SQL

① Creating & Restoring database

Creating a database

→ Create a database

Click on database → Click Create database → Name it

That's it

→ Restoring database

→ Click on database → restore → location

That's it

② Restoring a table schema

↳ Table names & Data types only, No actual data

↳ Very Common task you should know well

Create a New database → only schema

↓

Restore → place it in filepath

↓

Restore option 1

↳ type of objects

↳ only schema

Done

No data is there ! just table schema

→ New one

to make it in existing database already had data

↓

Restore → Restore option 1

↳ type of objects

↳ only schema

then → Restore option 2

↳ Queries

↳ clean before restore

(3) SQL Statement Fundamentals

(3-1) SELECT Statetement

→ Syntax

" SELECT Column 1, Column 2, ... FROM table name "

→ First, you specify a list of Columns in the table from which you want to query data in the SELECT Clause, you use a Comma between each Column in case you want to query data from multiple Columns.

→ If you want to query data from all Column, you can use an asterisk (*) as the shorthand for all Columns.

→ Second, you indicate the table name after the FROM keyword.

→ It is not good practice to use the (*) in the SELECT statement.

→ It makes your database server work harder & increase the traffic between the database server & applications. As a result, it slows down your application.

→ Therefore; You should specify the column names in the SELECT Clause whenever possible to get only necessary data from a table.

Eg:- SELECT * (all column) FROM actor (table);

SELECT first_name, last_name FROM actor;

SELECT actor_id FROM actor;

=> Unique values in Python

3.2 SELECT DISTINCT Statement

→ In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.

→ The DISTINCT keyword can be used to return only distinct (different) values.

Syntax

```
" SELECT DISTINCT Column_1, Column_2.....
  FROM table_name; "
```

Eg :- SELECT DISTINCT release_year FROM film;

SELECT DISTINCT rental_rate FROM film;

3.3 SELECT WHERE Statement

→ In the previous lectures, you've learned how to use the SELECT statement to query data from a table.

→ What if you want to query just particular rows from a table?

→ In this case you need to use the WHERE clause in the SELECT Statement.

Syntax

```
SELECT Column_1, Column_2 ..... Column_n
FROM table_name
WHERE Conditions;
```

Conditions - operators =>

=
>
<
>=
<=
<> or != not equal

AND
OR

Eg:- Want to get all first Customer name = Jamie

```
SELECT last_name, first_name
FROM Customer
WHERE first_name = 'Jamie';   -> Just first name
```

```
WHERE first_name = 'Jamie' AND last_name = 'Rice';
                    ↳ Both first & last name
```

```
SELECT customer_id, amount, payment_date,
FROM payment
WHERE amount <= 1 or amount >= 8;
```

} Rental amount less than 1 & greater than 8 $

```
SELECT * FROM payment
WHERE amount = 7.99;
```

} where rows with amount is 7.99

```
WHERE amount <= 4.99;  ---> less than/Equal to 4.99
```

AND = Both Conditions
OR = Either Condition

3.4 COUNT Statement

→ The COUNT function returns the no of input rows that match a specific condition of a query.

Syntax

"SELECT COUNT(*) FROM table;"

→ The COUNT(*) function returns the NO of rows returned by a SELECT clause.

→ When you apply the COUNT (*) to the Entire table, Postgress SQL scans table sequentially.

→ You can also specify a specific column Count for readability.

SELECT COUNT(column) FROM table;

→ It does not include NULL values

→ We can use COUNT with DISTINCT,

Eg: SELECT COUNT (DISTINCT column) FROM table;

Eg:-

→ No of Rows.

```
SELECT COUNT(*) FROM payment;
```

```
SELECT COUNT(DISTINCT amount)
FROM payment;
```

→ distinct Count Values of amount

3.5 LIMIT Statement

→ Limit allows you to limit the No of rows you get back after a query
→ Useful when wanting to get all columns but not all rows
→ Goes at the End of Query.

```
SELECT * FROM customer
LIMIT 5;
```

Just 5 rows

3.6

ORDER BY Statement

→ When you query data from a table, PostgreSQL returns the rows in the order that they ~~~~ were inserted into the table.

→ In order to sort the result set, you use the ORDER BY clause in SELECT ~~~~

→ The ORDER BY clause allows you to sort the rows returned from the ~~~~, SELECT statement in ascending or descending order based on criteria specified.

Syntax

```
SELECT Column_1, Column_2
FROM table_name
ORDER BY Column_1 ASC/DESC;
```

→ If you leave it blank, the ~~~~ ORDER BY clause will use ASC by default.

Eg:

```
SELECT first_name, last_name, FROM customer
ORDER BY first_name. ASC;
```

} ascending order first name

```
SELECT first_name, last_name FROM customer
ORDER BY first_name ASC, last_name. DESC;
```

↳ orders by first_names ascending then last_names descending order keeping first name in ascending order.

```
SELECT first_name FROM customer
ORDER BY last_name
```

} only in Postgres.

↑

other SQL. they don't allow above

So, always go for basic format (upabove one)

3-7

BETWEEN Statement

→ We use BETWEEN operator to match a value against a range of values.

Syntax

Value BETWEEN low AND high;

↓ similar to below

Value >= low and Value <= high

→ Value NOT BETWEEN low AND high

Eg:-

```
SELECT cust_id, amount FROM payment
WHERE amount BETWEEN 8 AND 9;
```

amount in 8 & 9

NOT BETWEEN → Not between 8 & 9

```
SELECT amount, payment_date FROM payment
WHERE payment_date BETWEEN '2007-02-07' AND
'2007-02-15'
```

3.8

IN Statement

→ You can use the IN operator with WHERE Clause to check if a value matches any value in a list of values.

→ The Syntax of the IN operator is as follows:

Value IN (Value 1, Value 2, ...)

Subquery

Value IN (SELECT value FROM table name)

Eg:-

```
SELECT Cust_id, rental_id, return_date
FROM rental
WHERE Cust_id IN (1, 2)
ORDER BY return_date DESC;
```

```
SELECT * FROM payment
WHERE amount IN (7.99, 8.99);
```

3-9

LIKE Statement

→ Syntax

```
SELECT first_name, last_name
FROM Customer
WHERE first_name LIKE 'Jen%';
```

% & _ => are called wildcard character.

% => for matching any sequence of characters.

_ => for matching any single character

Eg:-

```
SELECT first_name, last_name.
FROM Customer
WHERE first_name LIKE '%y';   } Ending 'y'
```

LIKE '%er%'; } => Between 'er' in anywhere in the name

LIKE '_her%'; → _ = any single char

NOT LIKE

ILIKE => Not Case Sensitive

ILIKE 'Bar%';