SQL

1. INTRODUCTION TO SQL

Web Application Architecture

Front End

HTML, CSS, JavaScript,
jQuery, LESS, SASS,
Bootstrap, CoffeeScript
and more. Angular.js

Front End Developer
UI UX Designer..

Back End

PHP, Ruby, Python,
C#, Java, Node.js,
Swift and more

BackEnd Developers
Server side developers

Database

SQL

Database admins
Database developers

|		
- Manual Testing	- API Testing	- Database Tester
- Black Box Testing	- API Automation	- Automate Database
- Selenium		- Test Using JS
- Protractor		- JDBC

What is Data?
- Piece of information
- For ex. Bank account:
    - Account num
    - Account type
    - User firstname ...
- All above data needs to be stored somewhere, where **it is secure, easy to ready, fast to read, easy and fast to update**
- **In databases** we store data in an organized manner.

What is Database?
- Database is a systematic collection of data. Databases support storage and manipulation of data. Databases make data management easy.
- Data is stored into separate tables that are related to each other
- Database tables is similar to excel, webtable and consists of Columns and Rows.

What is Primary Key?	What is Foreign Key?

● It is unique column in every table in a database
● It can ONLY accept;
    ○ non-duplicate values
    ○ cannot be NULL
	● It is a column that comes from a different table and using Foreign key tables are related each other
	● It is the primary key of another table
	● It can be duplicate or null for another table

Database Schema	
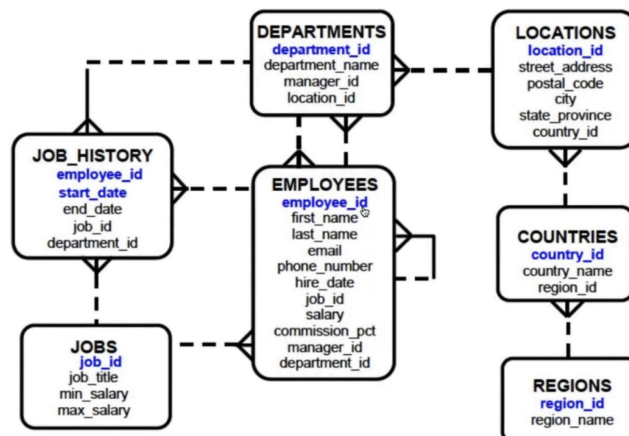
What is SQL?
- Structured Query Language
- It is a language that is used to work with Databases and manipulate data.

What is RDBMS?
- Relational Database Management System
- Relational Database → tables are related to each other using Primary and Foreign Keys
- All RDBMS are using SQL language.

2. SQL STATEMENT FUNDAMENTALS

- One of the most common tasks is to query data from tables by using the SELECT statement
- It has many clauses that you can combine to form a powerful query

SELECT STATEMENT
- Syntax →   **SELECT** column1,column2...
               FROM table_name;
- First, we specify a list of columns in the table from which we want to query data in the **SELECT** statement. We use a **comma** between each column in case we want to query data from multiple columns.
- If we want to query data from **all column**, we can use an **asterisk (*)** as the shorthand for all columns.
- Second, we indicate the **table name** after the **FROM** keyword
- SQL language is NOT case-sensitive

DISTINCT STATEMENT
- Syntax →   **SELECT DISTINCT** column1,column2...
               FROM table_name;
- We are trying to find that how many different type of value in the same column we have.

WHERE STATEMENT
- It is about what if we want to query (request) just particular rows from a table matching some sort of conditions.
- The **WHERE** clause appears right after the **FROM** clause of the **SELECT** statement.
- The conditions are used to **_filter the rows_** returned from the **SELECT** statement.
- PostgreSQL provides us with various standard operators to construct the conditions.
- Syntax →   **SELECT** column1,column2...
               FROM table_name
               WHERE conditions;
- We can use JS operator in the condition part;
  - =, >, <, >=, <=, <> or !=, AND, OR

```
1  SELECT first_name,last_name, email
2  FROM customer
3  WHERE first_name = 'Tracy' AND last_name='Cole';
```

Data Output   Explain   Messages   Notifications   Query History

first_name character varying (45)	last_name character varying (45)	email character varying (50)
1 Tracy	Cole	tracy.cole@sakilacusto...

```
   dvdrental on postgres@PostgreSQL 9.5
1  SELECT email
2  FROM customer
3  WHERE first_name = 'Tracy' OR last_name='Cole';
```

Data Output   Explain   Messages   Notifications   Query History

email character varying (50)
1 tracy.cole@sakilacustome...
2 tracy.herrmann@sakilacu...

If we want to know who paid the rental with amount is either less than $4 or greater than $8	**SELECT** * **FROM** payment **WHERE** amount<4 **OR** amount>8;
Different amount types greater than $5	**SELECT DISTINCT** amount **FROM** payment **WHERE** amount>5;
A customer forgot their wallet at our store. We need to track down their email to inform them. - What is the email for the customer with the name Jane Bennett?	**SELECT** email **FROM** customer **WHERE** first_name = 'Jane' **AND** last_name ='Bennett';
A customer wants to know what the movie "Angel's Life" is about. Could you give them the description for the movie "Angel's Life".	**SELECT** description **FROM** film **WHERE** title = 'Angels Life';
A customer is late on their movie return. I know their address is 604 Bern Place. I want to call them to let them know. Can you get me the phone number for the person who lives there.	**SELECT** phone **FROM** address **WHERE** address ='604 Bern Place';

COUNT STATEMENT
- The COUNT function returns the number of input rows that match a specific condition of a query.
- Similar to COUNT(*) function, the COUNT(column) function returns the number of rows returned by a SELECT clause.
- We can also use COUNT with DISTINCT.
  - However, it does not consider NULL values in the column.
- Syntax → **SELECT COUNT (*)**
  FROM table_name;

Please list how many different rental duration we have in our DVDRental Database	**SELECT COUNT(DISTINCT** rental_duration**)** **FROM** film**;**

ORDER BY STATEMENT
- The **ORDER BY** clause allows you to **_sort_** the rows returned from the SELECT statement in **ascending** or **descending** order based on criteria specified.
- If we sort the result set by multiple columns, use a **comma** to separate between two columns
- **ASC** → to sort the result set in ascending order (A-Z, 0-9)
- **DESC** → to sort the result set in descending order (Z-A, 9-0)
- If you leave it blank, the **ORDER BY** clause will use **ASC** by DEFAULT.
- Syntax → **SELECT COUNT (*)**
  FROM table_name;
  ORDER BY column_1 **ASC**/**DESC**;
-

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

LIMIT STATEMENT
- Allows you to limit the number of the rows you get back after a query.
- Useful when wanting to get all columns but not all rows.
- Goes at the end of a query.
- Syntax → **SELECT** column1,column2...
  FROM table_name;
  LIMIT number;

Get the customer ID numbers for the top 7 highest payment amounts	**SELECT** customer_id, amount **FROM** payment **ORDER BY** amount **DESC** **LIMIT** 7**;**

BETWEEN STATEMENT
- We use **BETWEEN** operator to match a value against a range of values.
- Syntax → value **BETWEEN** low **AND** high || value **NOT BETWEEN** low **AND** high
- If the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, or vice versa.
- We can rewrite the **BETWEEN** operator by using the greater than or **equal (>=)** or less than or **equal (<=)** operators as the following statement:
-

Provide me the list of first 5 movies where their replacement costs are between 15$ to $20 with title descending order	**SELECT** * **FROM** film **WHERE** replacement_cost **BETWEEN** 15 **AND** 20 **ORDER BY** title **DESC** **LIMIT** 5**;**

IN STATEMENT
- We use the IN operator with the WHERE clause to check if a value matches any value in a list of values.
  - Syntax → value **IN** (value1,value2…) || value **NOT IN** (value1,value2…)
  - The list of values is not limited to a list of numbers or strings but also a result of a SELECT statement as shown:
    value **IN** (**SELECT** value **FROM** table_name)
- **WHERE** customer_id = 5 **OR** customer_id=9 **OR** customer_id=13 === **WHERE** customer_id **IN**(5,9,13);

Display employee_id, first_name, last_name, salary for employees whose employee_id 101,105,117,123,125 order by salary high to low	**SELECT** employee_id, first_name, last_name, salary **FROM** employees **WHERE** employee_id **IN(**101,105,117,123,125**)** **ORDER BY** salary **DESC;**

LIKE

- ○ Suppose the store manager asks you find a customer that he does not remember the name exactly
- ○ He just remembers that customer's first name begins with something like Jen
- ○ We may find the customer in the customer table by looking at the first name column to see if there is any value that begins with Jen
- ○ Syntax →   **SELECT** column1,column2...
           FROM table_name
           WHERE conditions **LIKE** 'Jen**%** '
- ○

	SELECT first_name, last_name **FROM** customer **WHERE** first_name **LIKE** 'Jen**%'**; **--ILIKE** 'jen%'; **--> case INSENSITIVE** **--LIKE '%er'** **--> ends with er** **--NOT LIKE '%er'** **--> I don't want to see name which ends with er** **--LIKE '_er%'** **--> how many letter we expected before 'er'** **--LIKE '_ _ _ _ _'** **--> includes 5 characters**

3. FUNCTIONS in SQL

- **SINGLE ROW** FUNCTION**:** Function will run for <u>each</u> row and return a <u>value for each row</u>.
- **MULTIPLE ROW** FUNCTION**:** Function will run for <u>many</u> row and return a <u>single value</u>.

MATHEMATICAL FUNCTION

- SQL comes with a lot of math operators built-in that are very useful for numeric column types
- https://www.postgresql.org/docs/9.5/static/functions-math.htm

Operator	Description	Example	Result
+	addition	2 + 3	5
–	subtraction	2 – 3	–1
*	multiplication	2 * 3	6
/	division (integer division truncates the result)	4 / 2	2
%	modulo (remainder)	5 % 4	1
^	exponentiation (associates left to right)	2.0 ^ 3.0	8
|/	square root	|/ 25.0	5
||/	cube root	||/ 27.0	3
!	factorial	5 !	120
!!	factorial (prefix operator)	!! 5	120
@	absolute value	@ –5.0	5
&	bitwise AND	91 & 15	11
|	bitwise OR	32 | 3	35
#	bitwise XOR	17 # 5	20
~	bitwise NOT	~1	–2
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

```
1  SELECT first_name, salary*12 AS Annual_Salary
2  FROM employees;
```

Data Output   Explain   Messages   Notifications   Query History

	first_name character varying (20)	annual_salary numeric
1	Steven	288000.00
2	Neena	204000.00
3	Lex	204000.00

- ○ **SELECT** column_name **+,-,*,/** column_name **AS** new column **FROM** table

NOT: **SELECT** *
         FROM employees
         WHERE department_id **IS** NULL;

STRING FUNCTIONS

- https://www.postgresql.org/docs/9.5/static/functions-string.htm
  - Popular ones;

`string || string`	`text`	String concatenation	`'Post' || 'greSQL'`	`PostgreSQL`
`string || non-string or non-string || string`	`text`	String concatenation with one non-string input	`'Value: ' || 42`	`Value: 42`
`lower(string)`	`text`	Convert string to lower case	`lower('TOM')`	`tom`
`upper(string)`	`text`	Convert string to upper case	`upper('tom')`	`TOM`
`substring(string [from int] [for int])`	`text`	Extract substring	`substring('Thomas' from 2 for 3)`	`hom`
`substring(string from pattern)`	`text`	Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '...$')`	`mas`
`substring(string from pattern for escape)`	`text`	Extract substring matching SQL regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '%#"o_a#"_' for '#')`	`oma`
`initcap(string)`	`text`	Convert the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	`initcap('hi THOMAS')`	`Hi Thomas`
`length(string)`	`int`	Number of characters in `string`	`length('jose')`	`4`
`length(string bytea, encoding name )`	`int`	Number of characters in `string` in the given `encoding`. The `string` must be valid in this encoding.	`length('jose', 'UTF8')`	`4`

```
3   SELECT first_name,last_name,first_name||' '||last_name AS fullname
4   FROM employees;
```

Data Output Explain Messages Notifications Query History

	first_name character varying (20)	last_name character varying (25)	fullname text
1	Steven	King	Steven King
2	Neena	Kochhar	Neena Kochhar

Display uppercase first name, lowercase last name, initcap email, length of phone number from employees table	**SELECT UPPER(**first_name**), LOWER(**last_name**), INITCAP(**email**), LENGTH(**phone_number**),** first_name **||' '||**last_name**||' '||**email**||' '||**phone_number **AS** full_info **FROM** employees**;**
Create a password for each employee that consists of first 3 letter of first and last name	**SELECT SUBSTRING (**first_name,1,3**) || SUBSTRING(**last_name,1,3**) AS** password **FROM** employees**;**

AGGREGATE FUNCTION

- **MIN (**Check all the rows and shows minimum one)
  - **SELECT MIN**(amount)
    FROM payment;

What is the lowest length of the film?	**SELECT MIN**(length) **FROM** film;	**SELECT** length **FROM** film **ORDER BY** length **ASC** **LIMIT** 1;

- **MAX (**Check all the rows and shows maximum one)
  - **SELECT MAX**(amount)
    FROM payment;
- **AVG** (Add all rows and get average)
  - **SELECT AVG**(amount)
    FROM payment;
- **SUM** (Add all rows and shows SUM)
  - **SELECT SUM**(amount)
    FROM payment;
- **ROUND** (The result with given decimal)
  - **SELECT ROUND(AVG**(amount)**, 2)**   --2→ how many decimal we want to see
    FROM payment;

- **GROUP BY**
  - It divides the rows returned from the SELECT statement into groups
  - For each group, we can apply an aggregate function, for example:
    - calculating the sum of items
    - count the number of items in the group
  - Syntax → **SELECT** column_1, **aggregate_function(column_2)**
    FROM table_name
    GROUP BY column_1;

How many films we have for each rating type? Order by desc by the number	**SELECT** rating**, COUNT(*)** **FROM** film **GROUP BY** rating **ORDER BY** count **DESC;**

- **HAVING**
  - We often use the HAVING clause in conjunction with the GROUP BY clause to filter group rows that don't satisfy a specified condition.
  - Difference between HAVING and WHERE:
    - HAVING statement sets the condition for *group rows* created by the GROUP BY clause *after* the GROUP BY clause applies while
    - WHERE clause sets the condition for *individual rows before* GROUP BY clause applies.
  - It comes after GROUP BY and before ORDER BY!!!
  - Syntax → **SELECT** column_1, **aggregate_function(column_2)**
    FROM table_name
    GROUP BY column_1
    HAVING condition;

4. JOIN
- Suppose we want to get data from **Address** and **Customer** tables. The Customer table has the address_id field that relates to the **primary key** of the Address table.
- If we have same column name in two tables, we need to specify each of them like "**customer.first_name".**
- **FROM table_name → table_name must have only one column which is not in the other table**
- **JOIN table_name → BOTH tables has the same column_name which is the primary key of one table**
- **INNER JOIN === JOIN**

I want to see customer_id, first_name, last_name, email, phone, address in ONE TABLE. (in customer table, there is no address and phone info. we should get these infos from address table with **JOIN** and connection point is **ON** with address_id)	**SELECT** customer_id, first_name, last_name, email, phone, address **FROM** customer **JOIN address** **ON** customer.**address_id = address**.**address_id;** 　　　foreign key　　　　　　　primary key **SELECT** customer_id, first_name, last_name, email, phone, address **FROM** customer, address **WHERE** customer.**address_id =** address.**address_id;** **--we can rename the table to make short** **SELECT** c.customer_id, first_name, last_name, email, phone, address **FROM** customer **AS** c **JOIN** address **ON** c.**address_id =** address.**address_id;**
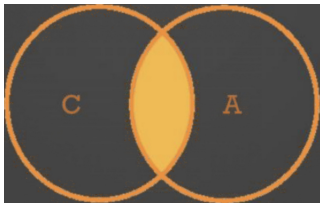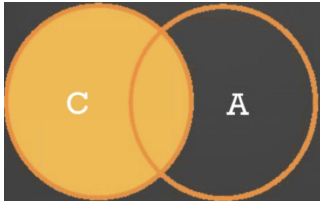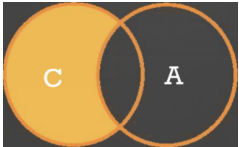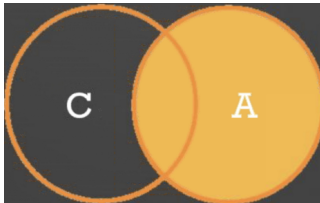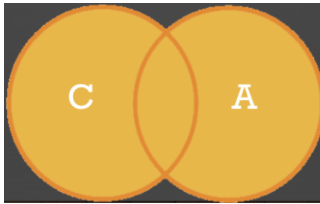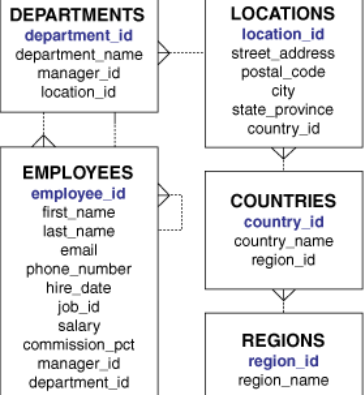
CUSTOMER TABLE				ADDRESS TABLE		
	first_name character vary	last_name character varyi	address_id integer		address character varying (50)	phone integer

	customer_id integer	first_name character vary	last_name character varyi	address_id integer		address_id integer	address character varying (50)	phone integer
1	1	Mary	Smith	5	1	5	1913 Hanoi Way	28303384
2	2	Patricia	Johnson	[null]	2	7	692 Joliet Street	44847719
3	3	Linda	Williams	7	3	8	1566 Inegl Manor	70581400
4	4	Barbara	Jones	8	4	10	1795 Santiago	86045262
5	5	Elizabeth	Brown	[null]	5	11	900 Santiago	16571220

INNER JOIN

It produces only the set of records that match in both Table Customer and Table Address


```
SELECT customer_id,first_name,last_name,address,phone
FROM customer INNER JOIN address
ON customer.address_id = address.address_id;
```

	customer_id integer	first_name character va	last_name character var	address character varying (5	phone integer
1	1	Mary	Smith	1913 Hanoi Way	28303384
2	3	Linda	Williams	692 Joliet Street	44847719
3	4	Barbara	Jones	1566 Inegl Manor	70581400

LEFT OUTER JOIN

It produces a complete set of records from Table Customer, with matching records (where available) in Table Address. If there is no match, the right side will contain null.


```
SELECT customer_id,first_name,last_name,address,phone
FROM customer LEFT OUTER JOIN address
ON customer.address_id = address.address_id;
```

	customer_id integer	first_name character var	last_name character var	address character varying (5	phone integer
1	1	Mary	Smith	1913 Hanoi Way	28303384
2	2	Patricia	Johnson	[null]	[null]
3	3	Linda	Williams	692 Joliet Street	44847719
4	4	Barbara	Jones	1566 Inegl Manor	70581400
5	5	Elizabeth	Brown	[null]	[null]


```
SELECT customer_id,first_name,last_name,address,phone
FROM customer LEFT OUTER JOIN address
ON customer.address_id = address.address_id
WHERE customer.address_id IS NULL;
```

RIGHT OUTER JOIN

It produces a complete set of records from Table Address, with matching records (where available) in Table Customer. If there is no match, the right side will contain null.


```
SELECT customer_id,first_name,last_name,address,phone
FROM customer RIGHT OUTER JOIN address
ON customer.address_id = address.address_id;;
```

	customer_id integer	first_name character va	last_name character var	address character varying (5	phone integer
1	1	Mary	Smith	1913 Hanoi Way	28303384
2	3	Linda	Williams	692 Joliet Street	44847719
3	4	Barbara	Jones	1566 Inegl Manor	70581400
4	[null]	[null]	[null]	900 Santiago	16571220
5	[null]	[null]	[null]	1795 Santiago	86045262

FULL OUTER JOIN


```
SELECT customer_id,first_name,last_name,address,phone
FROM customer FULL OUTER JOIN address
ON customer.address_id = address.address_id;
```

	customer_id integer	first_name character var	last_name character vary	address character varying (50	phone integer
1	1	Mary	Smith	1913 Hanoi Way	28303384
2	2	Patricia	Johnson	[null]	[null]
3	3	Linda	Williams	692 Joliet Street	44847719
4	4	Barbara	Jones	1566 Inegl Manor	70581400
5	5	Elizabeth	Brown	[null]	[null]
6	[null]	[null]	[null]	900 Santiago	16571220
7	[null]	[null]	[null]	1795 Santiago	86045262

```
SELECT customer.* --if I want to see just all customer table columns
FROM customer FULL OUTER JOIN address
ON customer.address_id = address.address_id;
```

	customer_id integer	first_name character var	last_name character var	address_id integer
1	1	Mary	Smith	5
2	2	Patricia	Johnson	[null]
3	3	Linda	Williams	7
4	4	Barbara	Jones	8
5	5	Elizabeth	Brown	[null]
6	[null]	[null]	[null]	[null]
7	[null]	[null]	[null]	[null]

THREE TABLE JOIN

<table>
<tr><td>

DEPARTMENTS
department_id
department_name
manager_id
location_id

EMPLOYEES
employee_id
first_name
last_name
email
phone_number
hire_date
job_id
salary
commission_pct
manager_id
department_id

LOCATIONS
location_id
street_address
postal_code
city
state_province
country_id

COUNTRIES
country_id
country_name
region_id

REGIONS
region_id
region_name

</td><td>

TASK: Display each employees first_name, last_name, department_name and city

SELECT first_name, last_name, department_name,city
FROM employees **AS** e **INNER JOIN** departments **AS** d
ON e.department_id = d.department_id
INNER JOIN locations **AS** l
ON l.location_id = d.location_id;

	first_name character varying (20)	last_name character varying (25)	department_name character varying (30)	city character varying (30)
1	Steven	King	Executive	Seattle
2	Neena	Kochhar	Executive	Seattle
3	Lex	De Haan	Executive	Seattle
4	Alexander	Hunold	IT	Southlake
5	Bruce	Ernst	IT	Southlake

</td></tr>
</table>

5. ADVANCED SQL COMMENTS

SET OPERATORS (UNION, UNION ALL, INTERSECT, EXCEPT**)**
- We need 2 independent queries
- SAME number of columns in Select statement and SAME data type in same order

  ○ **UNION & UNION ALL**
    ■ We use UNION operator to combine data from similar tables that are not perfectly normalized.
    ■ It combines result set of two or more SELECT statements into a single result set.
    ■ It removes all DUPLICATE rows unless the UNION ALL used.

    <table>
    <tr><td>

    Syntax →
    SELECT column1,column2...
    FROM table_name;
    UNION
    SELECT column1,column2...
    FROM table_name;

    </td><td>

    </td></tr>
    </table>
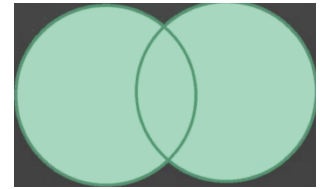
  ○ **EXCEPT (Oracle → MINUS)**
    ■ It only returns records/values from first query that is not present in second query
    ■ Help us to find difference between two queries.

    <table>
    <tr><td>

    Syntax →
    SELECT column1
    FROM table_name1;
    EXCEPT
    SELECT column1
    FROM table_name2;

    </td><td>

    </td></tr>
    </table>

  ○ **INTERSECT**
    ■ It returns records that are present / common/ appear in both query results
    ■ It will sort and remove duplicates.

    <table>
    <tr><td>

    Syntax →
    SELECT column1
    FROM table_name1;
    INTERSECT
    SELECT column1
    FROM table_name2;

    </td><td>

    </td></tr>
    </table>
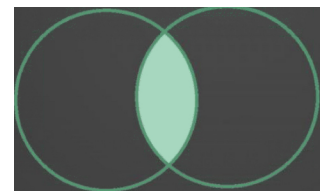
SELF JOIN
- There is a special case that we join a table to itself, which is known as self join.
- Let's say we want to print out employee full name with their managers last together.
-

SUBQUERY
- It allows us to use **multiple SELECT** statements, where we basically have a query within a query.
- Suppose we want to find the films whose rental rate is higher than the average rental rate.
  - 1st step → find the AVG rental_rate by using SELECT statement
  - 2nd step → use the result of the 1st query in the 2nd SELECT statement to find the films that we want
    - **SELECT** title
      FROM film
      WHERE rental_rate > (**SELECT AVG**(rental_rate) **FROM** film)
      2nd query

6. CREATING DATABASE and TABLES

DATA TYPES
- Boolean / bool
  - true   → 1, yes, y, t, true
  - false   → 0, no, n, f, false
  - NULL   → space character
- Character
  - char   → single character
  - char(n) → fixed-length character strings (if we insert a string that is shorter than the length of the column)
  - varchar(n) → variable-length character strings (store up to n character)
- Number
  - integers
    - Small integers (smallint) → 2 byte
    - Integer (int) → 4 byte
    - Big Integer (bigint) → 8 byte
    - Serial is the same as integer except that PostgreSQL populate value into column automatically
  - floating-point numbers
- Temporal (i.e. date and time-related data types)
- Special types

PRIMARY and FOREIGN KEY
- Primary: A table can have only one primary key. When we add a primary key to a table, PostgreSQL creates unique index.
- We define the primary keys through **primary key** constraints
  - Syntax →   **CREATE TABLE** table_name(
                     column_name **data_type primary key**)
- Foreign: it is defined in a table that refers to the primary key of the other table.

DDL & DML
- DDL: Data Definition Language   DML: Data Manipulation Language
  It is used to define data structures   It is used to manipulate data itself
  CREATE, DROP, TRUNCATE, ALTER   SELECT, INSERT, UPDATE, DELETE

- **CREATE TABLE**
  - First, we specify the name of the new table alter the CREATE TABLE clause.
  - Next, we list the column name, its data type, and column constraint.
  - PostgreSQL column <u>Constraints</u>;
    - **NOT NULL** - the value of the column cannot be NULL
    - **UNIQUE** - the value of the column must be unique across the whole table
    - **PRIMARY KEY** - it is the combination of NOT NULL and UNIQUE constraints

Syntax → **CREATE TABLE** table name 　　　　**(**column name **TYPE** column constraint**);**	**CREATE TABLE** students**(** student_id **serial primary key**, first_name **varchar(30) not null**, last_name **varchar(30) not null**, phone_number **bigint null);**

- **INSERT**
  - After creating a new table, we insert new rows into the table.

Syntax → **INSERT INTO** table(column1,column2) 　　　　**VALUES** (value1,value2,..), (value1,value2,..);	**INSERT INTO** students(first_name, last_name, phone_number) **VALUES** ('Mike', 'Smith', 9739739739);

- **UPDATE**
  - To change the values of the columns

UPDATE table_name **SET** column1 = value1, 　　column2 = value2 , … **WHERE** condition;	**UPDATE** students **SET** first_name = 'Jamal' **WHERE** student_id =1;

- **DELETE**
  - To delete rows in a table. (If you omit the WHERE clause, all rows in the table are deleted)
  - DELETE statement returns the number of rows deleted. If no rows are deleted, it returns zero.

DELETE FROM table_name **WHERE** condition;	**DELETE FROM** students **WHERE** student_id = 1;

- **ALTER**
  - To change existing table structure, you use ALTER TABLE statement.
  - PostgreSQL provides many actions that allow you to:
    - Add, remove, or rename column. (**ADD COLUMN, DROP COLUMN, RENAME COLUMN**)
    - Set default value for the column.
    - Add CHECK constraint to a column. (**ADD CONSTRAINT**)
    - Rename table (**RENAME TO**)

ALTER TABLE table_name **action**;	**ALTER TABLE** students **RENAME TO** st; **ALTER TABLE** st **RENAME COLUMN** phone_number **TO** phone; **ALTER TABLE** st **DROP COLUMN** phone;

- **DROP**
  - To **remove** existing table from the database, you use the DROP TABLE statement as shown following:

DROP TABLE table_name;	**DROP TABLE** st;

- **TRUNCATE**
  - Truncating will **remove** all data from the table <u>but not delete the table</u>. Lastly we can DROP TABLE.

TRUNCATE TABLE table_name;	**TRUNCATE TABLE** st;

7. DATABASE
- **How to get data from and how to pass to the website?**

POSTGRESQL CONNECTION
- We will use pg-promise modules to create connection between PostgreSQL server and NodeJS
- Installation→ to terminal → npm install pg-promise (for mac use sudo)
  - **ONE ROW**

create **spec.js** file (VISUAL STUDIO CODE)	**TO REACH SPECIFIC RESULTs**
| ```javascript
var pgp = require('pg-promise')(/*options*/);

//then we will create our Connection String
var cn = {
   host: 'localhost',      //database host
   port: 5432,             //port number
   database: 'dvdrental',  //database name
   user: 'postgres',       //database username
   password: 'abc'         //database password
}

//then creating database connection
var db = pgp(cn);   //cn-->Connection String object

//then we will use db to query data from database
//RETRIEVE ONE ROW --> db.one method
db.one(`SELECT title FROM film WHERE film_id=133`)
   .then(result=>{
       console.log(result) //printing the result
   })
   .catch(error=>{
       console.log(error); //printing the error
   })

//when we run node spec.js on terminal we will see;
{ title: 'Chamber Italian' }
``` | ```javascript
//to reach title directly → result.title
db.one(`SELECT title FROM film WHERE
film_id=133`)
   .then(result=>{
       console.log(result.title)
   })
   .catch(error=>{
       console.log(error);
   })

//TERMINAL → node spec.js
RESULT:
Chamber Italian
```
<br>
```javascript
//to reach all infos with *
db.one(`SELECT * FROM film WHERE
film_id=133`)
   .then(result=>{
       console.log(result)
   })
   .catch(error=>{
       console.log(error);
   })

//TERMINAL → node spec.js
RESULT:
{ film_id: 133,
  title: 'Chamber Italian',
  description:
   'A Fateful Reflection of a Moose And a
Husband who must Overcome a Monkey in
Nigeria',
  release_year: 2006,
  language_id: 1,
  rental_duration: 7,
  rental_rate: '4.99',
  length: 117,
  replacement_cost: '14.99',
  rating: 'NC-17',
  last_update: 2013-05-26T18:50:58.951Z,
  special_features: [ 'Trailers' ],
  fulltext:
   '\'chamber\':1 \'fate\':4 \'husband\':11
\'italian\':2 \'monkey\':16 \'moos\':8
\'must\':13 \'nigeria\':18 \'overcom\':14
\'reflect\':5' }
``` |

TASK: Print first_name and last_name of who made highest total payment in dvdrental

```javascript
var max = `SELECT first_name, last_name FROM customer WHERE customer_id = (SELECT
customer_id FROM payment GROUP BY customer_id ORDER BY SUM(amount) DESC LIMIT 1)`
db.one(max)
   .then(result=>{
       console.log(result.first_name +' '+ result.last_name);
   })
   .catch(error=>{
       console.log(error);
   })
```

- ○ **MULTIPLE ROWS**
  - **TASK:** Print first_name and last_name of whose customer_id is 3,4,5,6,7

```
for(var i=3;i<8;i++){
var max = `SELECT first_name,last_name
FROM customer WHERE customer_id
IN('${i}')`
db.one(max)
    .then(result=>{
        console.log(result.first_name +'
'+ result.last_name);
    })
    .catch(error=>{
        console.log(error);
    })
}
```

```
INSTEAD OF FOR LOOP → we use db.any → ARRAY

db.any(`SELECT first_name,last_name FROM customer
WHERE customer_id IN(3,4,5,6,7)`)
    .then(result=>{
        console.log(result);
    })
    .catch(error=>{
        console.log(error);
    })
}
RESULT:
[ { first_name: 'Linda', last_name: 'Williams' },
  { first_name: 'Barbara', last_name: 'Jones' },
  { first_name: 'Elizabeth', last_name: 'Brown' },
  { first_name: 'Jennifer', last_name: 'Davis' },
  { first_name: 'Maria', last_name: 'Miller' } ]
```

```
db.any(`SELECT first_name,last_name FROM customer
WHERE customer_id IN(3,4,5,6,7)`)
    .then(result=>{
        result.forEach(element => {
            console.log(element.first_name + ' ' +
element.last_name)
        });
    })
    .catch(error=>{
        console.log(error);
    })
RESULT:
Linda Williams
Barbara Jones
Elizabeth Brown
Jennifer Davis
Maria Miller
```

```
//to reach one information directly
db.any(`SELECT email FROM customer LIMIT 5`)
    .then(result=>{
        console.log(result[0].email);
        });
    })
    .catch(error=>{
        console.log(error);
    })
```

CONNECTING DATABASE WITH PROTRACTOR
- **AMAZON TASK**

We want to search movie names in amazon.com where their titles start with Z.

conf.js file	dbtest.js file

```javascript
exports.config ={
  framework:'jasmine',
  directConnect:true,
  specs:['dbtest.js'],
  jasmineNodeOpts:{
    defaultTimeoutInterval:50000
  }
}
```

```javascript
describe ('Amazon Database Testing', function(){
  var pgp = require('pg-promise')(/*options*/);
  var cn = {
    host: 'localhost',        //database host
    port: 5432,               //port number
    database: 'dvdrental',    //database name
    user: 'postgres',         //database username
    password: 'abc'           //database password
  }
  var db = pgp(cn);    //cn-->Connection String object

  var arr = [];

  it ('Should get the film names from database and search in amazon.com',()=>{
    browser.waitForAngularEnabled(false);
    browser.get('https://www.amazon.com');

    db.any(`SELECT title FROM film WHERE title LIKE 'Z%'`)
      .then(result=>{
        arr=result;
      })
      .catch(error=>{
        console.log(error);
      })
      .then(()=>{
        arr.forEach(function(i){
          element(by.id(id="twotabsearchtextbox")).sendKeys(i.title);
          browser.sleep(2000);
          element(by.css("#nav-search>form>div.nav-right>div>input")).click();
          browser.sleep(2000);
          browser.navigate().back();
        });
      })
  })
})
```