Complete SQL Bootcamp

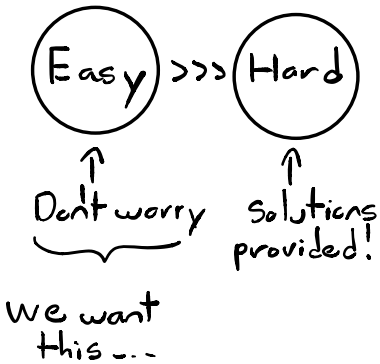
SECTION II: SQL Statement Fundamentals

「Video 10」 Overview of challenges

- Throughout the course you will receive challenges. → Allows you to test your skills of PostgreSQL and Basic / Advanced SQL statements.
- The basis of these challenge scenarios will be that you've just been hired as an SQL consultant for a DVD rental store.

↳ Solutions to challenges

↳
- challenges will start off very simple, basically asking you to repeat the previous lecture.
- Later, you will be provided with solutions or walkthrough video solutions for the tougher challenges.
- Challenges may show up unannounced, so always be ready for them.



「Video 12」 SQL Statement Fundamentals

- This section focuses on SQL Syntax.
- IMPORTANT -> The syntax learned in this Section of the course can be applied to any major SQL RDBMS! (MySQL, Oracle, etc...)
- We will start with the basics and have challenges along the way to test your understanding.

「Video 13」 SELECT Statement

- SELECT is the most common statement used, and it allows us to retrieve information from a table within a DB.
- Later -> we will learn how to use the SELECT statement combined with other statements to perform more complex queries.

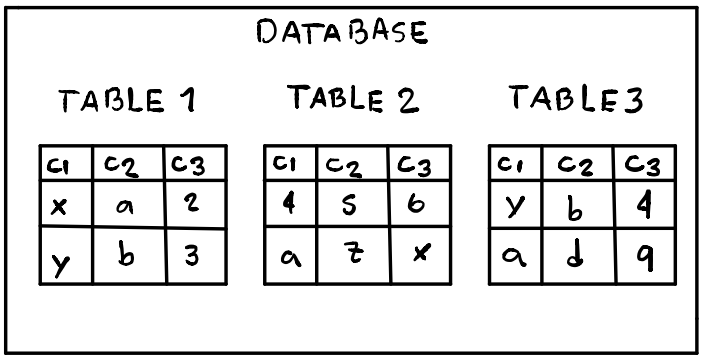
Example syntax for SELECT statement:

```
SELECT column_name FROM table_name
```

In general we capitalize the SQL keywords. This is not mandatory, but it makes it easier to read: SQL staments vs. Column/table names.

- SQL will still run if you don't do this.
- How does this work within PostgreSQL?

```
SELECT column_name FROM table_name
```



→ Each of the tables in this DB work like spreadsheets.

→ They all have columns & rows with made up data.

PostgreSQL → SELECT ---② FROM ---①.

→ PostgreSQL already knows you are referencing this DB (This is how you open the query editor in the first place)

→ First ①, it takes a look at which table you are looking at, then ② it takes a look at the corresponding column name.

- Selecting multiple columns syntax:

```
SELECT c1, c2 FROM table_name
```

Query output

output will be given in this order! c1,c2 ≠ c2,c1

- Selecting all the columns from a table:

  `SELECT (*) FROM table_name`

  ↑ Returns the whole table.

  ↑

  IMPORTANT → In general, it is not good practice to use an asterisk (*) in the SELECT statement if you don't really need all the columns.

  Automatically queries everything!

  ↓

  Increases traffic b/w DB Server and application. } slows down retrieval of results!

- Thus, if you ONLY need certain columns, do your best to only query for those columns!

Code notes

- ; At the end of a query means "end of query" → This is optional
- Pressing F5 executes the query. (pgAdmin)

[Video 15] SELECT DISTINCT statement

- Sometimes, a table contains a column that has duplicate values, and you may find yourself in the situation where you only want to list the unique / distinct values.

↓

The DISTINCT keyword can be used to return only the distinct values in a column.

↓

The DISTINCT keyword operates on a column. The syntax looks like this:

```
SELECT DISTINCT column FROM table
```

which is equivalent to:

```
SELECT DISTINCT (column) FROM table
```

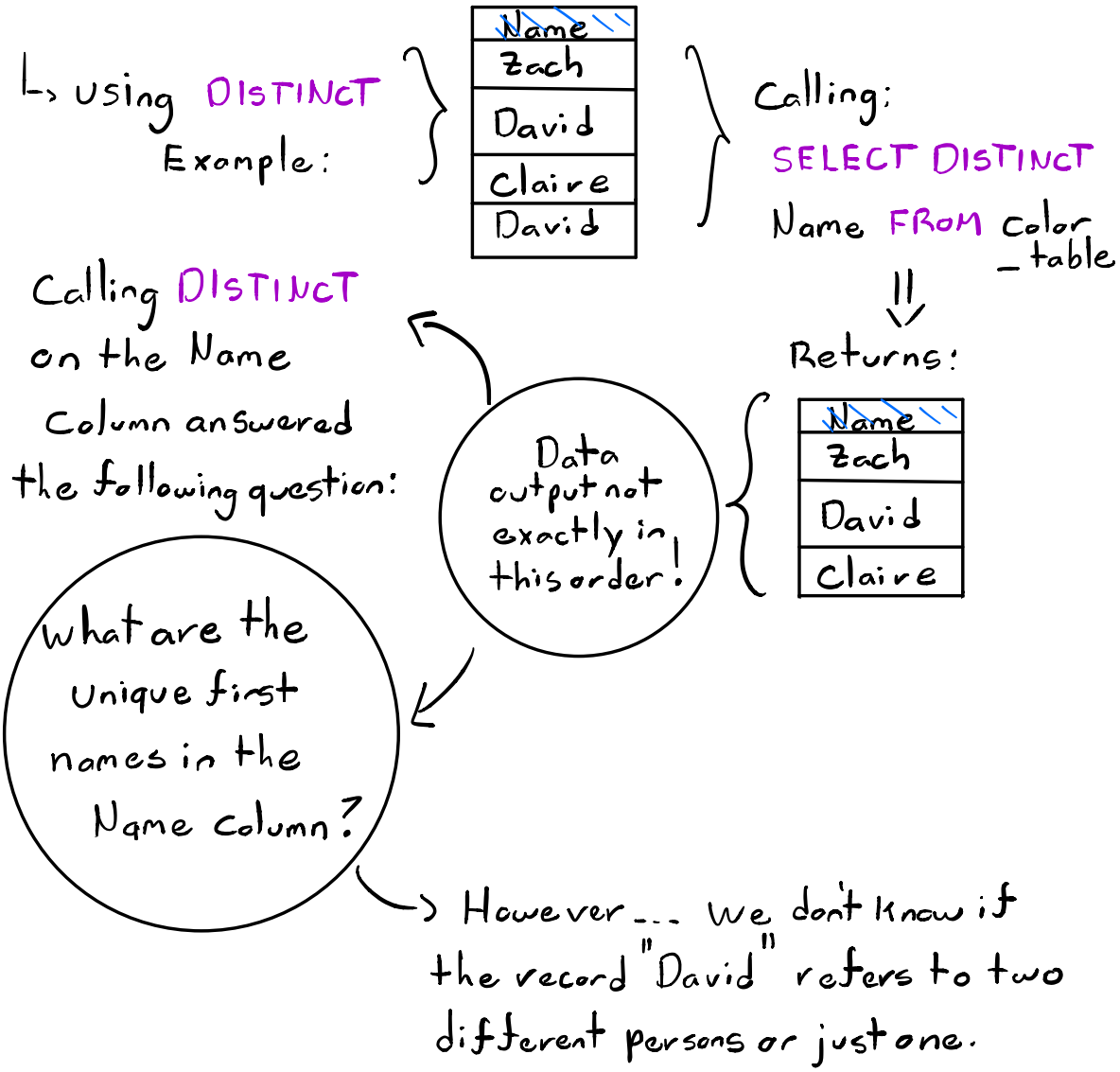
LATER... Parentheses will be necessary when we add other calls such as COUNT

You can add the parenthesis for clarity (In this case, they are optional)

Color_table

Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

Survey of people ↓ Chose a color.



「Video 17」 COUNT statement

- The COUNT function returns the number of input rows that match a specific condition of a query.
- We can apply COUNT on a specific column or just pass COUNT (*), we will soon see this should return the same result.

↳ In this case, COUNT merely reports the number of rows returned.

Regardless of the column you choose.

Example:

Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

↳ Syntax:

```
SELECT COUNT(Name) FROM color_table;
```

Paretheses are required!

↳ COUNT is a function acting on something!

Returns → 4

Number of rows in the table!

Each column has the same number of rows

Result is the same regardless of the column!

```
SELECT COUNT(Name) FROM color_table
SELECT COUNT(Choice) FROM color_table
SELECT COUNT(*) FROM color_table
```

All return the same thing since the original table had 4 rows.

It is sometimes useful to use a column name instead of * → Helps you remember the question you were trying to answer!

- Because of this, COUNT by itself simply returns back a count of the number of rows in a table.
- COUNT is much more useful when combined with other commands such as DISTINCT.

- Imagine if we wanted to know: How many unique names are there in a table?

Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

↳ Solution:

```
SELECT COUNT( DISTINCT Name) FROM color_table;
```

DISTINCT Name: Returns the unique names (3 rows)

COUNT will count the remaining rows → Returns 3.

Again, notice the parentheses structure. This structure informs us we are calling COUNT on the result of DISTINCT Name.

```
SELECT COUNT(DISTINCT(Name)) FROM color_table;
```

we can also use parentheses on the DISTINCT call to increase readability.

「Video 18」 SELECT WHERE Statement

- SELECT and WHERE are the most fundamental SQL statements and you will find yourself using them often.
- The WHERE statement allows us to specify conditions on columns for the rows to be returned.
- Basic syntax example:

```
SELECT column1, column2
FROM table
WHERE conditions;
```

- The WHERE clause appears immediately after the FROM clause of the SELECT statement.

what are these conditions?

=> The conditions are used to filter the rows returned from the SELECT statement.

- PostgreSQL provides a variety of standard operators to construct the conditions.

  * Conditions shown here -> Available in all SQL engines.

(1) Comparison operators

  L> Allows us to compare a column value to something.

  L> With them, we will be able to answer questions such as:

  - Is the price greater than $3.00?
  - Is the pet's name equal to "Sam"?

L> Comparison operators:

operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal

② Logical operators

↳ Allows us to combine multiple Comparison operators.

- AND
- OR
- NOT

Syntax Example

Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

```
SELECT Name, choice FROM
Color_table
WHERE Name = 'David';
```

--> single quotes

case sensitive

Given the condition, only the entries whose Name is David will be returned.

Name	Choice
David	Green
David	Red

- Testing multiple conditions:

```
SELECT Name, Choice FROM
Color_table
WHERE Name = 'David' AND
Choice = 'Red'
```

=> Returns a single entry:

Name	Choice
David	Red

「Video 21」 ORDER BY statement

- You may have notice that Postgre SQL sometimes returns the same request query results in a different order.
- You can use ORDER BY to sort rows based on a column value, in either ascending or descending order.
  - ↳ Alphabetical → string based column
    Ascending / Descending → Numeric based columns

Basic syntax

```
SELECT column 1, column 2
FROM table
ORDER BY column 1 ASC/DESC
```

Default is ASC

ASC : DESC → optional argument to sort in Ascending or Descending order

- Notice ORDER BY towards the end of the query → We WANT to do any selection or filtering first. Finally, we will sort.

- You can also ORDER BY multiple columns.
- This makes sense when one column has duplicate entries:

COMPANY	NAME	SALES
Apple	Andy	100
Google	David	500
Apple	Zach	300
Google	Claire	200
Xerox	Steven	100

Executing the query below

```
SELECT Company, name, sales
FROM table
ORDER BY Company, sales
```

→ order of parameters is relevant!

Returns!

COMPANY	NAME	SALES
Apple	Andy	100
Apple	Zach	300
Google	Claire	200
Google	David	500
Xerox	Steven	100

} ASC by default!

① First ordering by Company

② Second ordering by sales within each Company

- Duplicate values
- Sort them
- Further sort by another column.

* we can know who sold the most within each Company. *

[Video 22] LIMIT statement

- The LIMIT command allows us to limit the number of rows returned for a query.
- useful?
  - ① Returns specified number of rows
  - ② Gives an idea of the general layout of the table
- LIMIT also becomes useful in combination with ORDER BY.
- LIMIT goes at the very end of a query request and is the last command to be executed.
  - After filtering, selecting AND ordering!

It all boils down to... → How many rows do you want at the end of the day.

[Video 24] BETWEEN operator

- The BETWEEN operator can be used to match a value against a range of values.
  - ↳ Value BETWEEN low AND high.
  - ↳ This a condition you can use along the WHERE statement.

- The BETWEEN operator is the same as:

  Value ≥ low and value ≤ high

  Value BETWEEN low AND high.

  Endpoints both inclusive!

- The NOT BETWEEN operator is the same as:

  Value < low and value < high

  Value NOT BETWEEN low AND high.

  Endpoints are exclusive!

- The BETWEEN operator can also be used with dates.
  - ↓
  - Note that you need to format dates in the ISO 8601 standard format.
    - ↳ YYYY-MM-DD

↳ date BETWEEN '2007-01-01'
AND '2007-02-01'

- When using the BETWEEN operator with dates that also include timestamp information, pay careful attention to using BETWEEN versus <=, >= comparison operators.

datetime starts at 0:00

- Later → we will study more specific methods for datetime information types.

「Video 25」 IN operator

- In certain cases you want to check for multiple possible value options. For example, if a user's name shows up IN a list of known names.
- We can use the IN operator to create a condition that checks if a value is included in a list of multiple options.

→ Syntax pattern:

Value IN (option 1, option 2 --- option N)

↳ Example query:

```
SELECT color FROM table
WHERE color IN ('red', 'blue')

SELECT color FROM table
WHERE color IN ('red', 'blue', 'green')
```

An internal OR operator is used here.

↳ Note we can also use the NOT operator to exclude the values!

```
SELECT color FROM table
WHERE color NOT IN ('red', 'blue')
```

[Video 26] LIKE and ILIKE statements

Using Pattern matching

- This lecture is about pattern matching using string data.
- We've already been able to perform direct comparisons against strings such as:

```
WHERE first_name = 'John'
```

- But what if we want to match against a general pattern in a string?
  - ↳ Allows us to find patterns such as
    - All names ending with '@gmail.com'
    - All names that begin with 'A'

- The LIKE operator allows us to perform pattern matching against string data with the use of wildcard characters:

  1) Percent % -> Matches any sequence of characters

  2) Underscore _ -> Matches any single character

Examples (%)

- All names that begin with 'A'

  WHERE name LIKE 'A%'

  Finds any sequence of characters that begin with uppercase 'A'

- All names that end with 'a'

  WHERE name LIKE '%a'

  Case sensitive!

  ⇓

  We can use ILIKE which is case insensitive.

Examples (_)

- Using the underscore (_) allows use to replace just a single character.

↳ • Get all the Mission Impossible films

```
WHERE title LIKE 'Mission Impossible _'
```

↑ _ fills the wildcard place to fill in the # we are looking for.

• You can use multiple underscores

↳ • Imagine we had version string codes in the format 'Version #A4', 'Version #B7'

```
WHERE value LIKE 'Version #__'
```

Leaves open any two characters to fill up these underscore positions.

Examples (Complex patters)

↳ • We can also combine pattern matching operators to create more complex patterns!

e.g.

```
WHERE name LIKE '_her%'
```

↳ Returns:
- Sherri
- Cheryl
- Theresa

'_her%'

↑ Just one character allowed for this wildcard (_)

↳ Any sequence of characters for this wildcard (%)

- Keep in mind that PostgreSQL allows for full regex capabilities!

  Check out the documentation for more info!

↳ For now we will focus on LIKE and ILIKE!