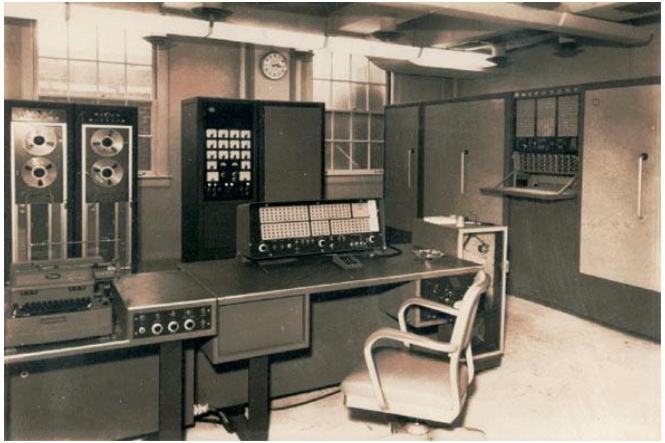
Introduction to Structured Databases I

Agenda - Schedule

1. Pandas Warm-Up
2. History of SQL
3. Introduction to SQL
4. Break
5. SQL Hands-on Lab



Database systems of the past

Agenda - Announcements

- Program Satisfaction Survey:

  https://theknowledgehouse.typeform.com/to/JyoK7IHd !!!!!!!!!!!

- Week 9 Pre-Class Quiz due 5/9 (*2 attempts*)
- TLAB #3 due 5/14
  - Early grade due date: 5/7
  - Extension due date: 5/13

Agenda - Goals

- Create simple tables using SQL and define their structure using basic DDL operations (CREATE, ALTER, RENAME, DROP)
- Retrieve specific rows or columns from a table using SELECT and WHERE
- Use aggregate functions (SUM, COUNT, AVG, MIN, MAX) to summarize data
- Group results using GROUP BY and filter grouped data using HAVING
- Perform JOINs to combine data across tables

Pandas LeetCode Question

620. Not Boring Movies

Solved

Easy | Topics | Companies

SQL Schema > Pandas Schema >

Table: `Cinema`

```
+----------------+----------+
| Column Name    | Type     |
+----------------+----------+
| id             | int      |
| movie          | varchar  |
| description    | varchar  |
| rating         | float    |
+----------------+----------+
id is the primary key (column with unique values) for this table.
Each row contains information about the name of a movie, its genre, and its
rating.
rating is a 2 decimal places float in the range [0, 10]
```

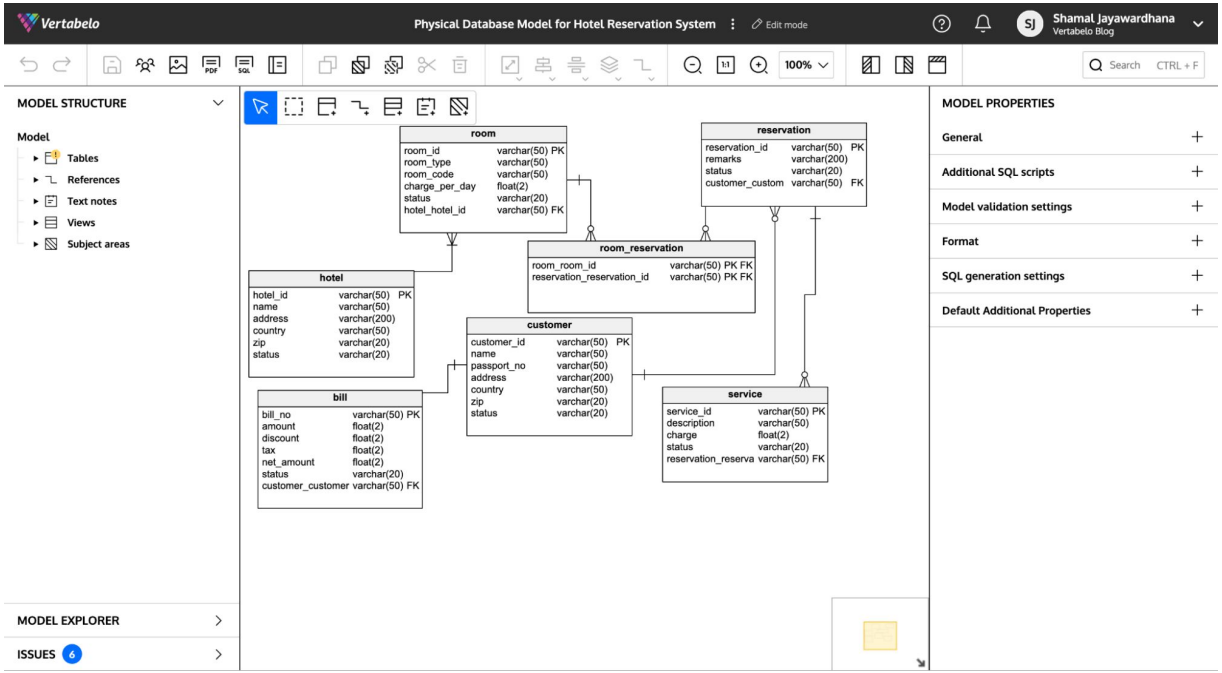
Write a solution to report the movies with an odd-numbered ID and a description that is not `"boring"`.

Take 10 minutes to complete “Not Boring Movies”:
https://leetcode.com/problems/not-boring-movies/description/

Necessity of SQL

Customer ID	Age	Gender	Item Purchased	Purchase Amou	Location	Size	Color	Season	Review Rating	Shipping Type	Promo Code Us
3475		Male	Jacket	30.9	Maine	M	Burnt orange	Fall	4	Standard	No
3698	21	Female	Backpack	31.59		L	Turquoise	Winter	2	Express	No
2756	31	Male	Leggings	24.23	Nevada	M	Terra cotta	Winter	4	Standard	No
3340		Male	Pajamas	33.92	Nebraska	M	Black	Winter	NA	Standard	No
3391	38	Male	Sunglasses	36.55	Oregon	S	Aubergine	Summer	NA	Standard	No
2599	26	Male	Leggings	23.6	Nevada	XL	Brown	Winter	NA	Standard	No
2591	43	Male	Dress	34.08	California	M	Terra cotta	Fall	5	Standard	No
3650	29	Male	Shorts	23.8	Minnesota	M	Lavender	Summer	2	Express	No
3353	25	Female	Jacket	31.6	Washington	M	Mauve	Fall	4	Standard	No
2477	39	Female	Shorts	32.37	Colorado	M	Fuchsia	Summer	NA	Standard	No
2075	45	Female	Jacket	35.55	Florida	M	Brown	Winter	NA	Standard	No
3278	23	Male	Backpack	34.44	Texas	M	Brown	Winter	NA	Standard	No
3341	27	Female	Handbag	29.43	Virginia	XL	Black	Summer	NA	Standard	No

We've learned so far how to manipulate data from a **local CSV** that is saved to your computer. However (*especially in the context of a business*), you will not only be interacting with locally saved datasets, but you will also manipulate datasets from a…



…database! We use databases to store business-related data (*purchases*, *customers*, *orders*, *etc*) on a remote infrastructure. As a data analyst/engineer we should understand the language and systems that we use to interface with databases. Before we get into specifics, let's first learn **why we need SQL.**

Necessity of SQL

Throughout your journey in learning tech, you will be inundated with advice to learn framework ***XYZ*** or ***ABC***.

Learning a new **framework** is **expensive** (*opportunity-cost*, *time*, *etc*), therefore you must always have a **good reason** for learning a technology.

Let's figure out why **SQL is a good idea**.

First Database Models (1960's)

Problem:

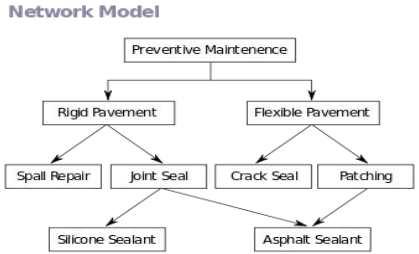
- Companies need to store data on **sales**, **services**, **employees**, etc
- Developers stored digital data on **systems of unrelated files** with no standard
- **Error-prone**, **slow**, **poor data integrity**, **difficult to maintain** systems
- Machines were **expensive & limited on memory**

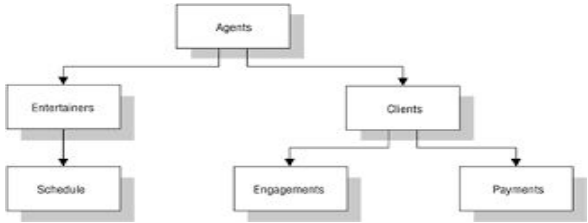
First Database Models (1960's)

Why might this be a problem? What's the risk to everyone having their own standard?

Solution:

- Developers made their own "**intermediate layers**" between applications & data aka "database models" (logical structures to explain how data is represented)

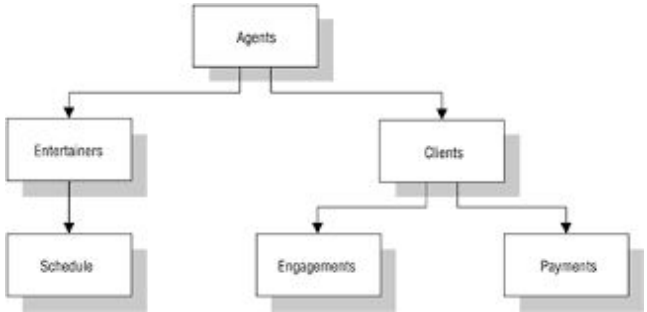




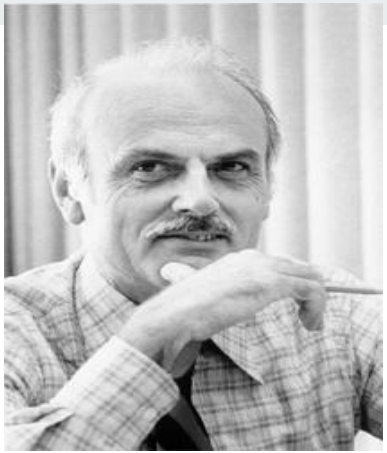
Relational Database Models (1970's)

Problem:

- Previous db logical models were **complex to implement & maintain**
- Programmers had to **understand entire structure** to have efficient (or working) queries



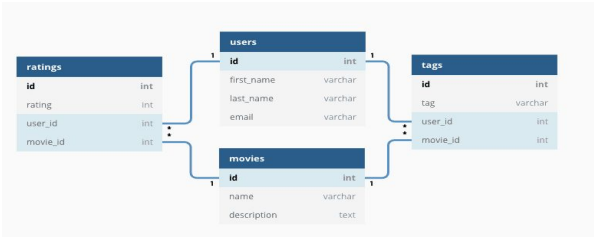
Relational Database Models (1970's)



Solution:

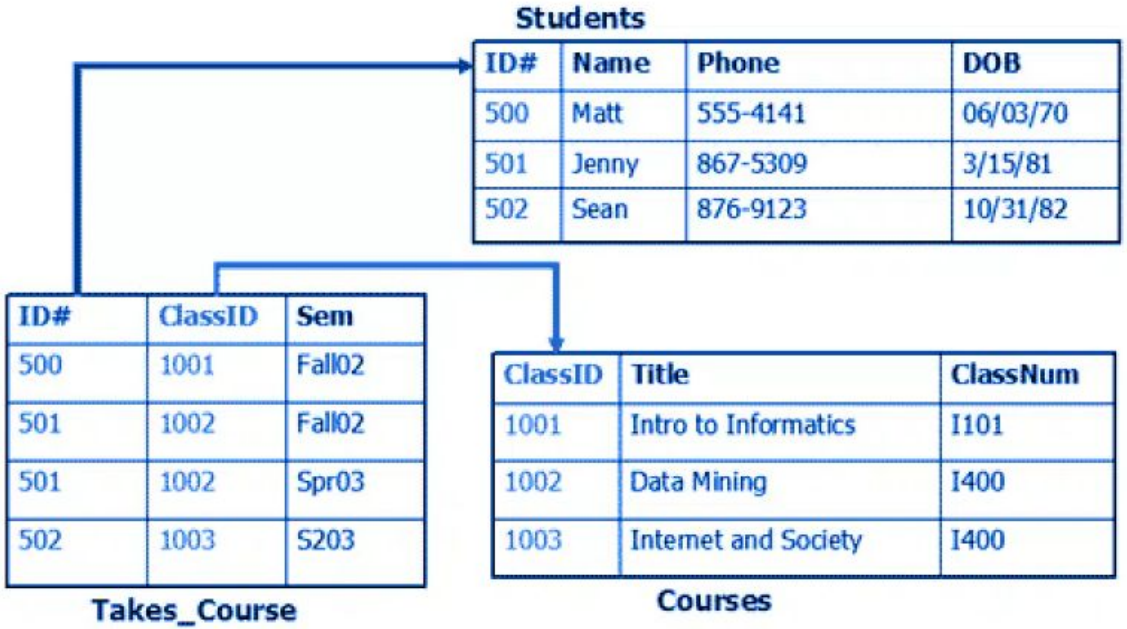
1970: Mathematician from IBM, Edgar F. Codd, published a paper describing a **relational database model**

- Access data via **structured query language**
- Programmers do not worry about **physical storage**
- Implements **many kind of relationships**



One of the most important progressions in software

Relations have “primary keys” which can be “foreign keys” in other relations



Students

ID#	Name	Phone	DOB
500	Matt	555-4141	06/03/70
501	Jenny	867-5309	3/15/81
502	Sean	876-9123	10/31/82

Takes_Course

ID#	ClassID	Sem
500	1001	Fall02
501	1002	Fall02
501	1002	Spr03
502	1003	S203

Courses

ClassID	Title	ClassNum
1001	Intro to Informatics	I101
1002	Data Mining	I400
1003	Internet and Society	I400

Each “table” is called a “relation.”

Hence the name relational database management system

This allows us to store and relate different attributes in a highly efficient and understandable fashion.

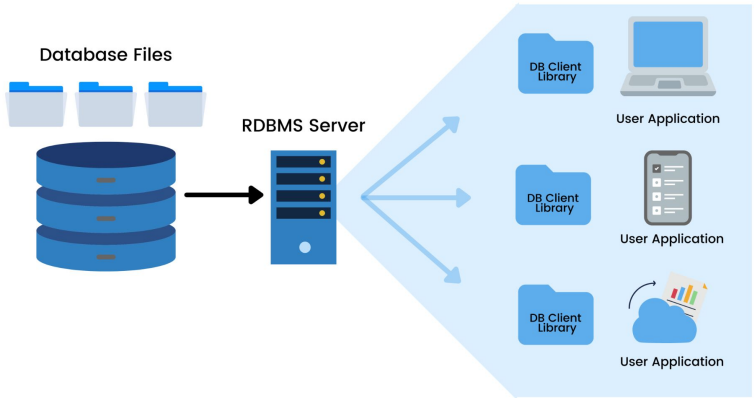
RedSwitches

Relational Database Management Systems in a nutshell

Necessity of SQL

Let's make a big deal as to why this is such a good idea:

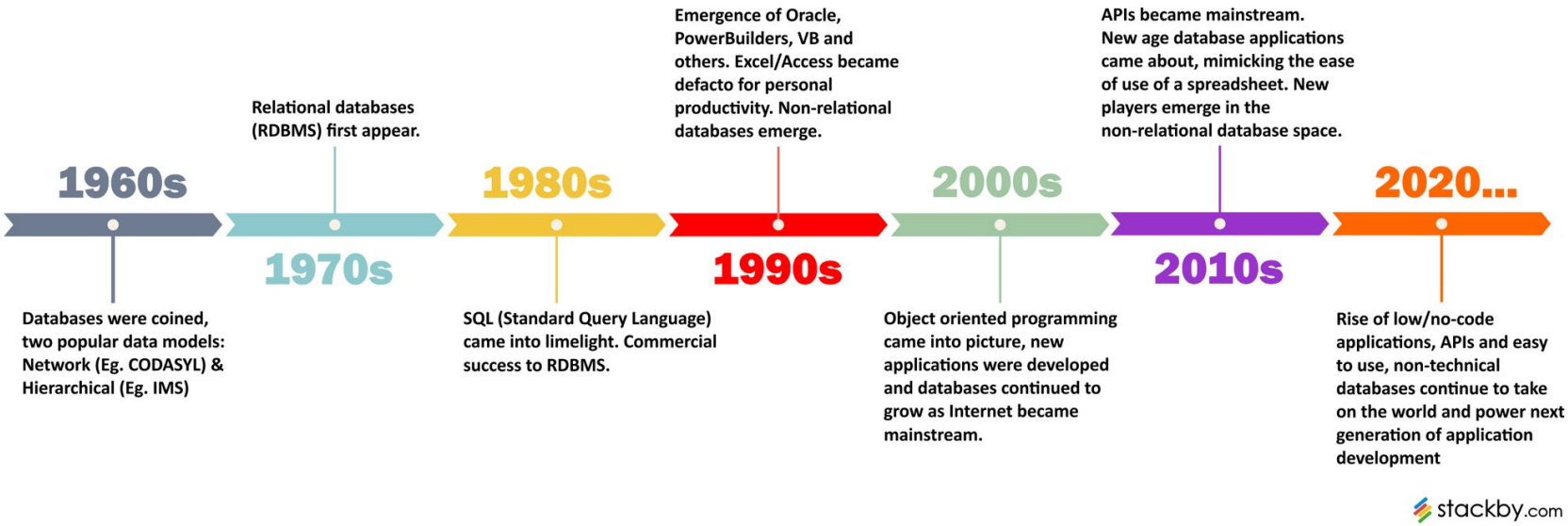
- **ONE language standard** to interact with data
- **HUMAN-LIKE** language to understand data querying
- **DIFFERENT ways** to express the relationships between data
- **BUILT-IN** query optimization to make the fastest query possible
- **and a lot more...**



There are many different types or “flavors” of SQL out there, but they all abide by the **ANSI standard.**

We will focus on **PostgreSQL**

History of Databases (1960-2020)



Many new abstractions were implemented after **Codd** (*aka the CoddFather*), **HOWEVER**, SQL has survived all of these advances.

SQL is **such a good idea** that it will most likely be in use for a **long long time**

PARTING THOUGHTS

The demarcation lines of DBMS categories will continue to blur over time as specialized systems expand the scope of their domains.

→ Every NoSQL DBMS (except for Redis) now supports SQL

The relational model and declarative query languages promote better data engineering.

Keep in mind! We don't want you to be dogmatic (set in your technological ways forever), but we just want to point out that **SQL is a good idea**

For the past 50 years, many API's have been made to "replace" SQL. **All of them have been absorbed into SQL.**

CMU Dr.Andy Pavlo: https://youtu.be/LWS8LEQAUVc?t=4380

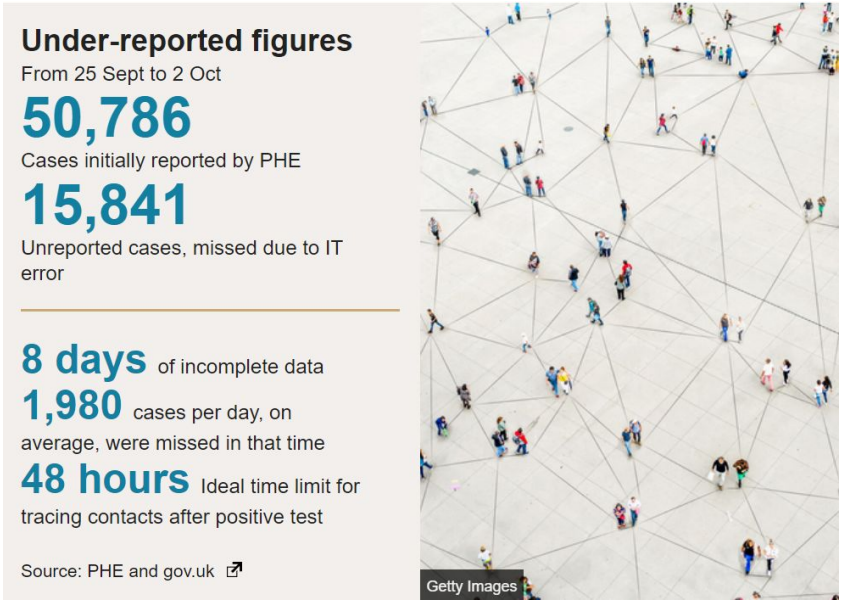
Necessity of SQL

Now that we understand the permanence of SQL, let's get into specifics and define what a **database** is: **A central system for storing & querying data.**

We want databases to:

- ***Store massive amounts of data***
- ***Allow access via a query language***
- ***Allow durability even during power failures***
- ***Allow multiple users to interact with data***

This is the de-facto system of storing data and with reason! What happens when we refuse to acknowledge good data ideas?



During the height of COVID, the UK health department used excel to store data.

This version of excel they were using had a 65,000 row limit. What do you think happened if our tables contained more than 65,000 rows?
https://www.bbc.com/news/technology-54423988

DDL

SQL - Logical Groups

There are 3 main logical groups of SQL that split up functionality.

Data Definition Language (DDL)

Data Control Language (DCL)

Data Modification Language (DML)

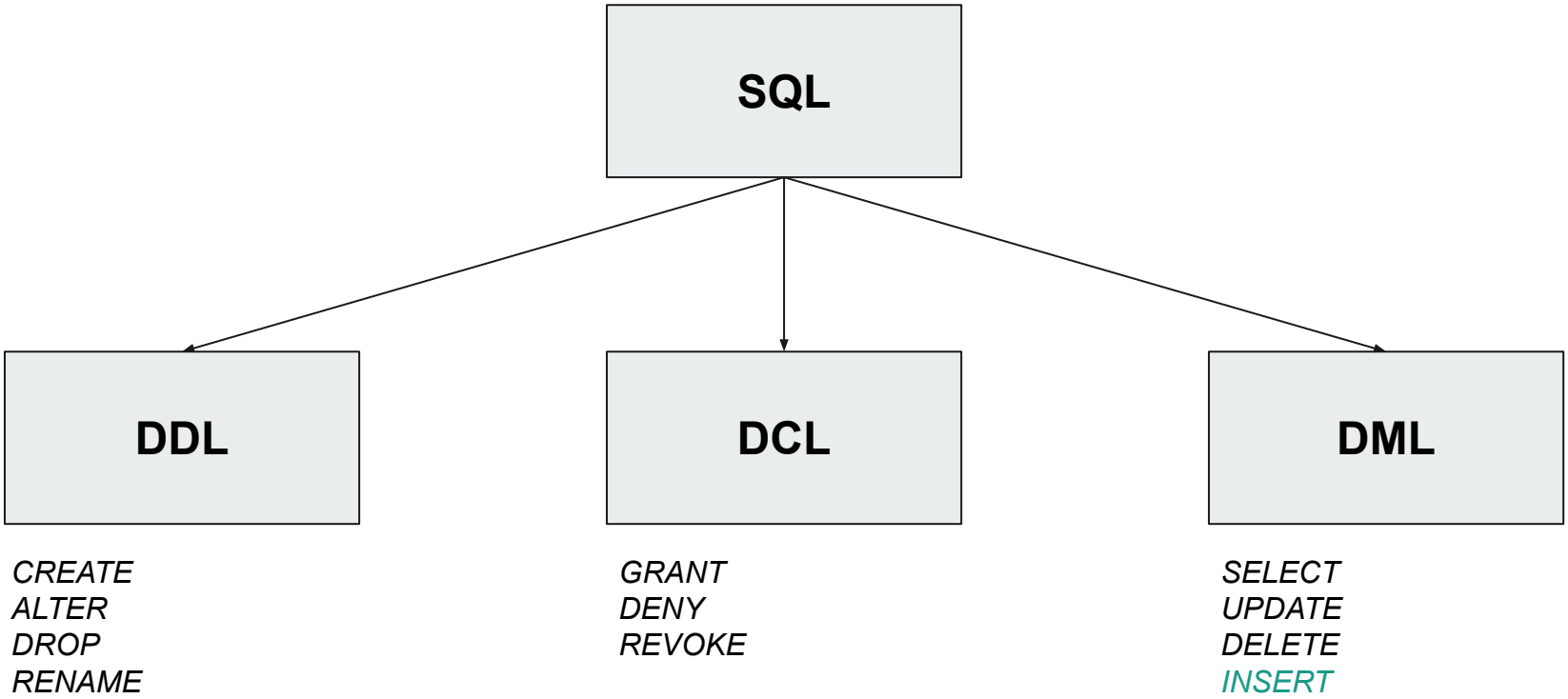
Can you guess what each group of SQL is responsible for?

SQL - Logical Groups

Data Definition Language (DDL): Used to create, modify, & remove tables & db objects.

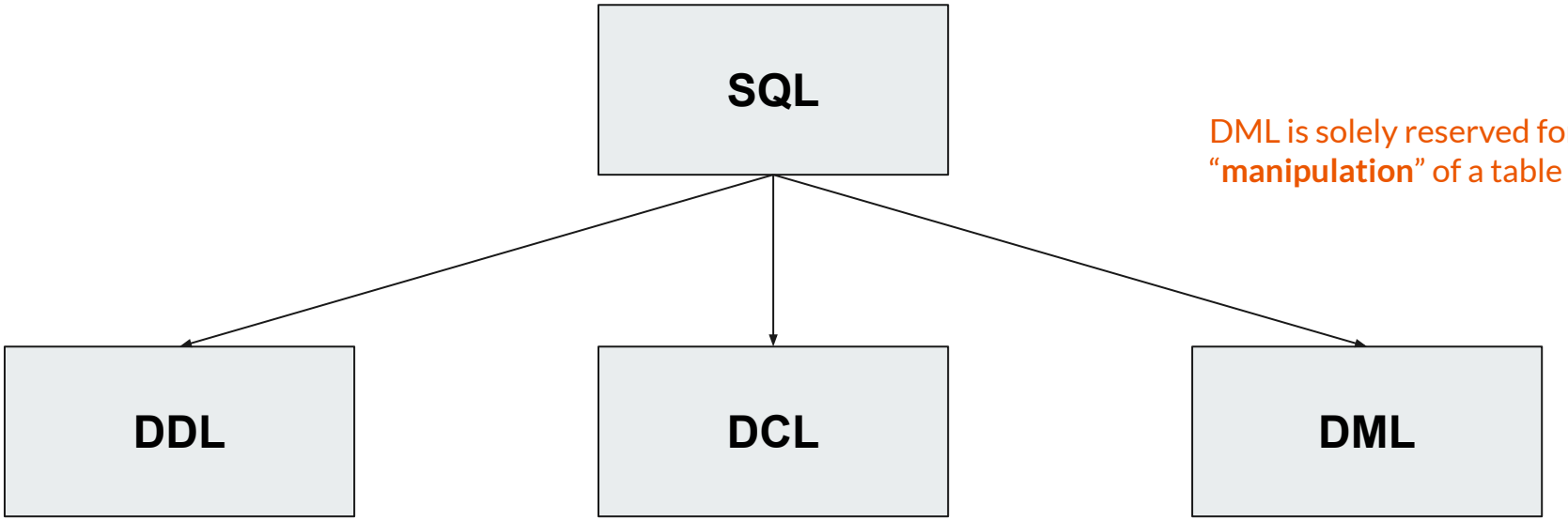
Data Control Language (DCL): Used to manage access to db objects by giving or revoking permissions to users or groups.

Data Modification Language (DML): Used to manipulate, delete, & query rows in tables.



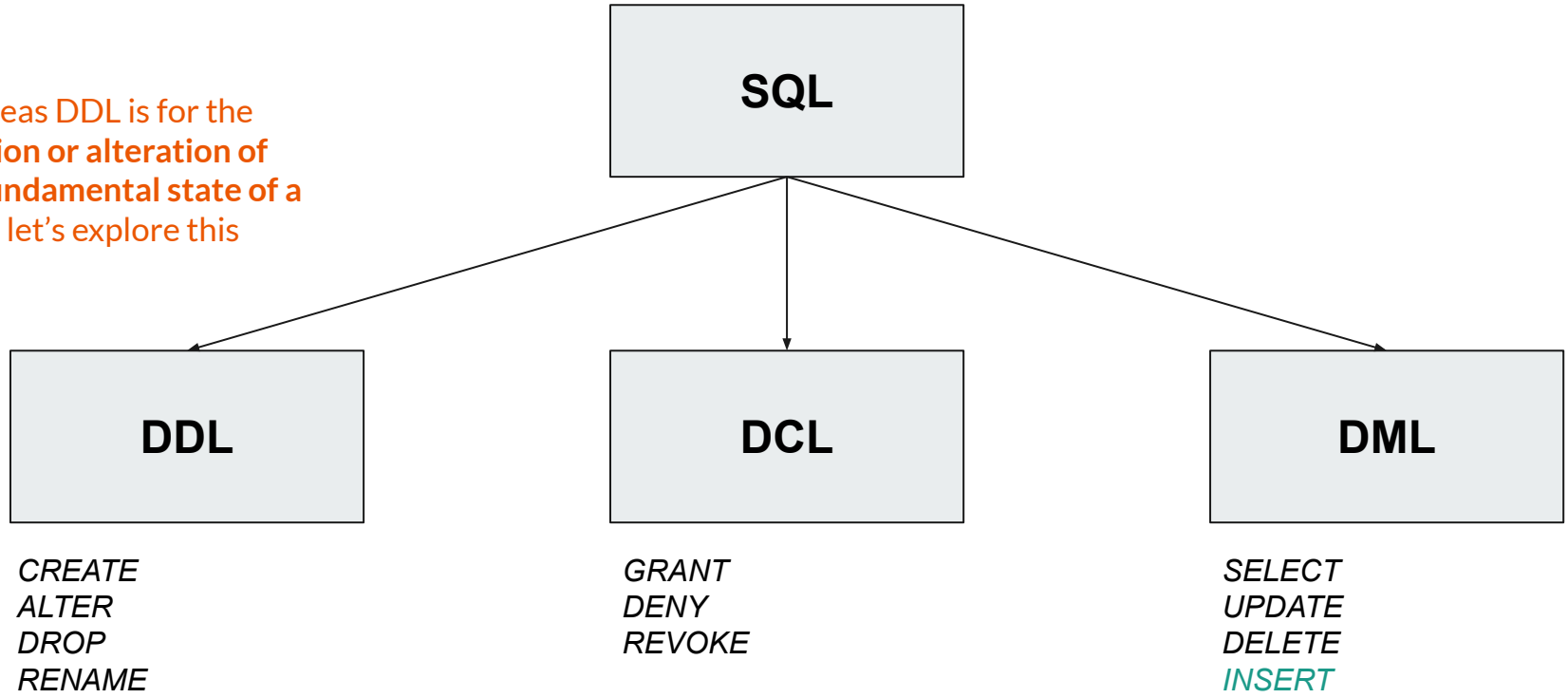
Based on what kind of operation we want to do, we use a "**different part**" of the SQL language. The basic 3 we will go over include **DDL, DCL, and DML**.

Note: DCL is for access control, we don't have to worry about this yet.



DML is solely reserved for "**manipulation**" of a table

Whereas DDL is for the **creation or alteration of the fundamental state of a table**, let's explore this more.



Statement	Description
CREATE	Create a new object in the database, such as a table or a view.
ALTER	Modify the structure of an object. For instance, altering a table to add a new column.
DROP	Remove an object from the database.
RENAME	Rename an existing object.

Taken from the Azure DP-900 prep

SQL - Logical Groups - DDL

DDL statements are usually one-liners…

***ALTER** [TABLE | PROCEDURE | …] name*

ACTION…

DROP *[TABLE | PROCEDURE | …] name*

***RENAME** [TABLE | PROCEDURE | …] name*

SQL - Logical Groups - DDL

But not when creating a table/defining a ***relation***!

```
CREATE TABLE Product (
    ID INT PRIMARY KEY,
    NAME VARCHAR(20) NOT NULL,
    PRICE DECIMAL DEFAULT 0.0
);
```

Notice that we not only define type, but we also often use specialized key-words to define some behavior.

Let's explore this behavior a bit more…

SQL - Logical Groups - DDL

```
CREATE TABLE Product (
    ID INT PRIMARY KEY,
    NAME VARCHAR(20) NOT NULL,
    PRICE DECIMAL DEFAULT 0.0
);
```

PRIMARY_KEY: uniquely identifies the sample (row)

FOREIGN_KEY: this row is connected to another table

NOT NULL means the value must NOT be empty (null)

DEFAULT gives a default value if nothing is put in during insertion

SQL - Logical Groups - DDL

```
CREATE TABLE Product (
    ID INT PRIMARY KEY,
    NAME VARCHAR(20) NOT NULL,
    PRICE DECIMAL DEFAULT 0.0
);
```

Furthermore, here are some tips for naming columns:

1. Make **sensible** names
2. Make **specific** names
3. Do not name columns the same as types, i.e. do not create “date” column, instead be specific and say “**purchase_date**”
4. **No spaces in column names**

ALTER TABLE

ALTER TABLE *(name)*
[ADD]
[DROP]
[RENAME COLUMN]
[ALTER/MODIFY COLUMN]

ALTER TABLE is used to add, delete, modify columns
after each of these commands is the column name

It is similar to how we would create a table

ADD column VARCHAR(50) NOT NULL
DROP old_column
RENAME COLUMN column TO new_name
MODIFY new_name VARCHAR(40)

the modify allows us to change the column from VARCHAR(50) to VARCHAR(40) or even other data types like INTEGER

```
ALTER TABLE Customers
RENAME COLUMN customer_id TO id;
```

For example, the following query renames an individual column “customer_id” to “id.”

UPDATE specific rows

```
UPDATE (name)
SET column1 = value1, column2 = value2
WHERE condition
```

UPDATE is special because it allows us to update very specific information

UPDATE table
SET changes specific information
WHERE condition

So you have to be careful and very specific
If you want to change a specific user's info, then you might want something like
WHERE user_id = '1231541521'

because if you are not specific, it will change ALL the records for EVERYTHING

It is better to test with a
SELECT column
FROM table
WHERE condition
first to make sure you are pulling in the right results before changing anything

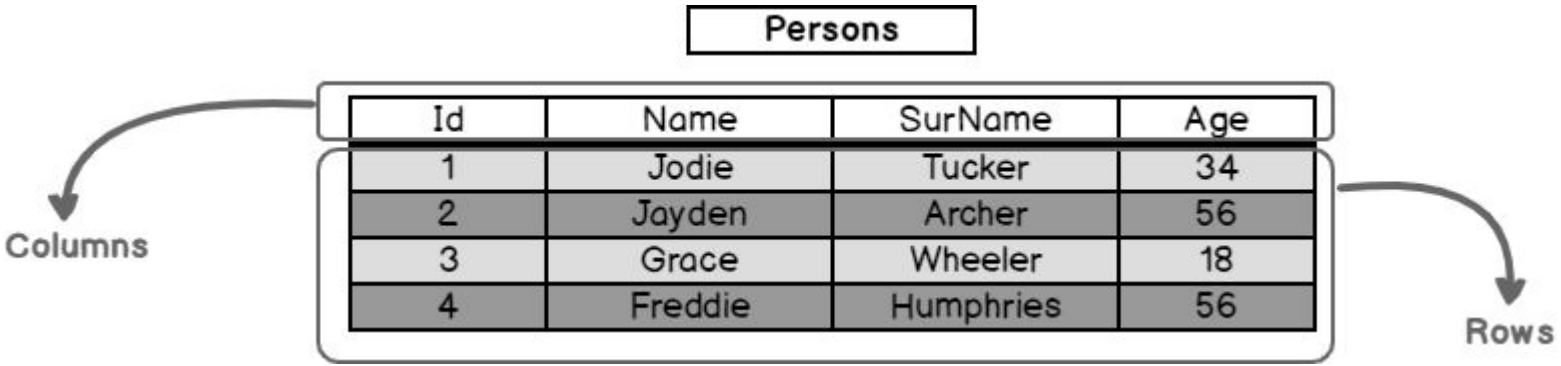
```
UPDATE Customers
SET country="USA"
WHERE country="UK";
```

Let’s say we made a mistake and accidentally labeled all customers from the US as from the UK. We could fix this mistake with the following syntax.

DML

SQL Overview/Review

Each column can have different types, here are some of the main ones:
INTEGER
VARCHAR(length) - this is equivalent to a Python string but we can set a maximum length
FLOAT
BOOL



SQL

When working in DML, we make **queries** that extract specific information from a database.

You can think of this as a new programming language! However it's best to keep this within the context of natural language…

```
SELECT first_name
FROM Customers
WHERE age > 27;
```

SQL Again

In English:

“Get all first names, from the “Customers” table, where the customer is above the age of 27”

```
SELECT first_name
FROM Customers
WHERE age > 27;
```

One step at a time:

SELECT * = get everything

We will usually choose specific columns

SELECT column_a

FROM table

SELECT *

FROM table

One step at a time:

We can rename tables (and columns) using aliases

```
SELECT first_name AS name
FROM Customers
WHERE age > 27;
```

NOTE: As our queries get larger, we start to omit the “AS.”

One step at a time:

We can filter the data using a **WHERE** clause

For instance, if we just want all data from orders that are larger than 500, we can simply **filter** on the **amount column**

```
SELECT *

FROM Orders

WHERE amount > 500;
```

Group By

We can aggregate our data by using a “GROUP BY” clause

When you “**group by**” something, you need to make sure that the column is **either selected or the other columns you select are aggregated in some way**

Here is an example:

```
SELECT country, COUNT(country)
FROM Customers
GROUP BY country;
```

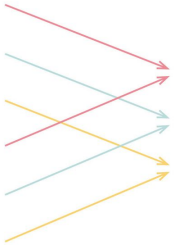
What do you think would happen if we didn’t have the “GROUP BY ”?

SQL GROUP BY

We use **GROUP BY** to combine (group) on column variables to get a result
The below query could be

```
SELECT genre, SUM(qty)
FROM books
GROUP BY genre;
```

title	genre	qty
book 1	adventure	4
book 2	fantasy	5
book 3	romance	2
book 4	adventure	3
book 5	fantasy	3
book 6	romance	1



genre	total
adventure	7
fantasy	8
romance	3

Note how the genre column “collapses” into the unique values and that is added together

Aggregate Functions

Let's look at other aggregate functions, you might be familiar with some of these

These take a group of data and summarize it as a single number

COUNT() - the # of rows for a particular feature (count the number of people per country)

SUM() - the sum of all values for a particular feature (add up all the costs per product)

AVG() - the average of all values for a particular feature (average purchase per customer id)

MIN() - the smallest value for a particular feature (smallest size per product)

MAX() - the largest value for a particular feature (largest size per product)

HAVING

When we group by, one of the ways we can filter is on an aggregate function. We do this by using **HAVING** with an aggregate function, this can be combined with a where statement as well

the **HAVING** clause *must* be used with an aggregate function

Try this without "HAVING." What happens?

for example:

```
SELECT country, COUNT(country) AS place_count

FROM Customers

GROUP BY country

HAVING place_count = 2;
```

ORDER BY/LIMIT

SQL results by default are random, if we care about order we can sort use ORDER BY and choosing a feature

```
SELECT *
FROM Customers
ORDER BY first_name
```

the default behavior is ascending (A-Z), if we want descending order we have to do:

```
ORDER BY cname DESC
```

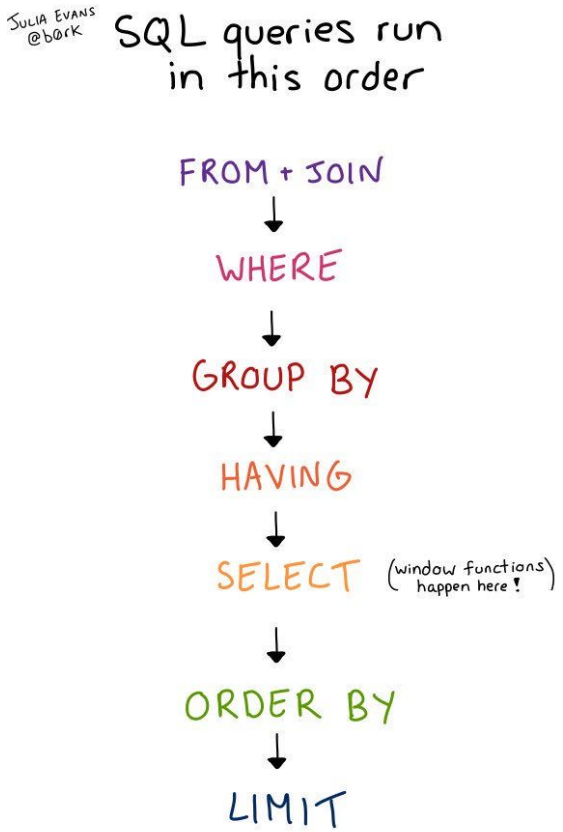
We can also limit the number of results through the usage of a limit clause

```
SELECT *
FROM Customers
ORDER BY first_name
LIMIT 3
```

SQL Order of Operations

It would be great if SQL would just run the statements from the top down… that would make sense, right?

Unfortunately, SQL has rules and things do not execute in the order you write them (which you *have* to write in that order)



Joins

Joins

We use joins to **combine tables with each other** (and sometimes to itself)

This allows us to keep information separate as needed (such as sales information from customer profiles) then bring them together as needed

Instead of having massive tables which take a long time to query, this is more efficient and reduces repetitive information (*for example, not having to save the customers email every time they make a purchase*)

SQL JOIN Syntax

The general syntax goes like so:

```
SELECT table1.column, table2.column
FROM table1 JOIN table2
    ON table1.foreign_key = table2.primary_key
```

Types of SQL JOIN

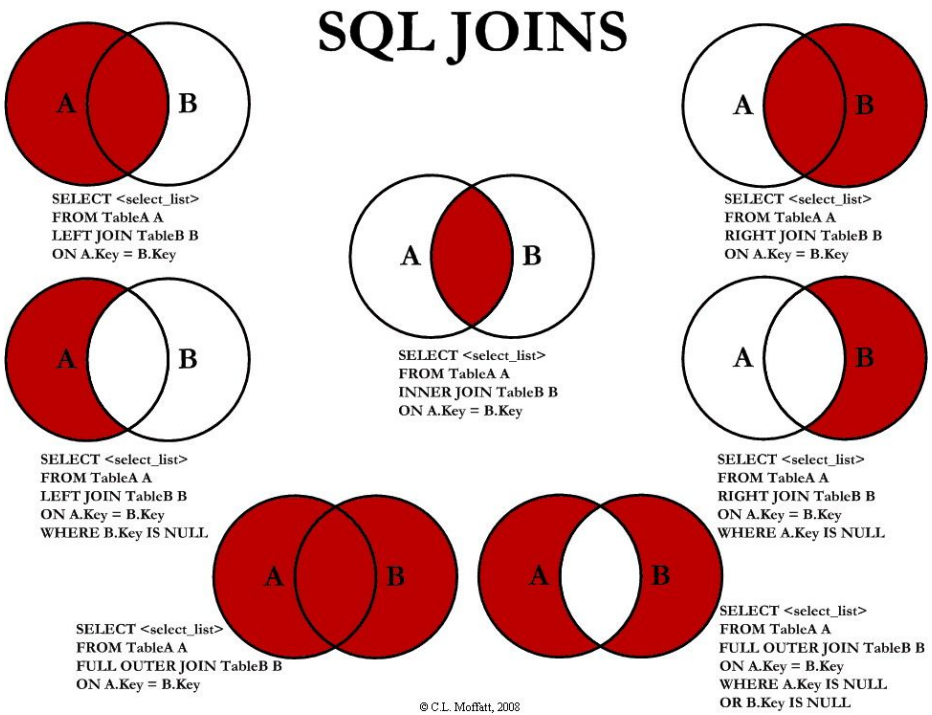
Inner Join: Returns only the rows where there is a match in ***both*** tables. (keep only matches)

Left Join (or Left Outer Join): Returns **all rows** from the **left** table and the **matched** rows from the **right** table. (Keep all left, keep matched right)

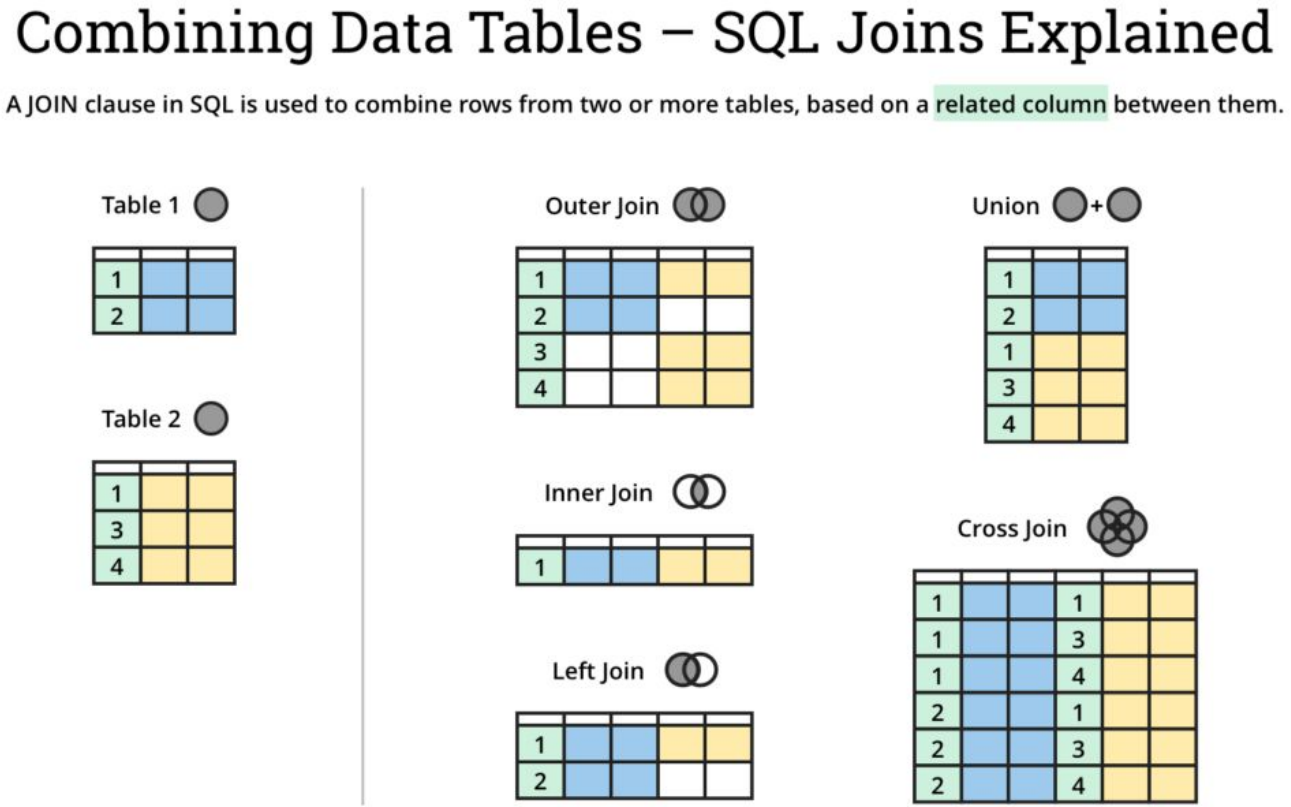
Right Join (or Right Outer Join): Returns all rows from the right table and the matched rows from the left table. (Keep all right, keep matched left)

Full Join (or Full Outer Join): Returns all rows when there is a match in either the left or right table. (keep all rows)

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

SQL JOINS


Combining Data Tables – SQL Joins Explained
A JOIN clause in SQL is used to combine rows from two or more tables, based on a related column between them.
Table 1
1
2
Table 2
1
3
4
Outer Join
1
2
3
4
Union
1
2
1
3
4
Inner Join
1
Cross Join
1 1
1 3
1 4
2 1
2 3
2 4
Left Join
1
2

SQL JOINS

We join by using a **Primary Key** from one table that is stored as a **Foreign Key** in another table

Here - the Customers Table has a customer_id, this is the **primary key** because it uniquely identifies each User

The Orders table has the **foreign key customer** because it relates to a **foreign table**: **customer**

We connect the primary key ID on Customers to Orders to get Customer Info on the order

SQL JOINS

We join by using a **Primary Key** from one table that is stored as a **Foreign Key** in another table

Here - the Customers Table has a customer_id, this is the **primary key** because it uniquely identifies each User

The Orders table has the **foreign key customer** because it relates to a **foreign table**: **customer**

We connect the primary key ID on Customers to Orders to get Customer Info on the order

When performing joins, we often alias table names by placing single letters after these tables. We use these letters in subsequent clauses.

```
SELECT *
FROM Customers c JOIN Orders o
    ON c.customer_id = o.customer_id
;
```

Here we join together customers & orders to coalesce these two tables and gather more information on a customer and their order. **Notice that we have the same column in two tables. This is not by coincidence!**

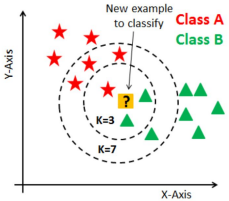
```
SELECT *
FROM Customers c LEFT JOIN Orders o
    ON c.customer_id = o.customer_id
;
```

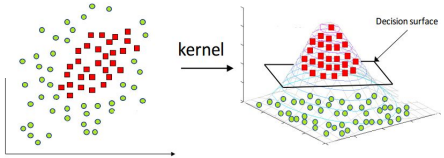
Try varying this type of join. What do you notice happens to your resultant table?

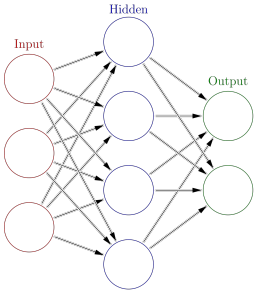
SQL Lab - SQL Bolt

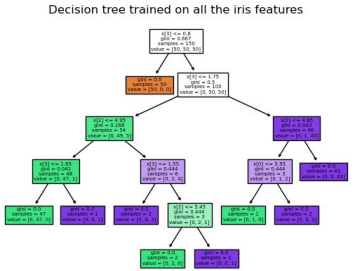
SQL Bolt

For the remaining lab time, work through [SQL Bolt (Lessons 1 - 8)]









Tuesday

On Tuesday we will review…

- *What is a “window”*
- *What is “advanced SQL”*
- *What is a common table expression?*
- *How do we use SQL with Python?*

How do different flavors of SQL differ?