

# SQL Introduction

## Overview

### What We'll Cover

- Introduction to relational databases
- SQL: the “Structured Query Language”
- Creating Tables
- SELECT, INSERT, UPDATE, DELETE
- NULL
- Databases in Context
- Creating Databases
- Interacting with Data

### Objectives

1. Understand the purpose of relational databases in the context of software development.
2. Understand basic datatypes within SQL databases (PostgreSQL database).
3. Be able to apply SQL syntax to create database queries on a single table for multiple columns.
4. Student can set up a hosted database.
5. Student can execute database queries using a GUI tool.

# Relational Databases

Keep data in “relations” (tables)

## Customers

ID	Customer	Phone
1	Jessica	555-1212
2	Jada	123-4567
3	Jane	999-0000

## Orders

Date	Customer ID	Order Total
1/1	1	\$200
1/5	1	\$35
2/12	2	\$100
2/13	3	\$900
2/13	2	\$100

- Terminology
  - **Table:** base unit of information
  - **Record** (row): individual item
  - **Field:** individual attribute
  - **Database:** collection of tables
- Relational databases are most useful:
  - When objects have similar kinds of information
  - When you have complex questions to ask about data

## Non-Relational Databases

Relational DBs aren't the only kind.

But they're the most common.

For years, relational databases have been the most common way for programmers to persist data, though other strategies have always been also available. There's a trend toward using “NoSQL” databases (like MongoDB) for some applications but these are still a far less common choice than relational databases—so, here at Devmountain, we primarily talk about relational databases.

In addition, there are also simple “key-value stores” for storing very simple data. These kind of databases are often able to be even faster than relational data and use less memory – but offer far less sophisticated searching.

# SQL

- Language for querying / creating / updating relational databases
- Standard across different database products
  - Yay! Learn once and get to use with every database

(At least in theory. Though SQL is standardized, not all databases follow all parts, so as you get more advanced in your SQL, you may have to learn some small tweaks for different database products)

## Writing SQL

```
SELECT color, price FROM melons;
```

- Plain text, English-like
- Whitespace insignificant
- Case-insensitive
  - Conventional: `SQL KEYWORD`, `tablename`, `fieldname`
- In interactive systems, statements end with a semicolon
- String: `'value'` **must use single quotes**
- Int: `5`
- Float: `5.0`
- Date: `'2015-12-25'`

# Creating Data

## Basic Data Types

- `INTEGER`: whole numbers
- `DECIMAL`: unlimited decimal values
- `FLOAT`: up to 15 decimal places
- `TEXT`: unlimited characters in a string
- `VARCHAR(n)`: defined number of characters in a string
- `BOOLEAN`: true or false

## SERIAL & PRIMARY KEY

- `SERIAL` is like `INTEGER`, however, it is an automatically incrementing integer.
- `PRIMARY KEY` sets the data type to be a unique value, meaning no two rows can have the same value.

### Note: SERIAL

When inserting data into this table, you will not need to give a value for `employee_id` because `SERIAL` will handle that for you.

## NULL

- `NULL` is a placeholder for unknown
  - Similar to Python's `None` / JavaScript `null`
- All fields are “nullable” unless declared `NOT NULL`
  - Including numbers, boolean, etc!
- Best practice: Make everything `NOT NULL` where possible!

### Note: NULL Comparison

#### Python:

```
>>> 7 + None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

#### JavaScript:

```
7 + null    // --> 7
```

#### SQL:

```
SELECT 7 + NULL;    -- NULL
```

```
SELECT 7 > NULL;    -- NULL
```

```
SELECT 7 = NULL;    -- NULL
```

```
SELECT NULL = NULL; -- NULL
```

```
SELECT 7 IS NULL;   -- FALSE
```

```
SELECT NULL IS NULL; -- TRUE
```

## CREATE TABLE

We create tables as places to store data.

When we create tables, we declare which data type a column will have as well as any constraints (like `NOT NULL`).

What information do we need to make a table about melons?

Field	Info	Required?
ID	Number	Yes
Name	String	Yes
Color	String	No
Price	Amount of money	No

How do we create the melons table using SQL?

```
CREATE TABLE melons(  
  id SERIAL PRIMARY KEY, -- this will never be null because it's serialized  
  name VARCHAR(40) NOT NULL,  
  color VARCHAR(20),  
  price INTEGER  
);
```

## INSERT

To create data within tables, we can **INSERT** it.

```
INSERT INTO melons (name, color, price)  
VALUES ('Watermelon', 'green', 4);
```

Or to create more than 1 row at a time:

```
INSERT INTO melons (name, color, price)  
VALUES ('Spring Melon', 'green', 3),  
('Sad Melon', null, null),  
('Berry Melon', 'blue', null);
```

- We don't have to provide **id** (that's job of serial)
- Had to provide a name for each since it can't be null
- Had to explicitly provide `NULL` for missing fields

# Getting Data

## SELECT Statements

Query your database.

Doesn't change data.

A **SELECT** statement returns a result but it doesn't create or edit any data – it just answers a question.

Determines fields/expressions to list.

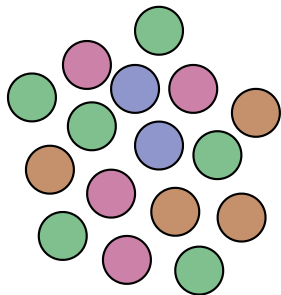
```
SELECT * FROM melons; -- "*" means all columns
```

```
SELECT id, color FROM melons; -- only gets the id and color columns
```

```
SELECT price / 2 FROM melons; -- gets the price divided in half for each row
```

## FROM

```
SELECT * FROM melons;
```



Gets raw data from tables

Name	Color	Price
Watermelon	green	\$4
Spring Melon	green	\$2
Zest Melon	green	\$4
Fresh Melon	green	\$4
Yum Melon	green	\$5
Honeydew	brown	\$2
Paper Melon	brown	\$6
Thin Melon	brown	\$7
Sad Melon	brown	\$6
Blue Melon	blue	\$1
Berry Melon	blue	\$6
Pink Melon	pink	\$4
Spice Melon	pink	\$3
Summer Melon	pink	\$6
Best Melon	pink	\$9

## WHERE

Selects row(s) **WHERE** a certain condition is met. We can check conditions with certain operators.

### Operators

- Equal: `=`
- Not Equal: `<>` (or `!=`)
- Comparison: `>`, `<`, `>=`, `<=`
- Match any string: `LIKE 'StartsWith%'`
- Boolean: `AND`, `OR`, `NOT`, `(`, `)`
- Between (inclusive): `BETWEEN x AND y`
- In List: `IN (1, 2, 3, 4)`

#### Note: Syntax

Some databases allow `!=` to be used for inequality, but `<>` is standard.

Many SQL databases are case-sensitive when searching strings with `=`, but some are not.

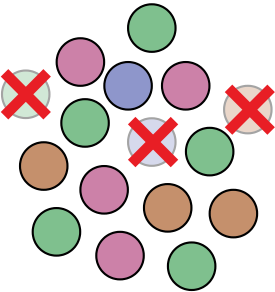


## Examples

```
SELECT * FROM melons WHERE color IN ('pink', 'blue'); -- gets all pink and blue melons

SELECT * FROM melons WHERE color = 'green' AND price <> 5; -- gets green melons that are NOT $5

SELECT * FROM melons WHERE price > 2; -- shown next
```



Throws out non-matching rows. (The grayed out ones in the table).

Name	Color	Price
Watermelon	green	\$4
Spring Melon	green	\$2
Zest Melon	green	\$4
Fresh Melon	green	\$4
Yum Melon	green	\$5
Honeydew	brown	\$2
Paper Melon	brown	\$6
Thin Melon	brown	\$7
Sad Melon	brown	\$6
Blue Melon	blue	\$1
Berry Melon	blue	\$6
Pink Melon	pink	\$4
Spice Melon	pink	\$3
Summer Melon	pink	\$6
Best Melon	pink	\$9

## GROUP BY

This will group your results together.

If you don't use **GROUP BY**, you're working with individual melons. Selecting like this will get us a nice list of values with no repeats.

```
SELECT color  
FROM melons  
GROUP BY color;
```

Color

green

brown

blue

pink

## Aggregates

```
SELECT COUNT(*) FROM melons;

SELECT AVG(price) FROM melons;

SELECT SUM(price) FROM melons;

SELECT MIN(price) FROM melons;

SELECT MAX(price) FROM melons;
```

`COUNT(*)` is “count all the items in this group”

`AVG(price)` is “find the average of this column”

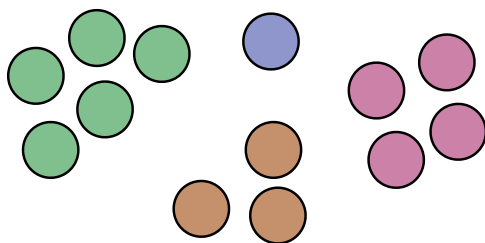
`SUM(price)` is “add all the numbers in this column”

`MIN(price)` is “find the smallest value in this column”

`MAX(price)` is “find the largest value in this column”

Using aggregates with `GROUP BY` opens up lots of possibilities:

```
SELECT color, COUNT(*)
FROM melons
GROUP BY color;
```



Cluster by attribute(s).

Color	Count
green	5
brown	3
blue	1
pink	4

### Note: Without GROUP BY

If you use an aggregate expression without an explicit **GROUP BY** clause, the entire result set is put into one group, so you’ll often just get a single row.

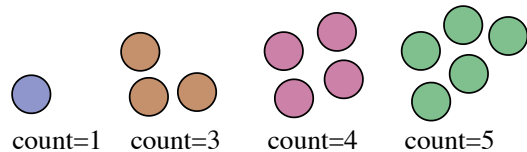
```
SELECT COUNT(*)
from melons;
```

This query would just return to us the number of rows that are in our table.

## ORDER BY

If you don't use **ORDER BY**, you can't predict order.

```
SELECT color, COUNT(*)
FROM melons
GROUP BY color
ORDER BY COUNT(*);
```



Order groups by fields/expressions.

Color	Count
blue	1
brown	3
pink	4
green	5

The SQL standard actually requires that, for expression, **ORDER BY** be given a number which refers to the column # in the select statement (so our example “should” be written `ORDER BY 2 DESC`). Almost all databases, however, allow the **ORDER BY** clause to refer to expressions by name, as shown here – including PostgreSQL.)

## Batching

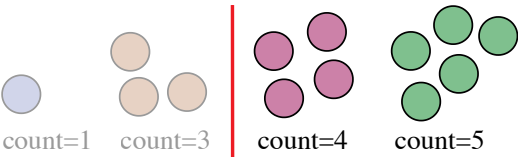
**OFFSET** and **LIMIT** are often used together to “batch results”, like you might see when browsing lots of data:

Items per page: 15 1 2 3 ... 8396

OFFSET

Tells the query how many rows to skip starting from the **top**.

```
SELECT color, COUNT(*)
FROM melons
GROUP BY color
ORDER BY COUNT(*)
OFFSET 2;
```



count=1   count=3   count=4   count=5

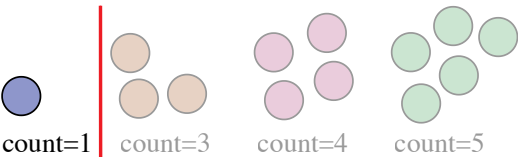
	Color	Count
	blue	1
	brown	3
	pink	4
	green	5

Number groups to skip (default 0)

LIMIT

Tells the query how many rows to cut off starting from the **bottom**.

```
SELECT color, COUNT(*)
FROM melons
GROUP BY color
ORDER BY COUNT(*)
LIMIT 1;
```



count=1   count=3   count=4   count=5

	Color	Count
	blue	1
	brown	3
	pink	4
	green	5

Number of rows to return (default all)

## SELECT Overview

**From** this table get rows,  
throwing out **where** they fail this,  
**group them up** like so.  
Now, **select** this info  
and put it this **order**,  
**offset** (skip) this many,  
and **limit** results to this many.

That's the order it happens in, even though that's not the order we write it in.

You do not have to use every clause every time you select.

### Note: HAVING

There is another clause that you can use when selecting using SQL called **HAVING**. It's the way to provide a conditional for a **GROUP BY** or an aggregate.

# UPDATE

## Purpose

Update is used to update the value(s) stored in the field(s) of a table.

It does not change the structure of the table, just the value(s) stored.

## Example

Let's say we have the following data stored in our database. We want to update the price of Spring Melons to be \$3.

ID	Name	Color	Price
1	Watermelon	green	4
2	Spring Melon	green	2
3	Zest Melon	green	4
4	Fresh Melon	green	4
5	Yum Melon	green	5
6	Honeydew	brown	2
7	Paper Melon	brown	6
8	Thin Melon	brown	7
9	Sad Melon	brown	6
10	Blue Melon	blue	1
11	Berry Melon	blue	6
12	Pink Melon	pink	4
13	Spice Melon	pink	3
14	Summer Melon	pink	6
15	Best Melon	pink	9

Syntax

```
UPDATE melons
SET price = 3
WHERE id = 2;
```

ID	Name	Color	Price
1	Watermelon	green	4
2	Spring Melon	green	3
3	Zest Melon	green	4
4	Fresh Melon	green	4
5	Yum Melon	green	5
6	Honeydew	brown	2
7	Paper Melon	brown	6
8	Thin Melon	brown	7
9	Sad Melon	brown	6
10	Blue Melon	blue	1
11	Berry Melon	blue	6
12	Pink Melon	pink	4
13	Spice Melon	pink	3
14	Summer Melon	pink	6
15	Best Melon	pink	9



# DELETE

## Purpose

Delete is used to delete row(s) from a table.

Be careful with this, as once something is deleted, it's gone forever.

## Example

Let's say we wanted to delete all the green melons because we don't like to eat green things.

ID	Name	Color	Price
1	Watermelon	green	4
2	Spring Melon	green	2
3	Zest Melon	green	4
4	Fresh Melon	green	4
5	Yum Melon	green	5
6	Honeydew	brown	2
7	Paper Melon	brown	6
8	Thin Melon	brown	7
9	Sad Melon	brown	6
10	Blue Melon	blue	1
11	Berry Melon	blue	6
12	Pink Melon	pink	4
13	Spice Melon	pink	3
14	Summer Melon	pink	6
15	Best Melon	pink	9

## Syntax

```
DELETE  
FROM melons  
WHERE color = 'green';
```

ID	Name	Color	Price
6	Honeydew	brown	2
7	Paper Melon	brown	6
8	Thin Melon	brown	7
9	Sad Melon	brown	6
10	Blue Melon	blue	1
11	Berry Melon	blue	6
12	Pink Melon	pink	4
13	Spice Melon	pink	3
14	Summer Melon	pink	6
15	Best Melon	pink	9

## Database Context

### Where do databases live?

#### Locally

- for development/testing purposes, or small personal projects, you can create local DBs on your own machine
- some companies keep data on computers locally, meaning computers that are accessible to them physically/in office

#### Remotely

- for personal or production applications, data can also be stored on remote computers that are accessed over the internet
- there are companies that own tons of computers and allow people to pay to use them to host whatever websites/applications they want (both front and back end)

# Creating Databases

## Remotely

- in Foundations, we'll work with remote databases
- to create and host our databases, we'll be using a service called bit.io, which takes care of lots of setup for us
- once the database is created, we can access it through its `URI` (uniform resource identifier)
- then we **seed** the database with initial information so that we have data to work with

## Other Routes

In some Specializations, you'll be setting up local Postgres databases on your computer. This is done with command line tools and commands that are out of the scope of Foundations.

There are also other hosted options, but we'll use bit.io in Foundations.

# Interacting with Data

## Intro

The ability to interact with our data is *super* important.

Without interaction, there wouldn't be much of a reason to store data.

## What do we do with data and why?

A broad overview:

what	why
store	information in databases persists so we can use it however/whenever we need
analyze	to understand trends, solve problems, gain insight, and <b>make decisions</b>
connect (to apps)	so users can easily read/create/update/delete data

## How do we access our data?

There are lots of options out there! Some are developer focused while some are more business focused.

- writing code (Sequelize, MassiveJS, etc.)
- using apps for developers (PG Admin, Postico, Retool, etc.) or even through the command line
- with services that offer tools for storing and interacting with your data (Salesforce, HubSpot, Shopify, etc.)
- through business intelligence apps (Google Analytics, Databox, Domo, etc.)

## What We'll Use: bit.io

- bit.io is both a DB hosting service and an app used to interact with those hosted DBs

## The End