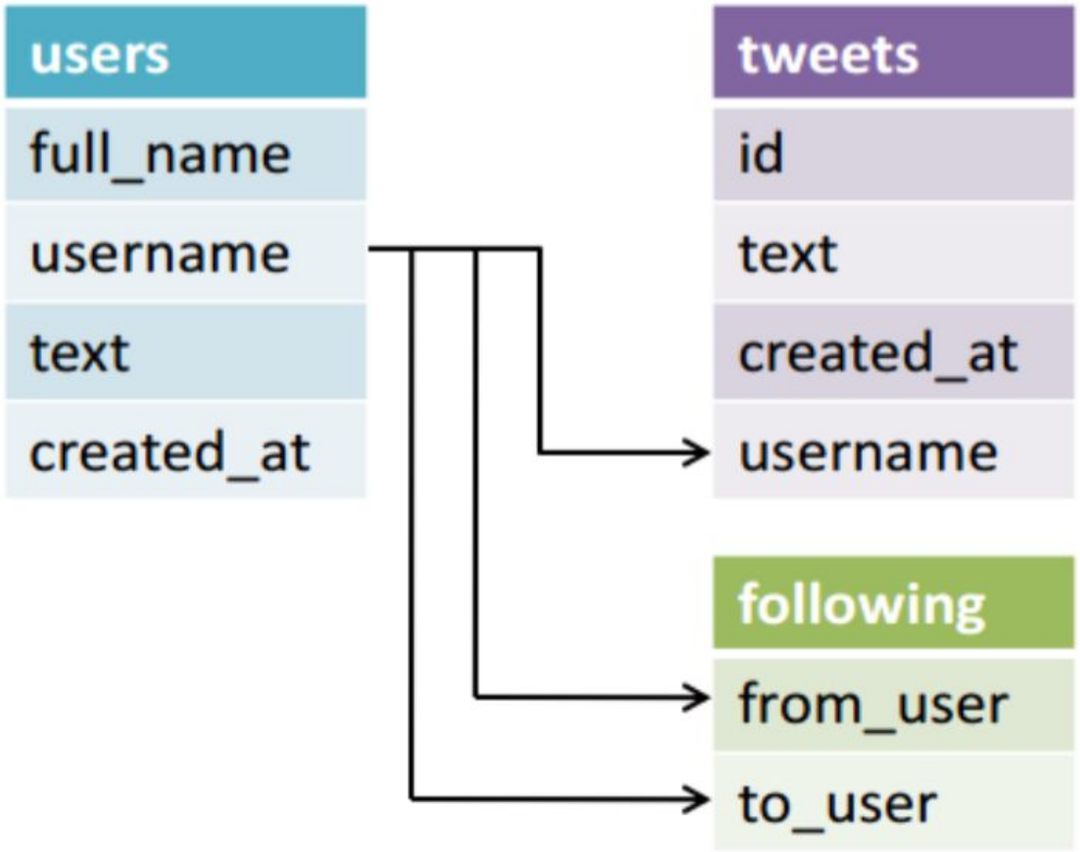
*Create a table with all the users' full names and their tweets*




| full_name | tweet |
|---|---|
| Joe Smith | Hello, world! |
| Joe Smith | Just tweetin' |
| Michelle Ng | I am eating pizza tonight |

# JOINS

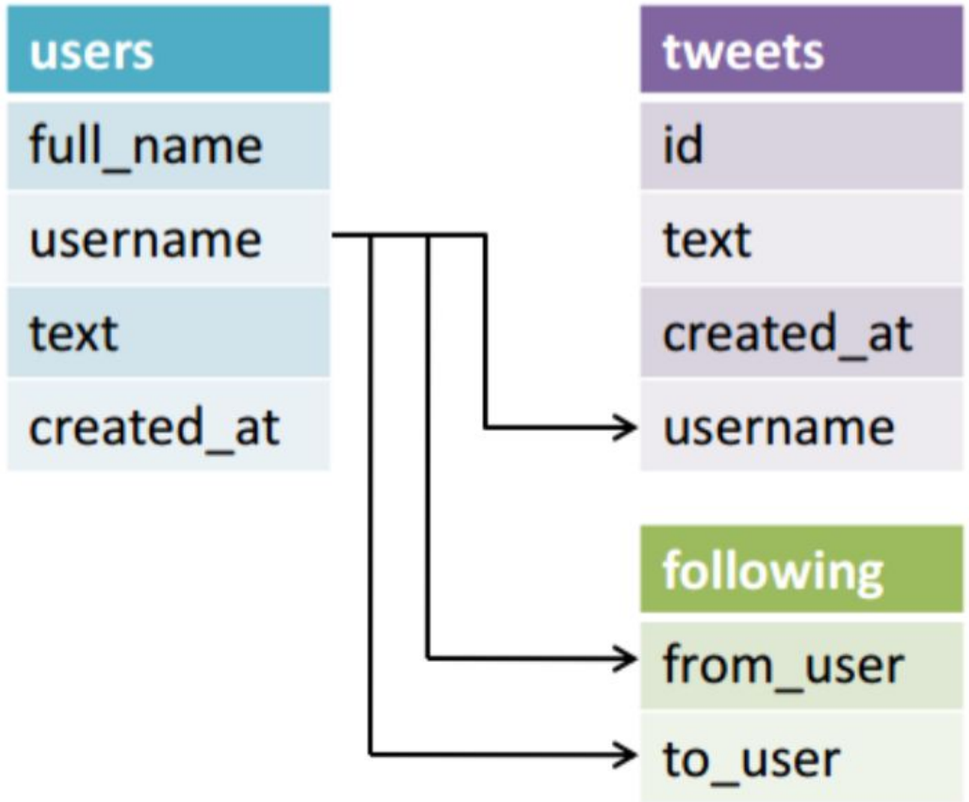
*Create a table with all the users' full names and their tweets*




```
SELECT
  users.full_name,
  tweets.text
```

# JOINS

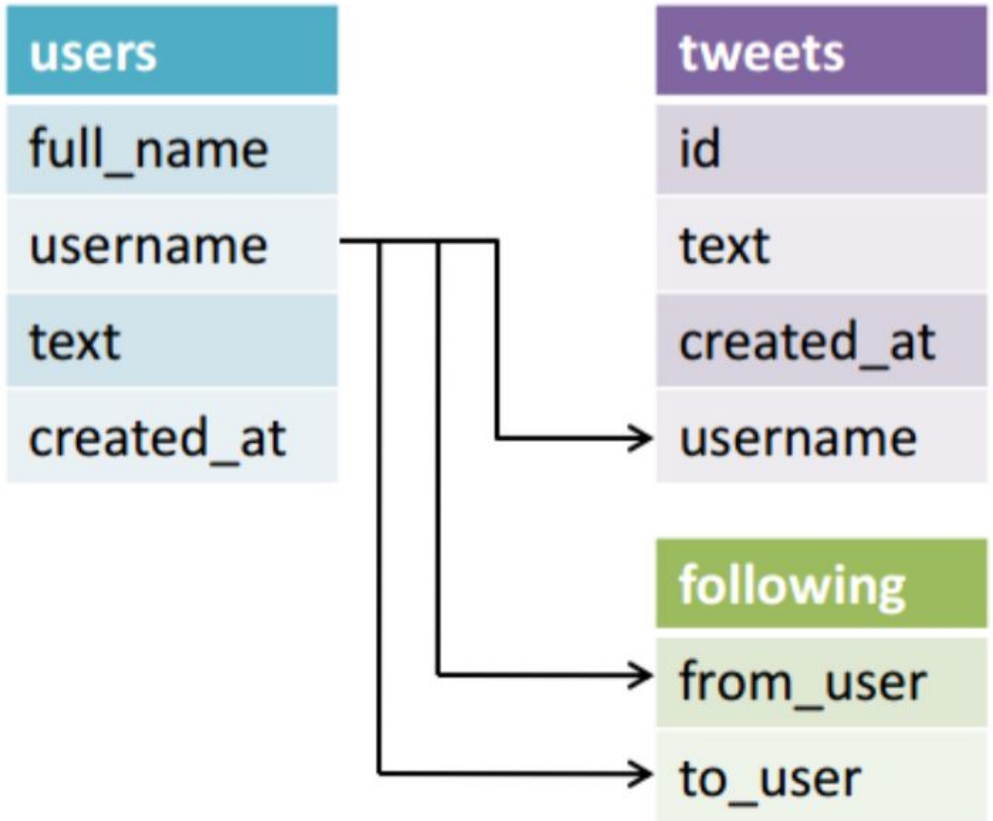
*Create a table with all the users' full names and their tweets*




```
SELECT
  users.full_name,
  tweets.text
FROM users
JOIN tweets
ON users.username = tweets.username
```

# JOINS

*Create a table with all the users' full names and their tweets*




*Will users who never tweeted appear in the list?*

# JOINS

**JOIN** *will only include entries that occur in both tables.*

```
SELECT
  full_name,
  text
FROM users
JOIN tweets
ON users.username = tweets.username
```

| full_name | tweet |
|---|---|
| Joe Smith | Hello, world! |
| Joe Smith | Just tweetin' |
| Michelle Ng | I am eating pizza tonight |

# JOINS

**LEFT JOIN** *will always include all entries from the left table, even if there are no matches in the other table.*

```
SELECT
  full_name,
  text
FROM users
LEFT JOIN tweets
ON users.username = tweets.username
```

| full_name | tweet |
|---|---|
| Joe Smith | Hello, world! |
| Joe Smith | Just tweetin' |
| Michelle Ng | I am eating pizza tonight |
| Jim Rogers | |

# JOINS

***FULL OUTER JOIN** will always include all entries from <u>both</u> tables, even if there are no matches in the other table.*

```
SELECT
  full_name,
  text
FROM users
FULL OUTER JOIN tweets
ON users.username = tweets.username
```

| full_name | tweet |
|---|---|
| Joe Smith | Hello, world! |
| Joe Smith | Just tweetin' |
| Michelle Ng | I am eating pizza tonight |
| Jim Rogers | |
| | OK, deleting my account |

# JOINS

The holes in the resulting table are called **NULL**s.

**NULL** indicates missing data.

Note that **NULL** is not the same as zero or an empty string “”, it really means that there is no data.

| full_name | tweet |
| --- | --- |
| Joe Smith | Hello, world! |
| Joe Smith | Just tweetin’ |
| Michelle Ng | I am eating pizza tonight |
| Jim Rogers | NULL |
| NULL | OK, deleting my account |

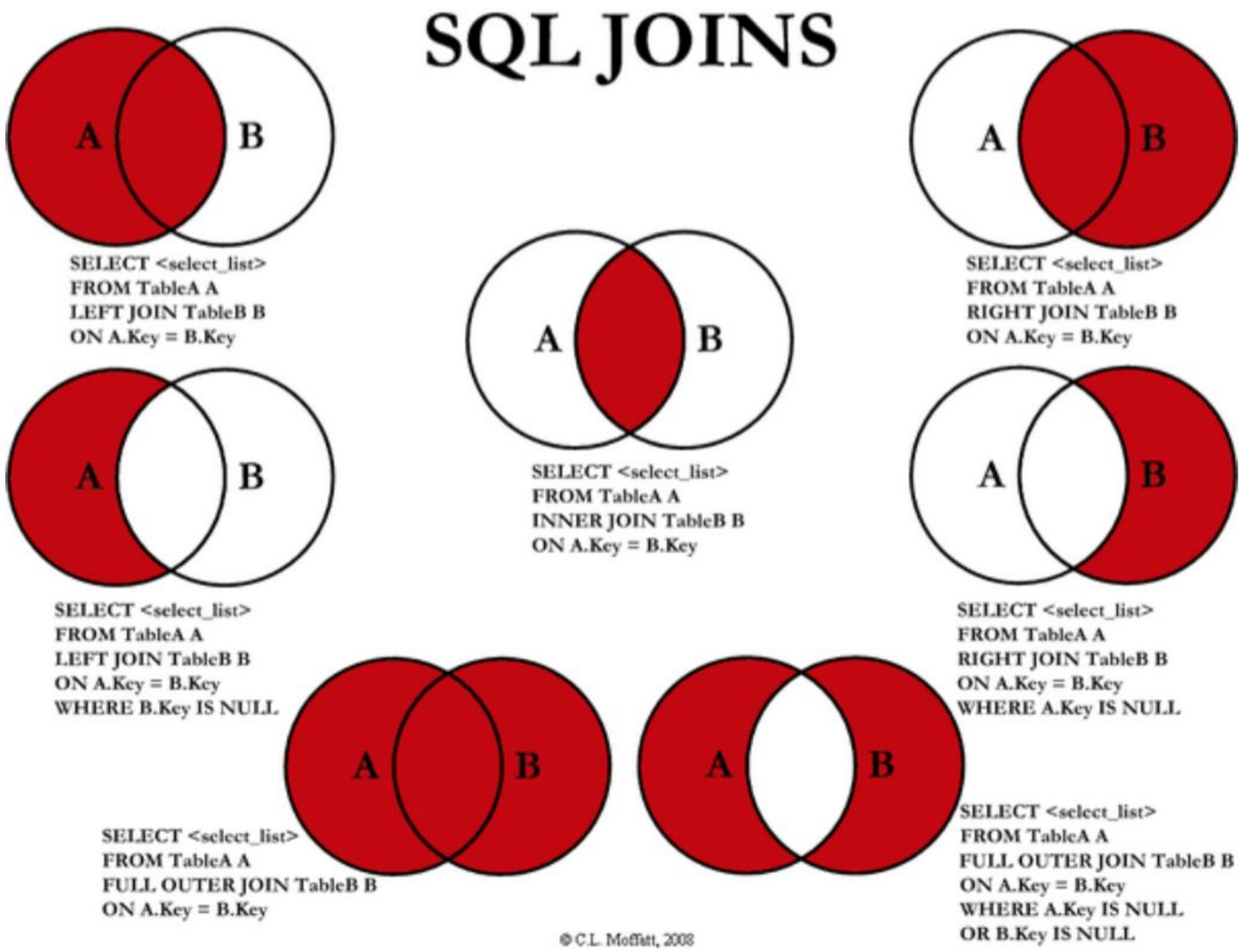
# JOINS

For example, to print a list of users without tweets, we'd write

```
SELECT full_name
FROM users
FULL OUTER JOIN tweets
ON users.username = tweets.username
WHERE tweets.text IS NULL
```

| full_name |
| --- |
| Jim Rogers |

# JOINS


SQL JOINS
A
B
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
© C.L. Moffatt, 2008

# JOINS

*General SQL structure*

```
SELECT <columns>
FROM <table>
JOIN <otherTable>
ON <table.key> = <otherTable.key>
JOIN <yetAnotherTable>
ON <otherTable.key> = <yetAnotherTable.key>
```

**NOTE**

You can combine as many **JOIN**s as you want!

# JOINS

*General SQL structure*

```
SELECT <columns>
FROM <table>
[INNER|LEFT|RIGHT|FULL OUTER] JOIN <otherTable>
ON <table.key> = <otherTable.key>
WHERE <condition>
GROUP BY <columns>
HAVING <condition>
ORDER BY <columns> [DESC|ASC]
LIMIT <number>
```

SQL BOOTCAMP

# DEALING WITH DATA

# DEALING WITH DATA

SELECT 1+1

What just happened?

# DEALING WITH DATA

SELECT 1+1

SELECT “hi”

What just happened?

# DEALING WITH DATA

SELECT first_name, last_name,

CASE WHEN last_name like “n%” THEN “Starts with n” ELSE “Doesn’t” END

FROM users

Think: if…then statement!

# DEALING WITH DATA

```
SELECT first_name, last_name,
FROM users
WHERE first_name in (
    SELECT DISTINCT first_name
    FROM presidents
)
```

This is called a subquery!

# DEALING WITH DATA

Check out the cheatsheet HERE

# NEXT STEPS

# NEXT STEPS FOR WEB DEVELOPERS

Install the LAMP Stack

You now know the M in LAMP!

Try creating a site and using SQL to store your data

# NEXT STEPS FOR DATA ANALYSTS / DATA SCIENTISTS

Get a toy dataset

Use in conjunction with a data visualization tool like Excel or Tableau

Excel has Power Query and Tableau has native SQL support!

# NEXT STEPS FOR BACKEND DEVELOPERS

Get a toy dataset and manipulate using your favorite scripting language, like Python (I’m biased!)

# OPINION

Be weary of people telling you that SQL is “too slow” or “too fast”

Make sure they’re not trying to sell you something!

# OPINION

You should not be “cleaning” your data in SQL

You should not be “formatting” your data in SQL for visualization

SQL is only for storing and retrieving raw data!

# OPINION

You now know probably 80% of the SQL you need

Many people will never need to use the remaining 20% EVER

Make sure your skills don’t atrophy!

# KEEP IN TOUCH

If you enjoyed my class, I am starting a blog:

Follow me on Twitter

If you're on Github:

Follow Me

If you're on Linkedin:

Connect with Me