
This material really helps you to practice and understand the basic psql.Kindly follow step by step for practice.

Before practicing PSQL,first get to know about PSQL.

# PSQL:

Postgre SQL is most advanced database management system which is widely used for handling large and complex datasets.

PSQl is the **open source** and **Object Relational Database Management System**(ORDBMS).

PSQL is the first DBMS which supports **MVCC**(Multi-Version Concurrency Control}feature.

PSQL was designed by the **C** programming language.

PSQL supports both SQL and **JSON** for relational and non-relational queries for extensibility and SQL compliance.

PostgreSQL is the cross platform that runs on various operating systems such as Windows,Unix,Linux,Mac OS,Solaris and so on.

The PostgreSQL is the existing database for *the* macOS server

PostgreSQL will offer us the facility to add custom functions with the help of various programming languages such as Java,c++ and C,.

The primary objective of PostgreSQL is to handle a variety of jobs from single technologies to web service or the data warehouse with several parallel users.

# Why PSQL:

There are number of popular database available like MYSQL,MangoDB,Oracle,Microsoft SQL server,etc.Since why we should go for PSQL and what speaciality comparitively between them.So,first we discuss about its features by knowing the features we might get to know.

**PostgreSQL Features:**

PSQL is one of the most popular databases supporting JSON (non-relational) queries and SQL for (relational) queries.

PSQL contains the various advanced data types and robust feature sets, which increase the extensibility, reliability, and data integrity of the software.

PSQL is a multi-model database supporting Spatial Data, Key-Value, Structured Data (SQL), and Semi-Structured Data (JSON, XML)

It is open-source, and we can easily download it from the official website of PostgreSQL.

**Compatible with Data Integrity:** It supports data integrity which includes the following:

- Primary Keys
- UNIQUE, NOT NULL
- Foreign Keys
- Explicit Locks, Advisory Locks
- Exclusion Constraints

**Support multiple features of SQL:** PostgreSQL supports various features of SQL which include the followings:

- MVCC (Multi-Version Concurrency Control).
- It supports multiple Indexing such as Multicolumn, Partial, B-tree, and expressions.
- SQL sub-selects.
- Complex SQL queries.
- Streaming Replication
- It supports transactions, Nested Transactions through Savepoints.
- Just-in-time compilation of expressions
- Table partitioning

**Compatible with multiple data types:** PostgreSQL support various data types such as:

- *Structured:* Array, Date and Time, UUID (Universally Unique Identifier), Array, Range.
- *Primitives:* String, Integer, Boolean, Numeric.
- *Geometry:* Polygon, Circle, Line, Point,
- *Document:* XML, JSON/JSONB, Key-value.

**Highly extensible:** PostgreSQL is highly extensible in several phases which are as following:

- It supports procedural Languages such as Perl, PL/PGSQL, and Python, etc.
- JSON/SQL path expressions
- Stored procedures and functions.
- For tables, it supports a customizable storage interface.
- It is compatible with *foreign data wrappers*, which connect to further databases with a standard SQL interface.

**Secure:** It is safe because it follows several security aspects

- PostgreSQL provides a robust access control system.
- PostgreSQL supports Column and row-level security.

**Highly Reliable:** It is highly reliable and also provide disaster recovery such as:

- Active standbys, PITR (Point in time recovery)
- It supports WAL (Write-ahead Logging)
- Tablespaces
- It supports different types of Replication like Synchronous,Asynchronous and Logical

PostgreSQL supports Internationization, which means that the international character sets include ICU collations, accent- insensitive and case-sensitive collations, and full-text searches.

In PostgreSQL, a table can be set to inherit their characteristics from a "parent" table.

It is compatible with ANSI-SQL2008. PostgreSQL will help us to improve the functionality of Server-Side programming.

# Procedural Languages Support

PostgreSQL supports four standard procedural languages, which allows the users to write their own code in any of the languages and it can be executed by PostgreSQL database server. These procedural languages are - **PL/pgSQL, PL/Tcl, PL/Perl and PL/Python**. Besides, other non-standard procedural languages like PL/PHP, PL/V8, PL/Ruby, PL/Java, etc., are also supported



Now,step into the practice.

1.Open the terminal.I have already installed postgresql,So,i directly connect to the database.

```
To connect psql,command---> postgres~$ sudo -i -u postgres
```

**Next command line ask you enter password to verify admin**

```
[sudo] password for ziavu: Type password
```

**To see list of database in psql,command--->\l**

```
                List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 moviedb   | postgres | UTF8     | en_IN   | en_IN |
 payilagam | postgres | UTF8     | en_IN   | en_IN |
 postgres  | postgres | UTF8     | en_IN   | en_IN |
 template0 | postgres | UTF8     | en_IN   | en_IN | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_IN   | en_IN | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(5 rows)
```

I have already created list of database.I will create new database here to practice.

**To create new database,command--->create database institute;**

```
CREATE DATABASE
Institute is the name of the database and the institute database is created.We
can see that below
```

```
\l
                              List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 institute | postgres | UTF8     | en_IN   | en_IN |
 moviedb   | postgres | UTF8     | en_IN   | en_IN |
 payilagam | postgres | UTF8     | en_IN   | en_IN |
 postgres  | postgres | UTF8     | en_IN   | en_IN |
 template0 | postgres | UTF8     | en_IN   | en_IN | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_IN   | en_IN | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(6 rows)
```

**To connect the institute database.Type--->\c institute**

```
postgres=# \c institute;
You are now connected to database "institute" as user "postgres".
```

**If we want to know the version of database.**
**command--->select version():**

```
                                                                        version
-------------------------------------------------------------------------------------------------
-----------------------------------------------------
 PostgreSQL 14.7 (Ubuntu 14.7-0ubuntu0.22.04.1) on x86_64-pc-linux-gnu, compiled
by gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, 64-bit
(1 row)
```

*I am using 14.7 version*

We can also see the current date and time here by using command.

**c**ommand---> **select current_date,current_time;**

```
 current_date |     current_time
--------------+----------------------
 2023-04-05   | 15:09:27.04801+05:30
(1 row)
```

**To create table with column:**

```
--->institute=# create table candidate(name varchar(50),role char(20),salary
int,joining_date date,email_id varchar(30));
CREATE TABLE
Table is created
Name,role,salary,joining_date,email_id are metadata
```

First we need to know about datatype in psql,There are several datatype in psql but we should know the basic and common datatype that will enough to start as a beginner.**Integer,date,char,varchar,time,point,double** are the datatype in psql. You can refer this below website to know the various datatypes and its uses.
https://www.geeksforgeeks.org/postgresql-data-types/

Now,we shall insert the data into the table,when inserting the data,it should be corresponding to the metadata

To insert--->

**insert into candidate values ('mohammed ziavudeen','software architecture',30000,'01/05/2023','mdziavu443@gmail.com');**

```
        name        |         role          | salary | joining_date |
email_id
--------------------+-----------------------+--------+--------------
+----------------------
```

```
 mohammed ziavudeen | software architecture |  30000 | 2023-05-01   |
mdziavu443@gmail.com
(1 row)
Succesfully inserted 1 column.
Now, we shall insert multiple column data

insert into candidate values('saravanan','sql
developer','35000','21/04/2023','saro123@gmail.com');
INSERT 0 1
insert into candidate values('sasi','team
lead','29000','23/04/2023','sasi123@gmail.com');
INSERT 0 1
 insert into candidate
values('dhinakaran','manager','50000','31/03/2023','dhinakaran123@gmail.com');
INSERT 0 1
insert into candidate
values('manjunathan','tester','40000','20/04/2023','manju1123@gmail.com');
INSERT 0 1


We can also insert the mutiple rows by single query
To insert multiple row by single query--->
insert into candidate values('benito','software
architecture',40000,'04/09/2023','benito@gmail.com'),('kannan','sql
developer',45000,'10/04/2021','kanna@gmail.com');
INSERT 0 2
Now we see the inserted columns.




To see to the data and metadata---> select * from candidate;
         name        |         role          | salary | joining_date |
email_id
--------------------+----------------------+--------+--------------
+-------------------------
 mohammed ziavudeen | software architecture |  30000 | 2023-05-01   |
mdziavu443@gmail.com
 manjunathan        | tester                |  40000 | 2023-04-20   |
manju1123@gmail.com
 dhinakaran         | manager               |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
 sasi               | team lead             |  29000 | 2023-04-23   |
sasi123@gmail.com
 saravanan          | sql developer         |  35000 | 2023-04-21   |
saro123@gmail.com
 thandavamoorthy    | web developer         |  35000 | 2023-04-02   |
moorthy@gmail.com
 benito             | software architecture |  40000 | 2023-09-04   |
benito@gmail.com
 kannan             | sql developer         |  45000 | 2021-04-10   |
kanna@gmail.com
(8 rows)

If we need to see all the rows and columns present in the table,usually in sql
there is a character asterisk(*).
*is used to retrieve all columns and rows from the table. Basically we can see
the entire table.
Query--->select * from candidate;
        name        |         role          | salary | joining_date |
email_id
```

```
-------------------+----------------------+--------+-------------
+-------------------------
 mohammed ziavudeen | software architecture |  30000 | 2023-05-01   |
mdziavu443@gmail.com
 manjunathan        | tester                |  40000 | 2023-04-20   |
manju1123@gmail.com
 dhinakaran         | manager               |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
 sasi               | team lead             |  29000 | 2023-04-23   |
sasi123@gmail.com
 saravanan          | sql developer         |  35000 | 2023-04-21   |
saro123@gmail.com
 thandavamoorthy    | web developer         |  35000 | 2023-04-02   |
moorthy@gmail.com
 benito             | software architecture |  40000 | 2023-09-04   |
benito@gmail.com
 kannan             | sql developer         |  45000 | 2021-04-10   |
kanna@gmail.com
(8 rows)
```

To view required rows.For,eq.name and role only

Query---> **select name,role from candidate;**

```
        name        |         role
--------------------+----------------------
 mohammed ziavudeen | software architecture
 manjunathan        | tester
 dhinakaran         | manager
 sasi               | team lead
 saravanan          | sql developer
 thandavamoorthy    | web developer
 benito             | software architecture
 kannan             | sql developer
(8 rows)
```

We can also change the table name as we want.
For eq.candidate to employee

query---> **alter table candidate rename to employee;**

```
ALTER TABLE
```

Table name would be altered.

SQL Statements can be categorized into five types
One of the type is **Data Retrieval or Data Query language** (DQL) - **SELECT**
SELECT---> is to retrieve data from the database.
We can apply select with various query,extracts data from a database.

To see all values without duplicate.we can use **distinct** keyword.
The distinct keyword is used with select keyword in conjunction. It is helpful when we avoid duplicate values present in the specific columns/tables.

query--->**select distinct role from employee;**

```
         role
----------------------
 manager
 tester
 sql developer
 web developer
 team lead
 software architecture
(6 rows)
```

**WHere clause**:
The WHERE clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

It is useful to get a data using another data.

**query--->select name from employee where role ='software architecture';**

```
        name
--------------------
 mohammed ziavudeen
 benito
(2 rows)
```

**Where and:**

If we need to satisfy 2 condition to get the data,we can use where and.

query--->**select name from employee where role ='software architecture' and salary =30000;**

```
        name
--------------------
 mohammed ziavudeen
(1 row)
```

**Where or:**

If we need to satisfy either 1 condition out of 1 condition to get the data,we can use where or.

Query---> **select name from employee where role ='software architecture' or salary =45000;**

```
        name
--------------------
 mohammed ziavudeen
 benito
 kannan
(3 rows
```

**as:**

To view the column name as we want

Query--->  **select name as employee_name from employee;**

```
   employee_name
--------------------
 mohammed ziavudeen
 manjunathan
 dhinakaran
 sasi
 saravanan
 thandavamoorthy
 benito
 kannan
(8 rows)
```

**Not equals to:**

**select name,role,salary from employee where not salary= 35000;**

```
        name        |         role          | salary
--------------------+-----------------------+--------
 mohammed ziavudeen | software architecture |  30000
 manjunathan        | tester                |  40000
 dhinakaran         | manager               |  50000
 sasi               | team lead             |  29000
 benito             | software architecture |  40000
 kannan             | sql developer         |  45000
(6 rows)
```

**Greater than:**

The SQL Greater Than comparison operator (>) is used to compare two values.It returns TRUE if the first value is greater than the second.If the second is greater, it returns FALSE.

You can also test for greater than or equal to by using >=.

To find something greater than it query---> **select name,role,salary from employee where salary> 35000;**

```
    name     |         role          | salary
```

```
-------------+----------------------+--------
 manjunathan | tester               |  40000
 dhinakaran  | manager              |  50000
 benito      | software architecture |  40000
 kannan      | sql developer        |  45000
(4 rows)
```

**Between**:
when applying between we always use and this is basic grammar,some of us may commit mistake on this even i too.
query---> **select name,role,salary from employee where salary between 15000 and 30000;**

```
  name        |         role          | salary
--------------------+----------------------+--------
 mohammed ziavudeen | software architecture |  30000
 sasi               | team lead            |  29000
(2 rows)
```

**Not:**
NOT is a logical operator in SQL that you can put before any conditional statement to select rows for which that statement is false.

**select * from employee where not role ='software architecture';**

```
      name       |     role      | salary | joining_date |        email_id
-----------------+---------------+--------+--------------
+-------------------------
 manjunathan     | tester        |  40000 | 2023-04-20   | manju1123@gmail.com
 dhinakaran      | manager       |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
 sasi            | team lead     |  29000 | 2023-04-23   | sasi123@gmail.com
 saravanan       | sql developer |  35000 | 2023-04-21   | saro123@gmail.com
 thandavamoorthy | web developer |  35000 | 2023-04-02   | moorthy@gmail.com
 kannan          | sql developer |  45000 | 2021-04-10   | kanna@gmail.com
(6 rows)
```

**in**:
The IN command allows us to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

**select * from employee where role in('web developer','team lead');**

```
      name       |     role      | salary | joining_date |     email_id
-----------------+---------------+--------+--------------+-------------------
 sasi            | team lead     |  29000 | 2023-04-23   | sasi123@gmail.com
 thandavamoorthy | web developer |  35000 | 2023-04-02   | moorthy@gmail.com
(2 rows)
```

As we see "and" and "or" command separately above,lets we apply both in single query.
institute=# **select * from employee where joining_date='2023/04/20' or role='manager';**

```
    name     |  role   | salary | joining_date |        email_id
-------------+---------+--------+--------------+-------------------------
 manjunathan | tester  |  40000 | 2023-04-20   | manju1123@gmail.com
```

```
 dhinakaran  | manager |  50000 | 2023-03-31   | dhinakaran123@gmail.com
(2 rows)
```

**not,in:**

NOT IN operator is used to replace a group of arguments using the <> (or !=) operator that are combined with an AND. It can make code easier to read and understand for SELECT, UPDATE or DELETE SQL commands(TBD).

**select * from employee where role not in ('software architecture','tester','manager');**

```
      name       |     role      | salary | joining_date |     email_id
-----------------+---------------+--------+--------------+------------------
 sasi            | team lead     |  29000 | 2023-04-23   | sasi123@gmail.com
 saravanan       | sql developer |  35000 | 2023-04-21   | saro123@gmail.com
 thandavamoorthy | web developer |  35000 | 2023-04-02   | moorthy@gmail.com
 kannan          | sql developer |  45000 | 2021-04-10   | kanna@gmail.com
```

**Order by:**

The ORDER BY clause in SQL will help us to sort the records based on the specific column of a table. This means that all the values stored in the column on which we are applying ORDER BY clause will be sorted, and the corresponding column values will be displayed in the sequence in which we have obtained the values in the earlier step.

**command--->select * from employee order by role;**

```
        name        |          role          | salary | joining_date |
email_id
--------------------+------------------------+--------+--------------
+-------------------------
 dhinakaran         | manager                |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
 mohammed ziavudeen | software architecture  |  30000 | 2023-05-01   |
mdziavu443@gmail.com
 benito             | software architecture  |  50000 | 2023-09-04   |
benito@gmail.com
 saravanan          | sql developer          |  35000 | 2023-04-21   |
saro123@gmail.com
 kannan             | sql developer          |  45000 | 2021-04-10   |
kanna@gmail.com
 sasi               | team lead              |  29000 | 2023-04-23   |
sasi123@gmail.com
 manjunathan        | tester                 |  40000 | 2023-04-20   |
manju1123@gmail.com
 thandavamoorthy    | web developer          |  35000 | 2023-04-02   |
moorthy@gmail.com
```

We can see that role is listed by alphabatically order.By default order by list at alphabatically order suppose we need to list it descending order we should give at last **desc**.

To order by descending order--->**select * from employee order by role desc;**

```
        name        |          role          | salary | joining_date |
email_id
--------------------+------------------------+--------+--------------
+-------------------------
 thandavamoorthy    | web developer          |  35000 | 2023-04-02   |
moorthy@gmail.com
 manjunathan        | tester                 |  40000 | 2023-04-20   |
manju1123@gmail.com
 sasi               | team lead              |  29000 | 2023-04-23   |
sasi123@gmail.com
 kannan             | sql developer          |  45000 | 2021-04-10   |
kanna@gmail.com
```

```
 saravanan          | sql developer         |  35000 | 2023-04-21   |
saro123@gmail.com
 benito             | software architecture |  50000 | 2023-09-04   |
benito@gmail.com
 mohammed ziavudeen | software architecture |  30000 | 2023-05-01   |
mdziavu443@gmail.com
 dhinakaran         | manager               |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
```

We can apply order by whatever we need.For eg,

**select * from employee order by name desc,joining_date asc,salary desc;**

```
       name         |         role          | salary | joining_date |
email_id
--------------------+-----------------------+--------+--------------
+-------------------------
 thandavamoorthy    | web developer         |  35000 | 2023-04-02   |
moorthy@gmail.com
 sasi               | team lead             |  29000 | 2023-04-23   |
sasi123@gmail.com
 saravanan          | sql developer         |  35000 | 2023-04-21   |
saro123@gmail.com
 mohammed ziavudeen | software architecture |  30000 | 2023-05-01   |
mdziavu443@gmail.com
 manjunathan        | tester                |  40000 | 2023-04-20   |
manju1123@gmail.com
 kannan             | sql developer         |  45000 | 2021-04-10   |
kanna@gmail.com
 dhinakaran         | manager               |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
 benito             | software architecture |  50000 | 2023-09-04   |
benito@gmail.com
```

**Limit:**

The LIMIT operator can be used in situations such as,where we need to find the top 3 high paying employee details and do not want to use any conditional statements.In such a case we use limit.

command---> **select * from employee order by salary desc limit 3;**

```
    name    |         role          | salary | joining_date |        email_id
------------+-----------------------+--------+--------------
+-------------------------
 benito     | software architecture |  50000 | 2023-09-04   | benito@gmail.com
 dhinakaran | manager               |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
 kannan     | sql developer         |  45000 | 2021-04-10   | kanna@gmail.com
```

**Limit along with Offset:**

The OFF SET value is also most often used together with the LIMIT keyword. The OFF SET value allows us to specify which row to start from retrieving data

Let’s suppose that we want to get a limited number of members starting from the middle of the rows, we can use the LIMIT keyword together with the offset value to achieve that.

command--->**select from employee order by name limit 2 OFFSET 5 ;**

```
   name    |     role      | salary | joining_date |     email_id
-----------+---------------+--------+--------------+------------------
 saravanan | sql developer |  35000 | 2023-04-21   | saro123@gmail.com
 sasi      | team lead     |  29000 | 2023-04-23   | sasi123@gmail.com
```

# Aggregate functions

First we shall see Aggregate functions.There are 5aggregate functions in PSQL.
1.Count
2.Average
3.Sum
4.Mn
5.Max

**1.Count:**

The COUNT() function returns the number of rows that matches a specified criterion.
To count the number of rows in the table.
Query—>**select count(*) from employee;**

```
 count
-------
     8
(1 row)
8 rows available in employee table
```

**2.Average**

The `AVG()` function returns the average value of a numeric column.

Query—>**select avg(salary) from employee;**

```
     avg
---------------------
 39250.000000000000
```

**3.Sum:**

The `SUM()` function returns the total sum of a numeric column.

Query—>**select sum(salary) from employee;**

```
  sum
--------
 314000
```

**4.Min and Max:**

The `MIN()` function returns the smallest value of the selected column.

The `MAX()` function returns the largest value of the selected column

To get minimum salary,Query—>**select min(salary) from employee;**

```
 min
-------
 29000
```

To get minimum salary,Query—>**select min(salary) from employee;**

```
 max
-------
 50000
```

The above query is all about basic query,Now we practice query inside query which is sub query.

As far as,we have learn to retrieve the highest salary,lowest salary.Now imagine if the question is to retreive the all the details of the highest salaried person or lower salary person.So,we should retreive through sub query.

Select all details from the highest salaried person.Query—>**select * from employee where salary=(select max(salary) from employee);**

```
    name    |         role          | salary | joining_date
 |        email_id
------------+-----------------------+--------+--------------
+-------------------------
 dhinakaran | manager               |  50000 | 2023-03-31   |
dhinakaran123@gmail.com
 benito     | software architecture |  50000 | 2023-09-04   |
 benito@gmail.com
 (2 rows)
```

Select 2nd highest salary from the table.Query–**->select * from employee where salary=(select max(salary) from employee where salary not in (select max(salary) from employee));**

```
   name  |     role      | salary | joining_date |    email_id
--------+---------------+--------+--------------
+-----------------
 kannan | sql developer |  45000 | 2021-04-10   | kanna@gmail.com
```

# Group by

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

query–>**select role,sum(salary) from employee group by role;**

```
role | sum
----------------------+-------
manager | 50000
tester | 40000
sql developer| 80000
web developer| 35000
team lead| 29000
 software architecture | 80000
```

To retreive minimum salary from all the role.query—>**Select role,min(salary) from employee group by role;**

```
        role          |  min
----------------------+-------
 manager              | 50000
 tester               | 40000
 sql developer        | 35000
 web developer        | 35000
 team lead            | 29000
 software architecture | 30000
```

To retreive sum of the salary with count from all the role.query—>**select count(name),sum(salary) from employee group by role;**

```
count | max
-------+-------
1 | 50000
1 | 40000
2 | 45000
1 | 35000
1 | 29000
2 | 50000
(6 rows)
```

To retreive average salary with roundoff and role and order should by average's ascending order.Query—>**select role,round(avg(salary),2) from employee group by role order by (avg(salary));**

```
Role | round

----------------------+----------
team lead | 29000.00
web developer | 35000.00
tester | 40000.00
sql developer | 40000.00
software architecture | 40000.00
manager | 50000.00
```

# Having:

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results.

Retrieve the count in which role has more than 1 role.Query—>**select count(salary),role from employee group by role having count(salary)>1;**

```
count |         role
-my------+----------------------
    2 | sql developer
    2 | software architecture
```

Now,discuss joints, PSQL is a Relational Database Management System .RDBMS defines the database in the form of table,the tables are **related to each other**.

Basically,we can connect the two tables by using join statement,any one of the column name in the tables must be same,that column defines the relationship between two tables.We shall do practicaly here to understand.

# Sql join

**SQL Join** statement is used to combine data or rows from two or more tables based on a common field between them. This blog presents a basic overview of what data from a particular SQL join will look like.

I have presented with examples corresponding to the venn diagram

Different types of Joins are as follows:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- OUTER [LEFT | RIGHT | FULL] JOIN
- NATURAL

First i shall create two tables based on connectivity.We may all familier with cricket so, i shall create two tables name **team** and **player** and **player_id** is common or relational between these two tables.

As we already practiced to create table and to insert the values in my previous blog.

To create table team.Query—>**create table team (team_id int,player_id int,joined_date date);**
CREATE TABLE

To insert value into the table—>**insert into team values(101,1,’31-03-2023′),(201,2, ’30-03-2023′), (301, 3, ’01-04-2023′), (401, 4, ’01-04-2023′),(501,7,’09/09/2023′);;**
INSERT 0 5

To create table player.Query—>**create table player(player_id int,player_name varchar(20),team_name varchar(20),country varchar(20));**

To insert value into the table—>**insert into player values (1, 'Dhoni', 'CSK', 'India'), (2, 'Warner', 'Delhi', 'Australia'), (3, 'Buttler', 'Rajasthan', 'England'), (4, 'Kohli', 'KKR', 'India'), (5, 'Rohit', 'Hyderabad', 'India'), (6, 'Jadeja', 'Rajasthan', 'India');**

Viewing the team table—>**select * from team;**

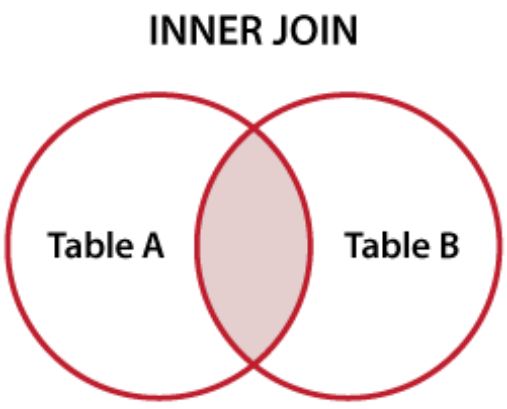
```
team_id | player_id | joined_date
---------+-----------+-------------
     101 |         1 | 2023-03-31
     201 |         2 | 2023-03-30
     301 |         3 | 2023-04-01
     401 |         4 | 2023-04-01
     501 |         7 | 2023-09-09
```

Viewing the player table—>**select * from player;**

```
player_id | player_name | team_name |  country
-----------+-------------+-----------+-----------
         1 | Dhoni       | CSK       | India
         2 | Warner      | Delhi     | Australia
         3 | Buttler     | Rajasthan | England
         4 | Kohli       | KKR       | India
         5 | Rohit       | Hyderabad | India
         6 | Jadeja      | Rajasthan | India
```

**Inner join:**

The SQL `INNER JOIN` joins two tables based on a common column, and selects records that have matching values in these columns.



Sample query for inner join—>**select team.team_id,player.player_name,team.joined_date from player inner join team on team.player_id=player.player_id;**

```
team_id | player_name | joined_date
---------+-------------+-------------
     101 | Dhoni       | 2023-03-31
     201 | Warner      | 2023-03-30
     301 | Buttler     | 2023-04-01
     401 | Kohli       | 2023-04-01
```

**Inner join with where clause:**

Incase we need some condition while joining,we can apply both join and where clause.For example,

Query—>**select player.player_name,team.player_id,player.country from player inner join team on team.player_id=player.player_id where player. player_id>=2;**

```
player_name | player_id |  country
-------------+-----------+-----------
 Warner      |         2 | Australia
 Buttler     |         3 | England
 Kohli       |         4 | India
```

**Inner join with like:**

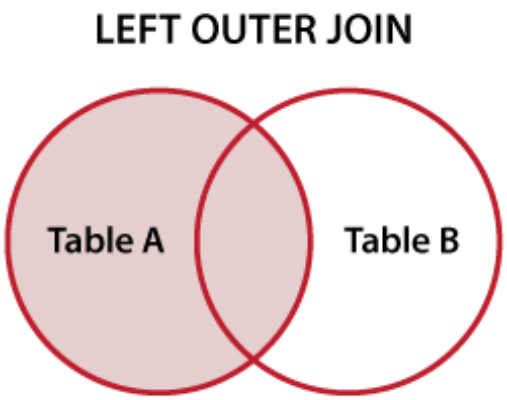
We can use inside `INNER JOIN` with like. For example,

Query—>**select player.player_name,team.player_id,player.country from player inner join team on team.player_id=player.player_id where player.country like 'I%';**

```
player_name | player_id | country
-------------+-----------+---------
 Dhoni       |         1 | India
 Kohli       |         4 | India
```

**Left outer join:**

A **LEFT OUTER JOIN** returns all rows from the left table (TableA) with the matching rows from the right table (TableB) or *null* – if there is no match in the right table.



Courtesy:learnsql

Query—>**select * from team left join player on team.player_id=player.player_id;**

```
 team_id | player_id | joined_date | player_id | player_name |
 team_name |  country
 ---------+-----------+-------------+-----------+-------------
 +-----------+-----------
1     101 |         1 | 2023-03-31  |         1 | Dhoni       |
2CSK       | India
3     201 |         2 | 2023-03-30  |         2 | Warner      |
4Delhi     | Australia
5     301 |         3 | 2023-04-01  |         3 | Buttler     |
6Rajasthan | England
7     401 |         4 | 2023-04-01  |         4 | Kohli       |
 KKR        | India
      501 |         7 | 2023-09-09  |           |
 |           |
```

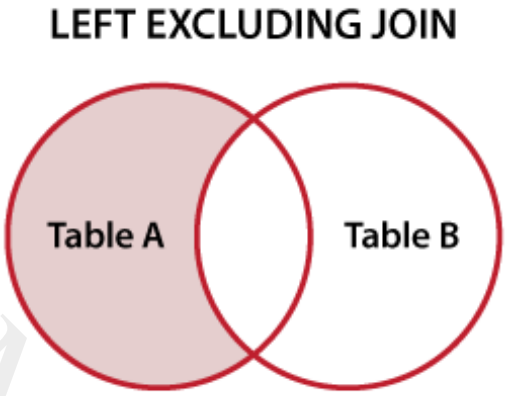
Query—>**select * from player left join team on team.player_id=player.player_id;**

```
 player_id | player_name | team_name |  country  | team_id |
 player_id | joined_date
 -----------+-------------+-----------+-----------+---------
 +-----------+-------------
1         1 | Dhoni       | CSK       | India     |     101
2|         1 | 2023-03-31
3         2 | Warner      | Delhi     | Australia |     201
4|         2 | 2023-03-30
5         3 | Buttler     | Rajasthan | England   |     301
6|         3 | 2023-04-01
7         4 | Kohli       | KKR       | India     |     401
8|         4 | 2023-04-01
          5 | Rohit       | Hyderabad | India     |
 |           |
          6 | Jadeja      | Rajasthan | India     |
 |           |
```

The above 2tables shows that left join select common data between two tables and all data in the left table.

**Left outer join:**

A **LEFT EXCLUDING JOIN** returns all of the records in TableA that don’t match any record in TableB.




Query—>**select * from player left join team on player.player_id = team.player_id where team.team_id is null;**

```
 player_id | player_name | team_name | country | team_id |
 player_id | joined_date
1-----------+-------------+-----------+---------+---------
2+-----------+-------------
3          5 | Rohit       | Hyderabad | India   |
4|           |
           6 | Jadeja      | Rajasthan | India   |
 |           |
```

**Right outer join:**

A **RIGHT OUTER JOIN** returns all rows from the right table (TableB) with the matching rows from the left table (TableA) or *null* – if there is no match in the left table.

Query—>**select player.player_name,team.joined_date,player.player_id from team right join player on team.player_id = player.player_id;;**

```
player_name |    joined_date    | player_id
-------------+------------+-----------
 Dhoni       | 2023-03-31 |         1
 Warner      | 2023-03-30 |         2
 Buttler     | 2023-04-01 |         3
 Kohli       | 2023-04-01 |         4
 Rohit       |            |         5
 Jadeja      |            |         6
```

Query—>**select player.player_name,team.joined_date,player.player_id from player right join team on team.player_id = player.player_id;**

```
player_name | joined_date | player_id
-------------+-------------+-----------
 Dhoni       | 2023-03-31  |         1
 Warner      | 2023-03-30  |         2
 Buttler     | 2023-04-01  |         3
 Kohli       | 2023-04-01  |         4
             | 2023-09-09  |
```

**Note:**From the above example we have noticed that left join and right join similar,if we give table name before the left join and after the right join and vice versa.

**Right Excluding join:**

A **RIGHT EXCLUDING JOIN** returns all of the records in TableB that don’t match any records in TableA.

Query—>**select * from player right join team on player.player_id = team.player_id where player.country is null;**

```
 player_id | player_name | team_name | country | team_id |
player_id | joined_date
-----------+-------------+-----------+---------+---------
+-----------+-------------
           |             |           |         |     501
|         7 | 2023-09-09
 (1 row)
```

**Full Outer join:**

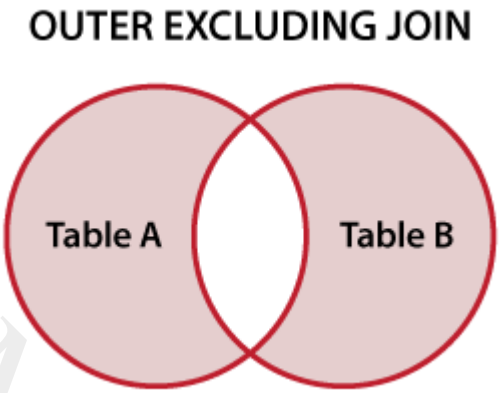
**FULL OUTER JOIN** returns matched and unmatched rows from both tables (it's an union of both). If there is no match, the missing side will contain null.

Query—>**select * from team full outer join player on player.player_id = team.player_id;**

```
 team_id | player_id | joined_date | player_id | player_name |
 team_name |  country
---------+-----------+-------------+-----------+-------------
+-----------+-----------
     101 |         1 | 2023-03-31  |         1 | Dhoni       |
CSK       | India
     201 |         2 | 2023-03-30  |         2 | Warner      |
Delhi     | Australia
     301 |         3 | 2023-04-01  |         3 | Buttler     |
Rajasthan | England
     401 |         4 | 2023-04-01  |         4 | Kohli       |
KKR       | India
     501 |         7 | 2023-09-09  |           |
|           |
         |           |             |         5 | Rohit       |
Hyderabad | India
         |           |             |         6 | Jadeja      |
Rajasthan | India
(7 rows)
```

**Outer Excluding join:**

An **OUTER EXCLUDING JOIN** returns all of the records in TableA and all of the records in TableB that don't match.




Query—>**select * from team full outer join player on player.player_id = team.player_id where team.team_id is null or player.player_id is null;**

```
 is null;
  team_id | player_id | joined_date | player_id | player_name |
team_name | country
1---------+-----------+-------------+-----------+-------------
2+-----------+---------
3      501 |         7 | 2023-09-09  |           |
4|           |
5          |           |             |         5 | Rohit       |
6Hyderabad | India
7          |           |             |         6 | Jadeja      |
 Rajasthan | India
 (3 rows)
```



References:

https://github.com/muthu1809/PostgreSQL/blob/main/postgresql_muthu.odt
https://www.geeksforgeeks.org/sql-tutorial/
https://www.w3schools.com/sql/
https://www.guru99.com/sql.html
https://learnsql.com/blog/sql-joins/

https://www.programiz.com/sql/right-join