

# Intro to Databases and Select

Module 2: 01

# Module 2 Overview

## SQL

Week 1: SQL Language

Week 2: Using SQL in Java

## API

Week 3: Consuming & Creating APIs in Java

Week 2: APIs continued

Module 2 API mini-capstone

# Week 5 Overview

## Monday

Intro to  
Databases

## Tuesday

Ordering,  
limiting, and  
grouping

## Wednesday

SQL Joins

## Thursday

INSERT,  
UPDATE, &  
DELETE

## Friday

Database  
Design

# Today's Objectives

1. Introduction to Databases
2. Tables, Rows, and Columns
3. ANSI-SQL Data Types
4. SQL Queries: SELECT

# Intro to Databases

A **database** is an organized collection of data that can be accessed, managed, and updated.

A **relational database** is a particular type of database built upon the relational model of data

Data in a **relational database** can be accessed and reassembled in many different ways without having to reorganize the data.

- Each **entity** is stored in a table.
- Columns are called **attributes**
- **Rows** represent individual records.

**Rows** represent individual records and consist of many attributes organized using **columns**.

# (R)DBMS

A Relational Database Management System ( (R)DBMS) is a software application designed to manage a database. It has four basic functions

1. Data Definition
2. Data Storage
3. Data Retrieval
4. Administration

RDBMSs include databases like Oracle, Microsoft SQL Server, PostgreSQL, MySQL, are relational, and are commonly called **SQL Databases**.

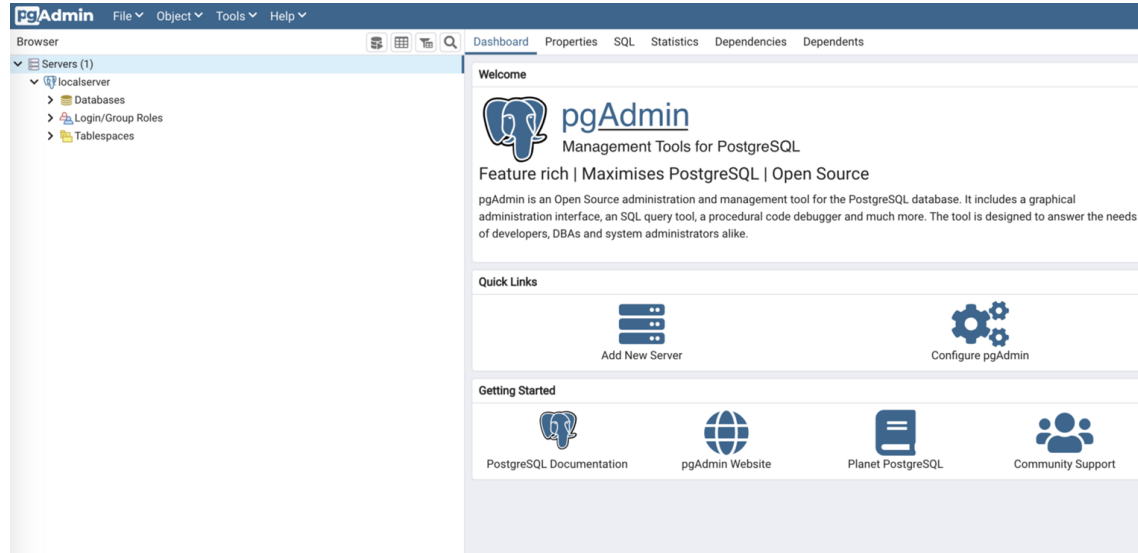
**NoSQL** Databases are those that do not use a relational structure, instead they structure data specific to the problem they are designed to solve. NoSQL databases include MongoDB, Cassandra, Google BigTable, HBase, DynamoDB, and Firebase.

[DB-Engines](#) has a ranking measuring popularity of current DBMS platforms.

# Database Management System (DBMS)

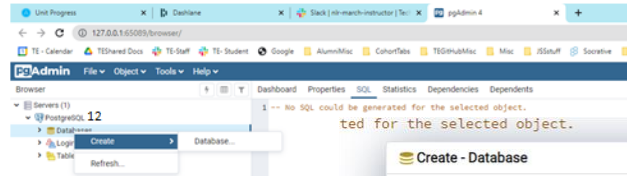
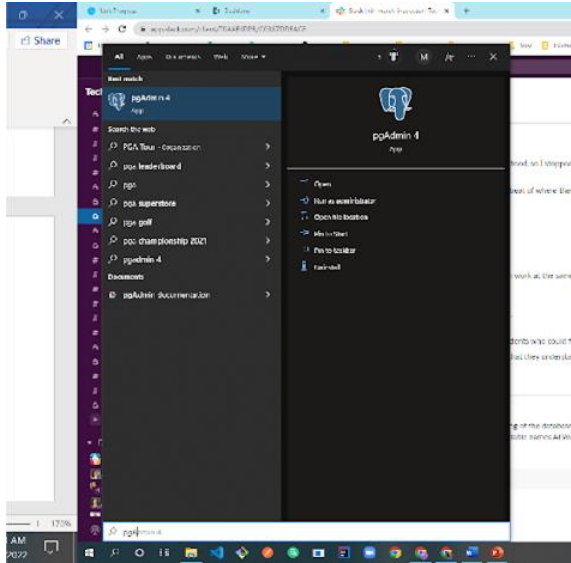


# RDBMS

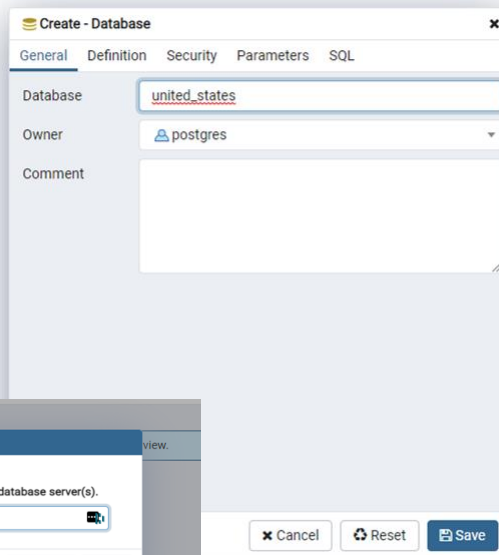




# Create a database in PgAdmin

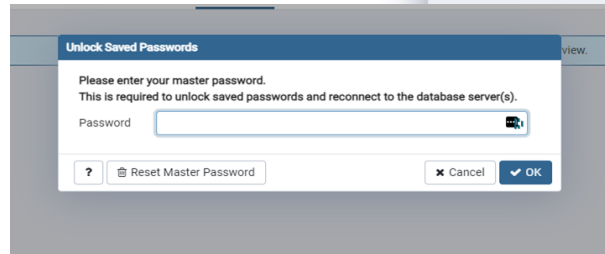


ted for the selected object.



Two passwords:

1. master password – defined your own
2. PostgreSQL 12 – postgres1



# Postgres - History

- “**PostgreSQL** ([/'poustɡrɛs ˌkjuː ˈɛl/](#), *POHST-gres kyoo el*),<sup>[13][14]</sup> also known as **Postgres**, is a [free and open-source relational database management system](#) (RDBMS) emphasizing [extensibility](#) and [SQL compliance](#). It was originally named POSTGRES, referring to its origins as a successor to the [Ingres](#) database developed at the [University of California, Berkeley](#).<sup>[15][16]</sup> In 1996, the project was renamed to PostgreSQL to reflect its support for [SQL](#). After a review in 2007, the development team decided to keep the name PostgreSQL and the alias Postgres.”
- - <https://en.wikipedia.org/wiki/PostgreSQL>


# Postgres - Ingres

- “**Ingres Database** ([/ɪnˈɡrɛs/](#) [ing-GRESS](#)) is a [proprietary SQL relational database management system](#) intended to support large commercial and government applications.”
- “**Ingres** stands for **IN**teractive **G**raphics **RE**trieval **S**ystem - a geographic database system for Berkeley's economics group.

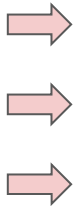
# Relational Database: Example

Suppose we are interested in storing data about cars. We can model a car entity into its own table:

This table has 4 attributes: CarName, Manufacturer, NumberOfDoors, FuelEconomy



CarName	Manufacturer	NumberOfDoors	FuelEconomy
Explorer	Ford	4	23
C-Class	Mercedes Benz	4	28
Jeep Wrangler	Fiat Chrysler	2	20



This table has 3 rows.

# Relational Database: Attribute Data Types

There is a large variety of data types in Postgresql, to name a few:

- **varchar**: holds text containing letters and numbers (somewhat like a String in Java).
- **char**: fixed length field containing letters and numbers.
- Various numeric data types:
- When referring to a non-numeric “text” field (i.e. varchar or char) we must surround them in single quotes (i.e. country=**'USA'**).
- Numeric literals do not need single quotes (numberOfDoors = **4**).

<https://www.postgresql.org/docs/9.3/datatype.html>

# Relational Database: SQL

- SQL is an acronym for Structured Query Language
- SQL is the language used to interact with relational database management systems.
- The exact implementation of SQL varies slightly depending on the database system involved, i.e. there will be minor differences in the language between PostgreSQL and MS SQL Server.
- This class will be using PostgreSQL.

# 3 types of commands

- DML

- Database Manipulation Language
  - INSERT, SELECT, DELETE, etc.

- DDL

- Data Definition Language
  - Commands for creating tables, defining relationships, etc.

- DCL

- Data Control Language
  - Commands that control permissions on the data and access rights

# SQL: SELECT

- The most basic SQL statement is a SELECT query, and it follows the following format:

SELECT **[column]**, **[column-n]** FROM **[table]**;

- **[column]** and **[column-n]** are stand ins for the attributes or columns that you want returned from your query.
- **[table]** refers to the name of the table you are querying.
- You can create column Aliases using the “**AS**” keyword followed by the alias.



# SQL: SELECT Example

Let's take the Vehicle table we just saw as an example:

- We could write the following SELECT statement:

***SELECT CarName, NumberOfDoors AS doors FROM Vehicle;***

The output of this would be:

CarName	doors
Explorer	4
C-Class	4
Jeep Wrangler	2

Note how the alias affects the column name in the output.

- Instead of listing specific columns we could use the wildcard \* to indicate that all columns should be returned: ***SELECT \* FROM Vehicle;***

# SQL: SELECT with WHERE clause

- We can include a WHERE clause in our select statements to limit the data returned by specifying a condition.
- The WHERE statement relies on comparison operators.
  - **Greater Than:** >
  - **Greater Than or Equal To:** >=
  - **Less Than:** <
  - **Less Than or Equal To:** <=
  - **Equal:** =
  - **Not Equal To:** <>, !=
- There is a special comparison operator called **LIKE** which is often used in conjunction with a wildcard (%) operator.

# SQL: SELECT with WHERE clause Example 1

Let's take the Vehicle table we just saw as an example:

- We could write the following SELECT statement:

***SELECT \* FROM Vehicle WHERE Manufacturer = 'Ford';***

- Only 1 row matches this criteria, and thus the results of the query will be:

CarName	Manufacturer	NumberOfDoors	FuelEconomy
Explorer	Ford	4	23

# SQL: SELECT with WHERE clause Example 2

Here is an example of the WHERE clause using the LIKE / Wildcard.

- We could write the following SELECT statement:

***SELECT \* FROM Vehicle WHERE CarName LIKE 'Ex%';***

- Only 1 row matches this criteria, and thus the results of the query will be:

CarName	Manufacturer	NumberOfDoors	FuelEconomy
Explorer	Ford	4	23

# Derived Columns with Math Operations

- A custom field containing math operations can be included in the SELECT.
- The basic math operators are present: **+**, **-**, **\***, **/**, **%**

```
SELECT employee_id, employee_name, salary, salary + 100  
       AS "salary + 100" FROM addition;
```

# Derived Columns Example

- Consider the following example:

***SELECT CarName, **FuelEconomy** \* 0.425144 AS kpl FROM Vehicle;***

CarName	kpl
Explorer	9.778312
C-Class	9.778312
Jeep Wrangler	8.50288

# SQL: AND / OR on WHERE statements

- Within the WHERE statement, various filter conditions can be combined using the AND / OR statement.
- Consider the following example:

***SELECT \* FROM Vehicle WHERE Manufacturer = 'Ford' OR NumberOfDoors = 4;***

- Two rows are returned:

CarName	Manufacturer	NumberOfDoors	FuelEconomy
Explorer	Ford	4	23
C-Class	Mercedes Benz	4	28

- Another look at:
  - **Tables, Columns, Rows**
  - **SQL**



# Tables, Columns, Rows

An **entity** is a set of data being stored a *table*.

A **Table** defines a set of data elements and the structure to store them. Tables are structured into *columns* and *rows*.

**Columns** - attributes of a table and define the name and data type. A table has a set number of defined columns.

**Rows** - the data being stored. A table has an unlimited number of rows.

# Column

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270

## Column

- Tables have a set number.
- Define the data the table will hold
- Provides a label for each part of the data being stored

## Columns on this Table

id, name, countrycode, district, population

# Row

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270

## Row

- Tables have a unlimited number (0...n)
- Contain the data
- Has a value for each column

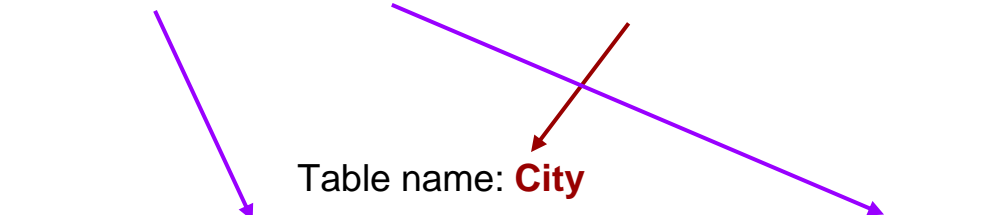
# SELECT

The **SELECT** clause *indicates what columns* to return in the results of the query

The **FROM** clause *indicates which table(s)* to retrieve the data from.

```
SELECT name, population FROM city;
```

Table name: **City**



id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270

# SELECT Modifiers

**\*** can be used to select all columns from a table.

```
SELECT * FROM country;
```

**AS** can be used with a column name to give it an **Alias** (new name)

```
SELECT name AS 'CityName' FROM city;
```

Or to give a name to a combined result:

```
SELECT ( col1 + col2 ) AS 'Sum'
```

**DISTINCT** can be used with a column name to return only unique values from that column.

```
SELECT DISTINCT name FROM city;
```

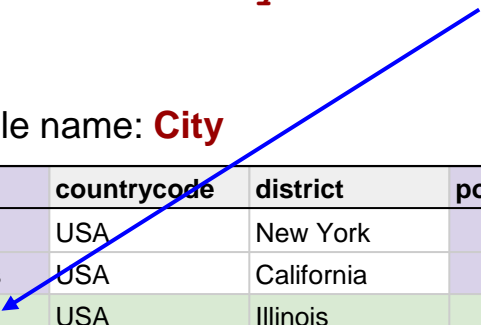
# WHERE

The **WHERE** clause is used to filter the rows returned in the results using a boolean condition. Rows that match that the expressions evaluates true for are returned by the query.

```
SELECT name, population FROM city WHERE name = 'Chicago';
```

Table name: **City**

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270



# WHERE Clause Conditionals

=	equal to
!=, <>	Not equal to
>, <, >=, <=	Greater/Less Than
IS NULL	value is null
IS NOT NULL	value is not null
IN (val1, val2, ...)	Value is IN the list
NOT IN (val1, val2, ...)	Value is NOT IN the list
BETWEEN va1 AND val2	The value is between the 2 values Example: number BETWEEN 2 AND 10;
LIKE (%)	The value matches a pattern created with % Example: a% - the value starts with a %a - the value ends with a %a% - the value contains an a

Conditionals can be chained together using **AND** and **OR**.

Precedent can be set using **()**

Example:

WHERE num > 5 AND  
(name LIKE 'A%'  
OR name LIKE 'B%')

Strings in SQL use *single quotes*.

name = 'John'