PostgreSQL JOIN

Introduction

- In PostgreSQL, **JOIN** is used with the SELECT statement to retrieve data from multiple related tables.
- It combines rows from two or more tables based on a related column (usually **primary key–foreign key**).

Purpose of JOIN

- Merge columns from one or more tables using a related column.
- Useful in normalized databases where data is split across multiple tables.

How JOIN Works

- Performed using a **common column** between tables.
- common column is typically the **primary key** in one table and **foreign key** in another.

PRIMARY KEY

- A column or combination of columns that uniquely identifies each row in a table.
- Properties:
    1. Must be unique for every row.
    2. Cannot contain NULL values.
    3. Only one primary key is allowed per table.

- Purpose: To ensure each record can be identified uniquely in the table.

- Example:
  StudentID in a Students table.

FOREIGN KEY

- A column or combination of columns in one table that refers to the primary key in another table.

- Properties:

    1. Can contain duplicate values.

    2. Can contain NULL values (unless NOT NULL is specified).

    3. Used to maintain referential integrity between two tables.

- Purpose: To create a link between two tables.

- Example:
  StudentID in an Orders table referencing StudentID in Students table.

Types of PostgreSQL JOIN

1. **INNER JOIN** – Returns only matching rows between tables.

2. **LEFT JOIN / LEFT OUTER JOIN** – Returns all rows from the left table, matched rows from the right table, and NULL for no match.

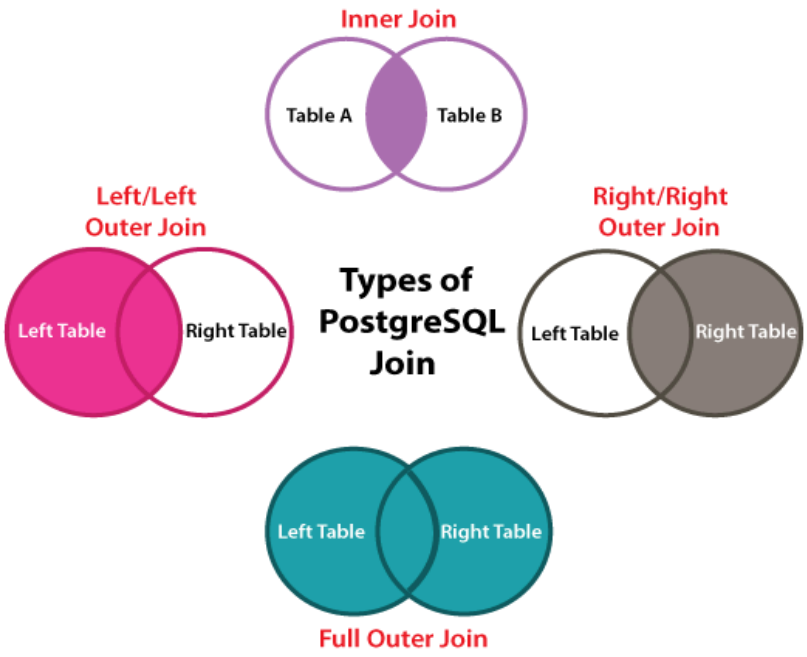
3. **RIGHT JOIN / RIGHT OUTER JOIN** – Returns all rows from the right table, matched rows from the left table, and NULL for no match.

4. **FULL OUTER JOIN** – Returns all rows when there's a match in either table, NULL where no match.

5. **CROSS JOIN** – Returns the **Cartesian product** of the two tables.

6. **NATURAL JOIN** – Joins tables automatically based on same column names and compatible data types.

7. **SELF JOIN** – Joins a table to itself.



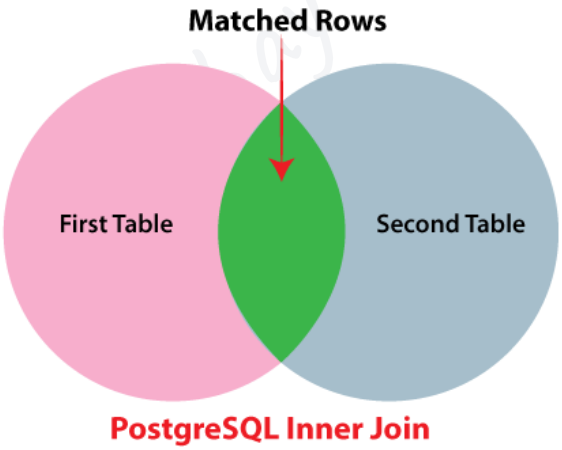
PostgreSQL INNER Join

understand the working of inner join, which is used to select data from many tables. We also learn how to use WHERE clause, USING clause, operators, and join three tables, table-aliasing in PostgreSQL inner join.

What is the INNER JOIN clause?

In a relational database, data is naturally spread in more than one table, and to select aggregate data, we often need to select data from various tables.

The **Inner Join is used to return only those records from the tables, which are equivalent to the defined condition and hides other rows and columns.** It has a default Join, therefore it is not compulsory to use the Inner Join keyword with the query.



Inner Join Syntax

The Inner Join keyword is used with the SELECT command and must be written after the FROM clause.

The below syntaxes describe it more clearly:

```
SELECT [column_list |*]

FROM table1

INNER JOIN table2

ON table1.column_name=table2.column.name;
```

Syntax of Inner Join with USING clause

```
SELECT [column_list |* ]

FROM table1

INNER JOIN table2

USING (column.name);
```

Syntax of inner Join with WHERE Clause

```
SELECT [column_list |*]

FROM table1, table2

WHERE table1.column_name=table2.column_name;
```

We will follow the below steps to join Table A with Table B:

- Firstly, we will define the column list from both tables (tables 1 and 2), where we want to select data in the SELECT condition.
- Then, we will define the base table, which is table 1 in the FROM clause.
- And lastly, we will describe the second table (table 2) in the INNER JOIN condition, and write the join condition after the ON keyword.

  Note: The Join condition returns the similar rows between the tables described in the Inner condition.

Example of Inner Join

To join two tables by using INNER JOIN

Firstly, we are going to create an ***Employee*** *and* ***department*** tables by using the CREATE command:

```
create table Employee(
emp_id int primary key,
emp_fname varchar not null,
emp_lname varchar not null,
location varchar(30) );
```

The below command is used to create a ***Department*** table:

```
Create table department
(emp_id int primary key,
```

dept_id int not null,

dept_name varchar NOT NULL);

The *Employee and department* tables have been successfully created after executing the above commands.

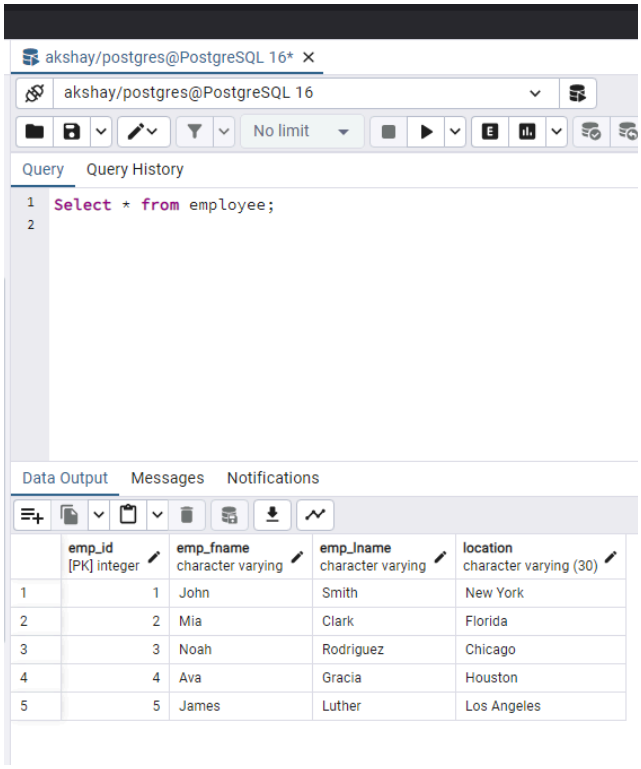
Once the both the tables have been generated, we are ready to insert some values into it by using the INSERT command as follows:

```
INSERT INTO Employee (emp_id, emp_fname, emp_lname, location)

VALUES

(1, 'John', 'Smith', 'New York'),

(2, 'Mia', 'Clark','Florida'),

(3, 'Noah','Rodriguez','Chicago'),

(4, 'Ava','Gracia','Houston'),

(5,'James','Luther','Los Angeles');
```

In the below command, we are inserting the values in the *department* table:

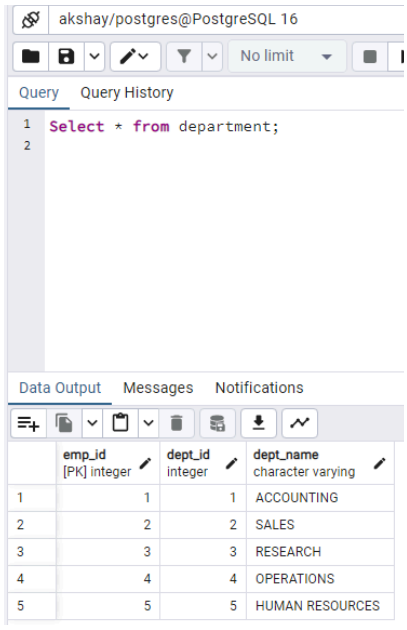
```
INSERT INTO department (emp_id, dept_id, dept_name)

VALUES

(1, 1, 'ACCOUNTING'),

(2, 2,'SALES'),

(3, 3,'RESEARCH'),

(4, 4,'OPERATIONS'),

(5, 5,'HUMAN RESOURCES');
```

Table1: Employee



	emp_id [PK] integer	emp_fname character varying	emp_lname character varying	location character varying (30)
1	1	John	Smith	New York
2	2	Mia	Clark	Florida
3	3	Noah	Rodriguez	Chicago
4	4	Ava	Gracia	Houston
5	5	James	Luther	Los Angeles

Table2: department



	emp_id [PK] integer	dept_id integer	dept_name character varying
1	1	1	ACCOUNTING
2	2	2	SALES
3	3	3	RESEARCH
4	4	4	OPERATIONS
5	5	5	HUMAN RESOURCES

The below query is used to select records from both tables **(Employee and department):**

```
SELECT emp_fname, emp_lname,location,dept_name
FROM Employee
INNER JOIN department
ON Employee.emp_id= department.dept_id;
```

Query | Query History

```
1  SELECT emp_fname, emp_lname,location,dept_name
2  FROM Employee
3  INNER JOIN department
4  ON Employee.emp_id= department.dept_id;
5
```

Data Output | Messages | Notifications

	emp_fname character varying	emp_lname character varying	location character varying (30)	dept_name character varying
1	John	Smith	New York	ACCOUNTING
2	Mia	Clark	Florida	SALES
3	Noah	Rodriguez	Chicago	RESEARCH
4	Ava	Gracia	Houston	OPERATIONS
5	James	Luther	Los Angeles	HUMAN RESOURCES

Working of Inner Join

- Compares rows from two tables based on a matching column.
- If values match → combines data from both tables into one row.
- If no match → row is ignored.
- Result shows only matching rows from both tables.

Table-aliasing with Inner Join

Generally, the tables we want to join will have columns with a similar name like the emp_id column.

If we reference columns with a similar name from different tables in a command, the error will have occurred, and to avoid this particular error, we need to use the below syntax.

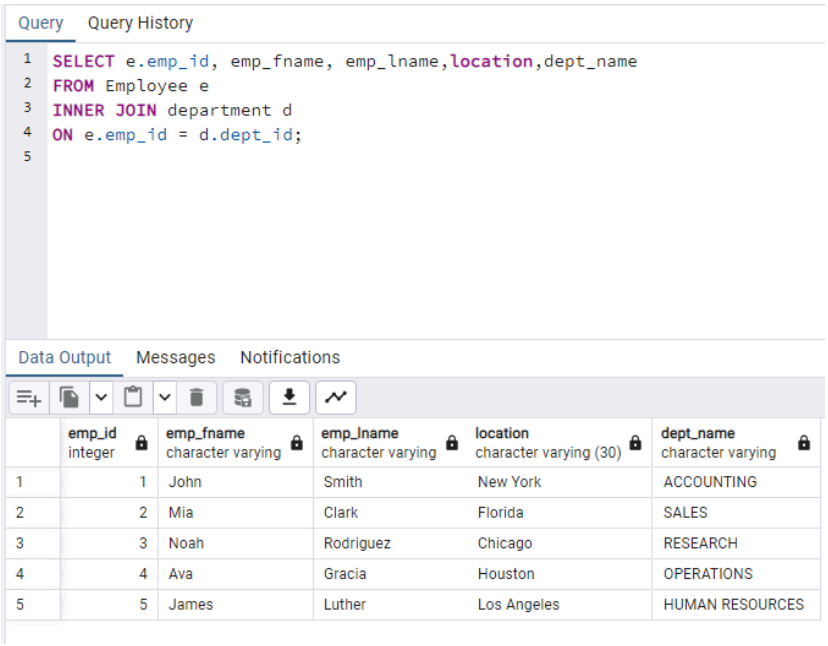
table_name.column_name

In real-time, we will use table aliases to assign the joined tables short names to make the command more understandable.

In the below command, we will use the table aliasing, and it returns the similar outcome as above:

```
SELECT e.emp_id, emp_fname, emp_lname,location,dept_name
FROM Employee e
INNER JOIN department d
ON e.emp_id = d.dept_id;
```

Output



	emp_id integer	emp_fname character varying	emp_lname character varying	location character varying (30)	dept_name character varying
1	1	John	Smith	New York	ACCOUNTING
2	2	Mia	Clark	Florida	SALES
3	3	Noah	Rodriguez	Chicago	RESEARCH
4	4	Ava	Gracia	Houston	OPERATIONS
5	5	James	Luther	Los Angeles	HUMAN RESOURCES

Inner Join with USING Clause

Here, we will see how the PostgreSQL inner join works with the USING clause because sometimes the name of the columns is similar in both the tables; that's why we can use the USING clause to get the values.

In the below example, we are using the USING clause as both tables have a similar emp_id column.

```
SELECT emp_id, emp_fname,location, dept_name
FROM Employee
INNER JOIN department
USING (emp_id);
```

Output

```
SELECT emp_id, emp_fname,location, dept_name
FROM Employee
INNER JOIN department
USING (emp_id);
```

Data Output | Messages | Notifications

	emp_id integer	emp_fname character varying	location character varying (30)	dept_name character varying
1	1	John	New York	ACCOUNTING
2	2	Mia	Florida	SALES
3	3	Noah	Chicago	RESEARCH
4	4	Ava	Houston	OPERATIONS
5	5	James	Los Angeles	HUMAN RESOURCES

Inner Join using WHERE clause

We can also use the inner join with a WHERE condition. The WHERE clause allows us to return the filter outcome.

In the below example, we will select rows from both tables ***Employee*** and ***department*** where **dept_name** is equal to Sales:

```
SELECT emp_fname, dept_id, dept_name, location

FROM Employee

INNER JOIN department

USING (emp_id) WHERE dept_name ='SALES';
```

Output

```
SELECT emp_fname, dept_id, dept_name, location
FROM Employee
INNER JOIN department
USING (emp_id) WHERE dept_name ='SALES';
```

Data Output | Messages | Notifications

	emp_fname character varying	dept_id integer	dept_name character varying	location character varying (30)
1	Mia	2	SALES	Florida

To join three tables using Inner Join

we have already created two tables as *Employee and department.* Now, we want to join one more table and get the records from that particular table with the help of the Inner join.

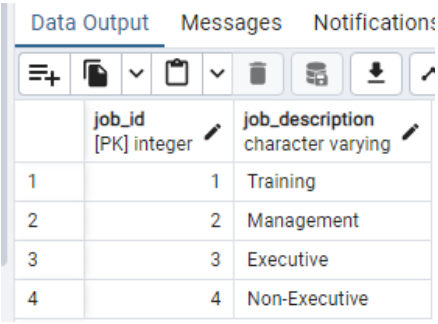
So, for this, we will create one more table as ***Jobs*** by using CREATE command as we can see in the following command:

```
CREATE TABLE Jobs(

job_id int primary key,

job_description varchar not null);
```

Once the *Jobs* table has been created successfully, we will insert some values into it with the help of INSERT command as we can see in the following command:

```
INSERT INTO Jobs (job_id, job_description)

VALUES (1, 'Training'),

(2, 'Management'),

(3, 'Executive'),

(4, 'Non-Executive');
```

Table3: Jobs

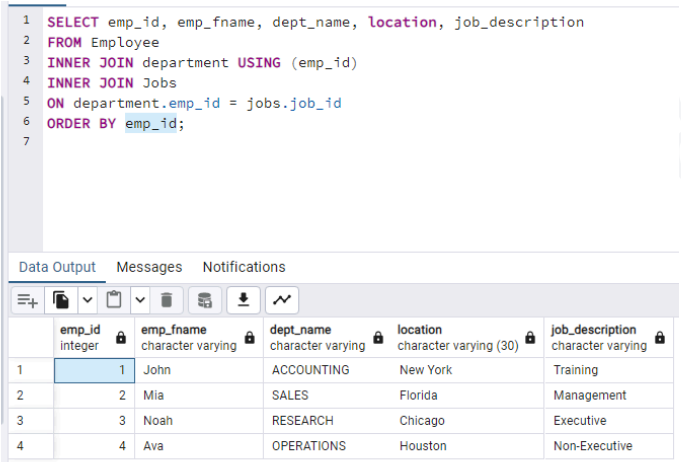


	job_id [PK] integer	job_description character varying
1	1	Training
2	2	Management
3	3	Executive
4	4	Non-Executive

The below command is used to join the three tables, such as *Employee, department, and Jobs*. So, we will use the second INNER JOIN clause after the first INNER JOIN clause:

```
SELECT emp_id, emp_fname, dept_name, location, job_description

FROM Employee

INNER JOIN department USING (emp_id)

INNER JOIN Jobs

ON department.emp_id = jobs.job_id

ORDER BY emp_id;
```

Output



```
SELECT emp_id, emp_fname, dept_name, location, job_description
FROM Employee
INNER JOIN department USING (emp_id)
INNER JOIN Jobs
ON department.emp_id = jobs.job_id
ORDER BY emp_id;
```

Data Output | Messages | Notifications

	emp_id integer	emp_fname character varying	dept_name character varying	location character varying (30)	job_description character varying
1	1	John	ACCOUNTING	New York	Training
2	2	Mia	SALES	Florida	Management
3	3	Noah	RESEARCH	Chicago	Executive
4	4	Ava	OPERATIONS	Houston	Non-Executive

Inner Join using Operators

PostgreSQL allows many operators, which we can use with Inner Join, like equal (=), not equal (!=), greater than (>), less than (<), etc.

```
SELECT emp_fname, emp_lname,location,dept_name

FROM Employee

INNER JOIN department

ON Employee.emp_id= department.dept_id

WHERE dept_name != 'SALES';
```

Output

```sql
SELECT emp_fname, emp_lname,location,dept_name
FROM Employee
INNER JOIN department
ON Employee.emp_id= department.dept_id
WHERE dept_name != 'SALES';
```

Data Output | Messages | Notifications

	emp_fname character varying	emp_lname character varying	location character varying (30)	dept_name character varying
1	John	Smith	New York	ACCOUNTING
2	Noah	Rodriguez	Chicago	RESEARCH
3	Ava	Gracia	Houston	OPERATIONS
4	James	Luther	Los Angeles	HUMAN RESOURCES

PostgreSQL Left Join

we are going to understand the working of **Left join**, which is used to return data from the left table. We also learn how to use **table-aliasing, WHERE clause, USING clause, and join multiple tables** with the help of the Left join clause.

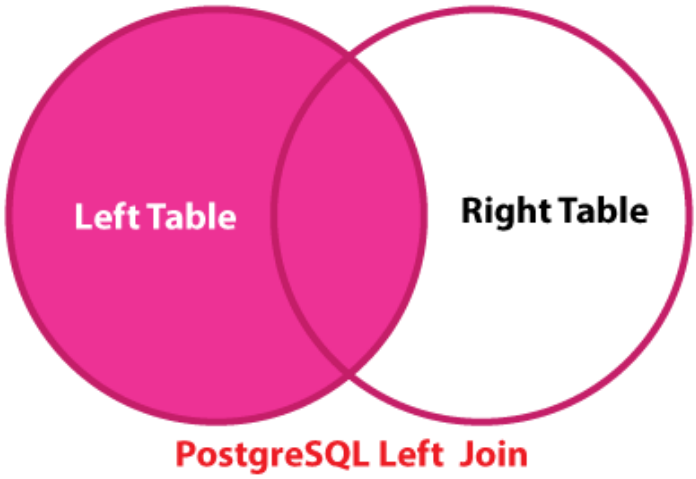
What is Left Outer Join or Left Join clause?

The **LEFT JOIN** (also called **LEFT OUTER JOIN**) is used in PostgreSQL to combine records from two tables. It ensures that **all rows from the left table** are included in the result set. If the join condition finds matching rows in the right table, those values are returned; otherwise, **NULL values** are placed for the right table's columns.

The keyword **OUTER** is optional, so both LEFT JOIN and LEFT OUTER JOIN mean the same.

In comparison to **INNER JOIN**, which only returns matching rows, the LEFT JOIN guarantees that no row from the left table is lost, even when there is no match in the right table.

This makes LEFT JOIN useful when we want to preserve all data from the primary (left) table while optionally pulling related information from another (right) table.



Left Join Syntax

Notes: LEFT JOIN Syntax and Steps

- The **LEFT JOIN** keyword is used with the **SELECT** command.
- It must be written **after the FROM keyword**.

Syntax:

- SELECT columns
- FROM table1
- LEFT [OUTER] JOIN table2
- ON table1.column = table2.column;
- **table1** → Left table (all its rows will be returned).
- **table2** → Right table (only matching rows will be returned, otherwise NULL).

Steps to use LEFT JOIN:

1. Define the column list from both tables in the **SELECT** statement.
2. Write the **Left table** (table1) in the **FROM clause**.
3. Write the **Right table** (table2) with the **LEFT JOIN clause**.
4. Specify the join condition using the **ON keyword**.

Example of Left join

Let us see an example to understand how the **Left join** works:

To join two tables by using Left Join

For this, we will create two tables named ***Client*** and ***Orders*** table with the help of the CREATE command and insert some values using the INSERT command.

we are going to create ***Client and Orders*** tables by using the **CREATE** command

CREATE TABLE Client(

client_id int primary key,

client_name varchar not null,

client_profession varchar not null,

client_qualification varchar not null,

client_salary int);

The below command is used to create an ***Orders*** table:

Create table Orders

(client_id **int** primary key,

order_id **int** not **null**,

price **int**,

order_date Date Not **null**);

The ***Client and Orders*** tables have been successfully created after executing the above commands.

Once the both the table have been generated, we are ready to insert some values into it by using the **INSERT** command as follows:

```
INSERT INTO Client (client_id, client_name,
client_profession, client_qualification, client_salary)
VALUES
(1, 'Emma Hernandez','Web Designer','BTech', 25000),
(2, 'Mia Clark','Software Engineer','BE',20000),
(3, 'Noah Rodriguez','Bussinessman','MBA',50000),
(4, 'Martha Brown','Doctor','MBBS',75000),
(5,'James Luther','HR','MBA',35000),
(6,'Maria Garcia','Astronaut','Msc', 100000),
(7,'Robert Smith','Software Tester','BTech',30000);
```

In the below command, we are inserting the values in the ***Orders*** table:

```
INSERT INTO Orders (client_id, order_id, price, order_date)
VALUES
(1, 101, 2000,'2020-05-14'),
(2, 102, 3500,'2019-08-30'),
(3, 103, 4000,'2020-06-23'),
(4, 104, 2500,'2017-12-11'),
(5, 105, 5000,'2018-10-26');
```

After creating and inserting the values in the ***Client and Orders*** table, we will get the following output on executing the below command:

Table1: Client

```
Select * from Client;
```

Output

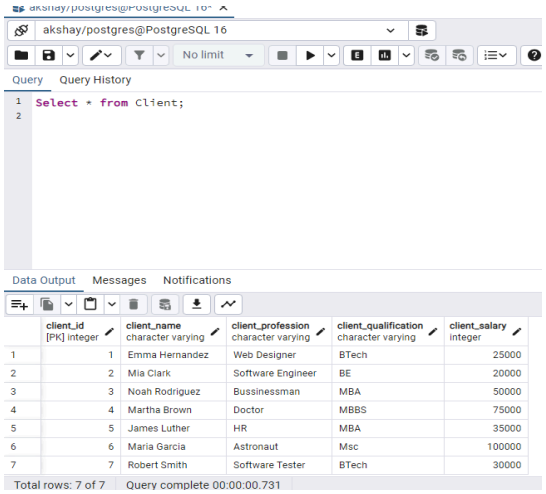
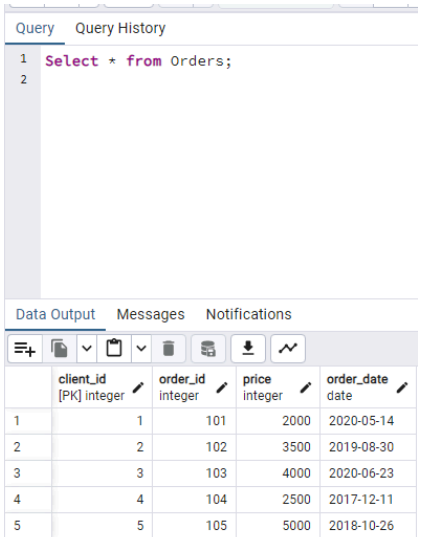


Table2: Orders

Select * from Orders;

Output

After executing the above command, we will get the data from the ***Orders*** table:



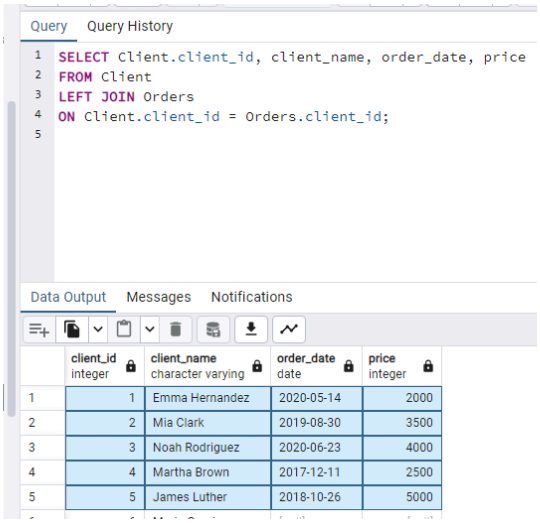
The below query is used to select records from both tables (**Client** and **Orders**):

SELECT Client.client_id, client_name, order_date, price

FROM Client

LEFT JOIN Orders

ON Client.client_id = Orders.client_id;



Or we use the **Left Outer Join keyword** in place of **Left Join keyword** in the above query as both will give similar output:

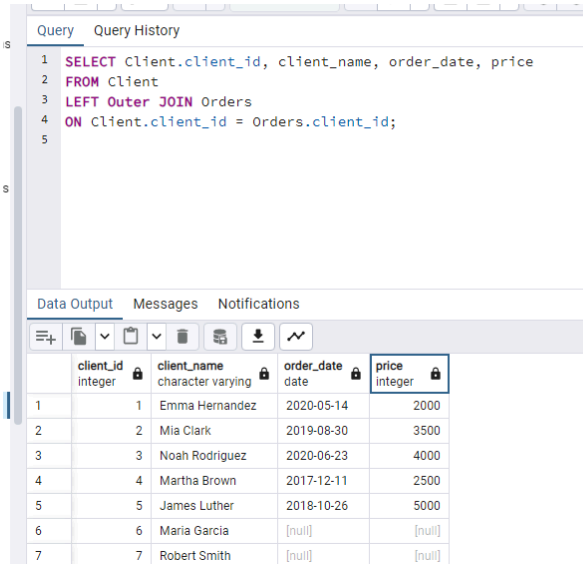
SELECT Client.client_id, client_name, order_date, price

FROM Client

LEFT Outer JOIN Orders

ON Client.client_id = Orders.client_id;

Output



Note: When a row from the Client table does not have a matching row in the Orders table, the value of the order_date and price column of the unmatched row would be NULL.

Working of Left join

- In the above screenshot, the left join condition selects the records from the **left table (Client)**, and it equates the values in the **client_id, client_name** column with the values in the **order_date, price** column from the ***Orders*** table.
- If these records are similar, then the **Left join** creates a new row, which has the columns that display in the **Select** Clause and adds the particular row to the result.
- Suppose, if the values are not similar, then the left join also generates a new row, which displays in the **SELECT** command, and it fills the columns that come from the right table (**Orders**) with **NULL**.

Table-aliasing with Left Join

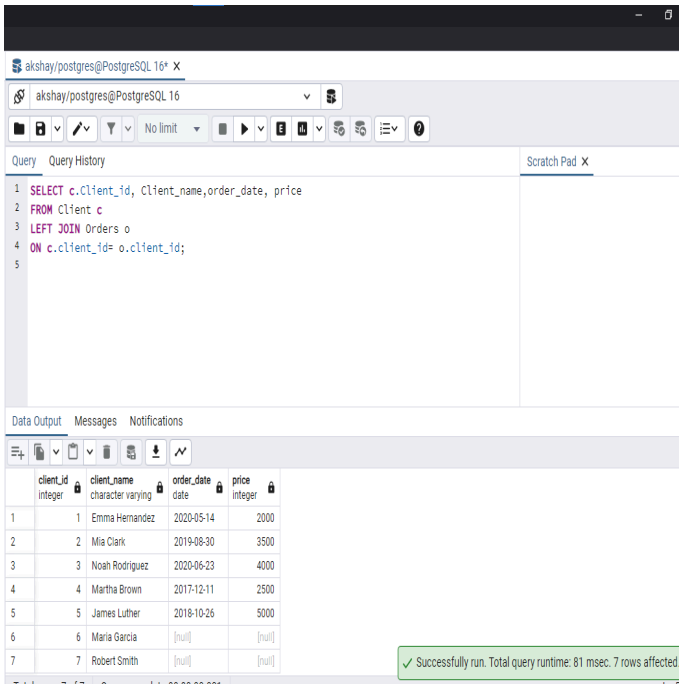
We will use **table aliases** to assign the joined tables short names to make the command more understandable.

Mostly, the tables we want to join will have columns with a similar name like **the Client_id** column. In the below command, we will use the table aliasing, and it returns a similar outcome as above:

```
SELECT c.Client_id, Client_name,order_date, price

FROM Client c

LEFT JOIN Orders o

ON c.client_id= o.client_id;
```

Output

Once we implemented the above command, we will get the below output:



Left join with USING Clause

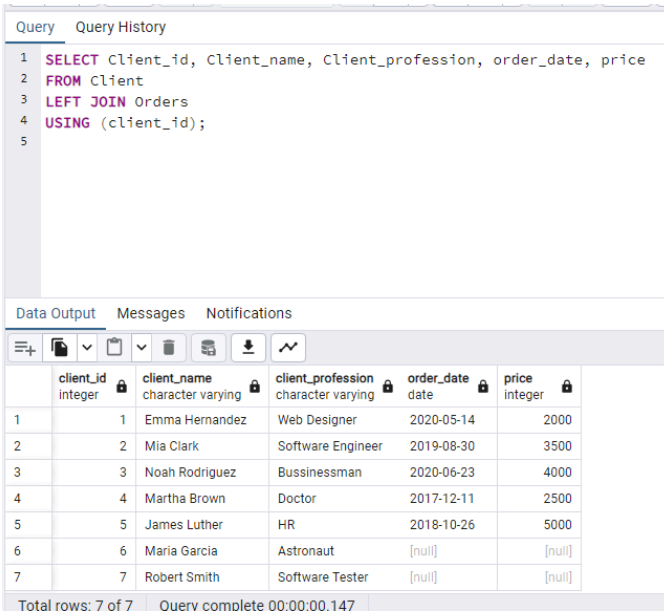
In this, we will see how the Left join works with the **USING** clause.

Both the tables (***Client and Orders***) contain a similar column name **Client_id;** that's why we can use the **USING** clause to get the values from the tables.

In the below example, we are using the **USING** clause in the **Left join**, which returns the values **Client_id, Client_name, Client_prof, price, and order_date** as both tables have a similar **Client_id** column.

```
SELECT Client_id, Client_name, Client_profession, order_date, price

FROM Client

LEFT JOIN Orders

USING (client_id);
```

Output



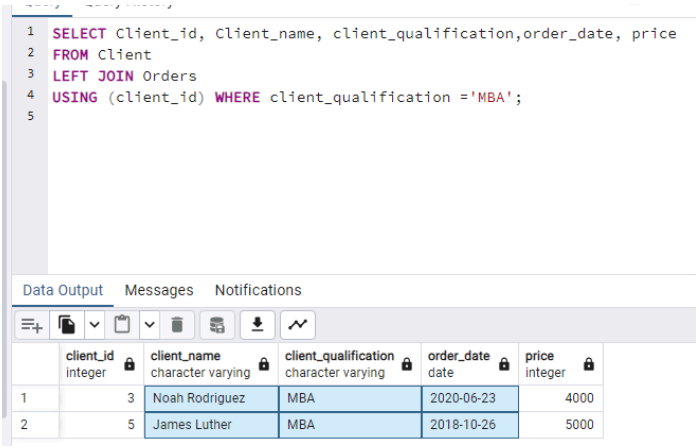
Left join using WHERE clause

Here, the **WHERE** clause allows us to return the filter outcome. And we can also use the WHERE condition with the **Left join**.

In the below example, we will select rows from both tables ***Client*** and ***Orders*** where **client_qualification** is equal to **MBA:**

```
SELECT Client_id, Client_name, client_qualification,order_date, price

FROM Client

LEFT JOIN Orders

USING (client_id) WHERE client_qualification ='MBA';
```

Output



Difference between WHERE and ON clause in PostgreSQL LEFT JOIN

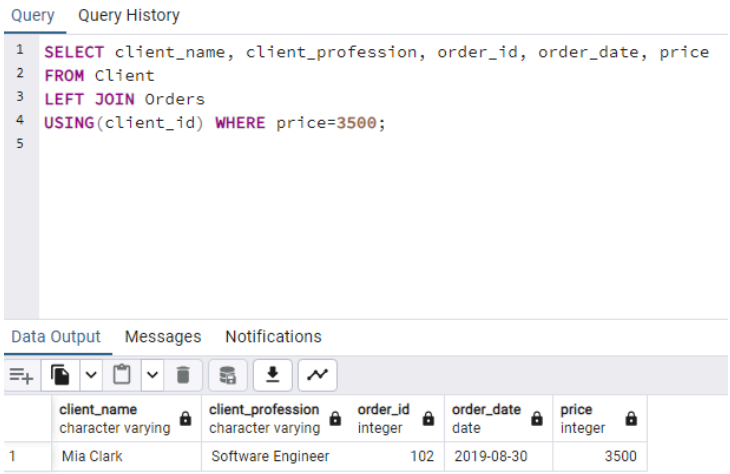
We can see the below command to understand the differences between WHERE, and **ON** clauses in the **PostgreSQL Left join**.

In the Left Join, the WHERE and ON clause gives us a different output.

Firstly, we are using the **WHERE Clause** with the Left join as we can see in the below command:

```
SELECT client_name, client_profession, order_id, order_date, price

FROM Client

LEFT JOIN Orders

USING(client_id) WHERE price=3500;
```

Output



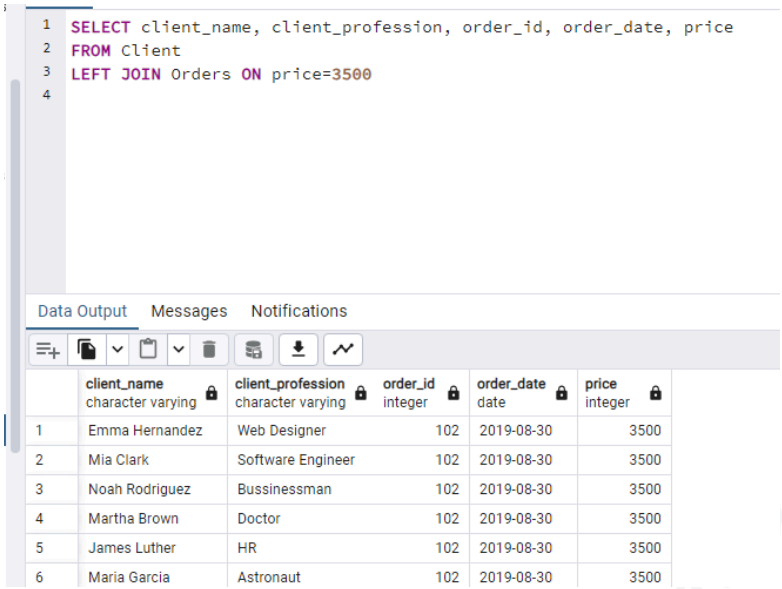
Now, we are using the **ON Clause** with the Left join as we can see in the below command:

```
SELECT client_name, client_profession, order_id, order_date, price

FROM Client
```

LEFT JOIN Orders ON price=3500;

Output

After executing the above command, we will get the below result:



```
SELECT client_name, client_profession, order_id, order_date, price
FROM Client
LEFT JOIN Orders ON price=3500
```

Data Output | Messages | Notifications

	client_name character varying	client_profession character varying	order_id integer	order_date date	price integer
1	Emma Hernandez	Web Designer	102	2019-08-30	3500
2	Mia Clark	Software Engineer	102	2019-08-30	3500
3	Noah Rodriguez	Bussinessman	102	2019-08-30	3500
4	Martha Brown	Doctor	102	2019-08-30	3500
5	James Luther	HR	102	2019-08-30	3500
6	Maria Garcia	Astronaut	102	2019-08-30	3500

Note: In PostgreSQL, the Inner join will always return a similar output if we using the WHERE and ON clauses in the command.

To join various tables using PostgreSQL Left JOIN

In the above section, we have already created two tables as ***Client*** and ***Orders***. Now, we are using the Left join to combine the various tables and get the records from that particular table.So, we will create one more table as ***Clieint_ details*** by using CREATE command as we can see in the following command:

```
CREATE TABLE Client_details (

Phone_id int primary key,

Mobile_number bigint,

address varchar not null

);
```

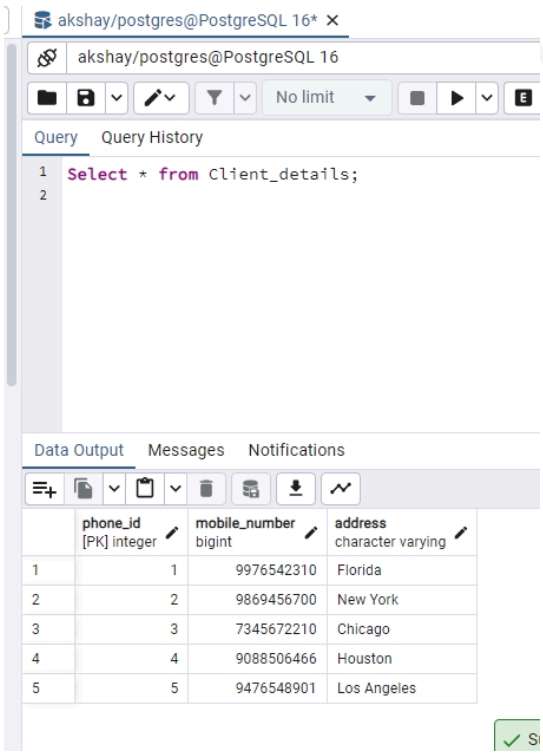
Once the ***Clieint_ details*** table has been created successfully, we will insert some values into it with the help of INSERT command as we can see in the following command:

```
INSERT INTO Client_details (Phone_id,Mobile_number,address)

VALUES (1, 9976542310,'Florida'),

(2, 9869456700,'New York'),

(3, 7345672210,'Chicago'),

(4, 9088506466,'Houston'),

(5, 9476548901,'Los Angeles');
```

After **creating and inserting** the values in the **Client_details** table, we will get the following output on executing the below command:

```
Select * from Client_details;
```

Table3: Client_details

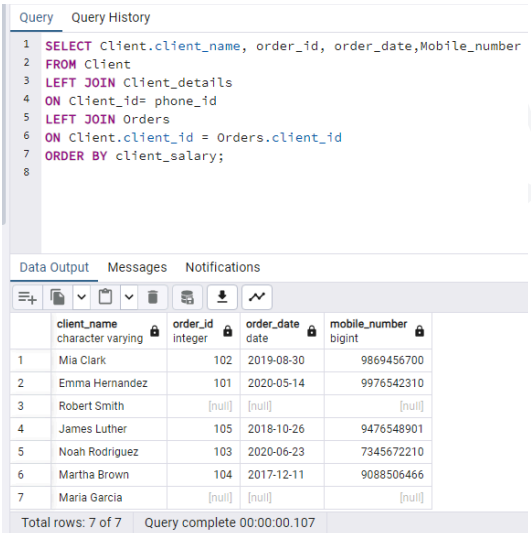


Now, we will join multiple tables such as Client, ***Orders, and Client_details*** with the help of **Left Join** as we can see in the following statement:

```
SELECT Client.client_name, order_id, order_date,Mobile_number
FROM Client
LEFT JOIN Client_details
ON Client_id= phone_id
LEFT JOIN Orders
ON Client.client_id = Orders.client_id
ORDER BY client_salary;
```

Output

After successful execution of the above command, we will give the below result:



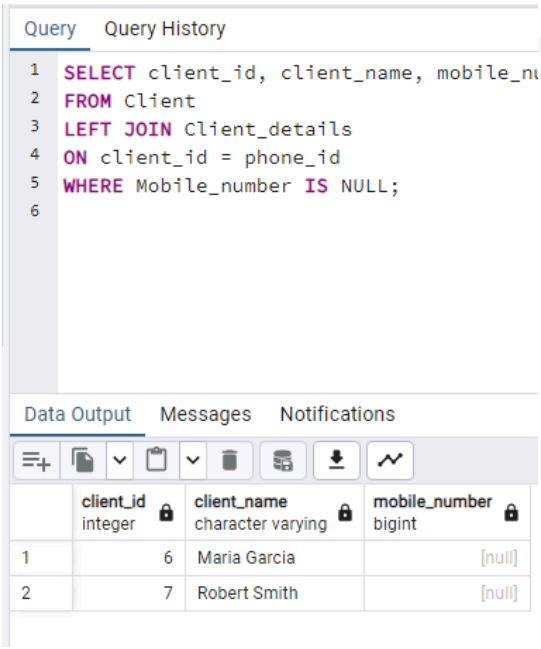
To get unmatched records by using of Left Join clause

If we want to get the data from the table, which does not contain any similar row of data from another table, so in those cases, we will use the PostgreSQL LEFT JOIN clause.

As we can see in the below example, the LEFT JOIN clause is used to identify a **client** who does not have a **Mobile_number**:

```
SELECT client_id, client_name, mobile_number

FROM Client

LEFT JOIN Client_details

ON client_id = phone_id

WHERE Mobile_number IS NULL;
```

Output



PostgreSQL Right Join

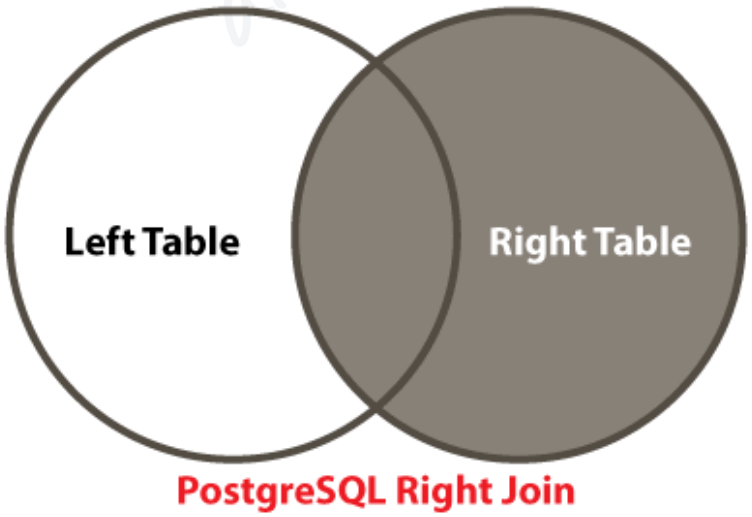
we are going to understand the working of **Right join**, which is used to return data from the Right table. We also learn how to use **table-aliasing, WHERE clause, USING clause, and join multiple tables** with the help of the **Right join clause**.

What is the Right Outer Join or Right Join clause?

The **PostgreSQL Right JOIN** or **Right Outer Join** is used to return all rows from the right table, and rows from the other table where the join condition is fulfilled defined in the ON condition. And if there are no corresponding records found from the **Left table**, it will return **null values**.

The **Right Join** can also be called as the **Right Outer Join** clause. Therefore, the **Outer** is the optional keyword, which is used in **Right Join**. In, the **Right join** is parallel to the **Left Join** condition, but it will give the opposite result of the join tables.

The following diagram displays the PostgreSQL Right join where we can easily understand that the **Right Join** returns all the data from the **Right table** and only a similar data from the **left table**:



Right Join Syntax

The Right Join keyword is used with the SELECT command and must be written after the **FROM** Keyword.

```
SELECT columns
FROM table1
Right [OUTER] JOIN table2
ON table1.column = table2.column;
```

In the above syntax, **table1** is referring to the **left table**, and **table2** is referring to the **right table**, which implies that the particular condition will return all records from **table 2** and matching records from **table1** according to the defined **join condition**.

We will follow the below steps to join the **Left and Right tables** with the help of the **Right or Right Outer Join** condition:

- Firstly, we will define the **column list** from both tables, where we want to select data in the **SELECT** condition.
- Then, we will define the **Right table**, which is **table 2** in the **FROM** clause.
- And lastly, we will describe the Left table, which is **table 1** in the **Right JOIN** condition, and write the join condition after the **ON** keyword.

NOTE: In the PostgreSQL Right Join, if the tables contain a similar column name, then USING and On clause produce the similar outputs.

Example of PostgreSQL Right join

To join two tables by using Right Join

For this, we will use the ***Employee and department*** table, which we created in the inner join section of the PostgreSQL tutorial.

Table1: Employee

Select * from Employee;

```
Select * from Employee;
```

Data Output | Messages | Notifications

	emp_id [PK] integer	emp_fname character varying	emp_lname character varying	location character varying (30)
1	1	John	Smith	New York
2	2	Mia	Clark	Florida
3	3	Noah	Rodriguez	Chicago
4	4	Ava	Gracia	Houston
5	5	James	Luther	Los Angeles

Table2: department

Select * from department;

```
Select * from department;
```

Data Output | Messages | Notifications

	emp_id [PK] integer	dept_id integer	dept_name character varying
1	1	1	ACCOUNTING
2	2	2	SALES
3	3	3	RESEARCH
4	4	4	OPERATIONS
5	5	5	HUMAN RESOURCES

The below query is used to select records from both tables (**Employee** and **department**):

```
SELECT department.emp_id,dept_name,location,emp_fname,
emp_lname

FROM Employee

Right JOIN department

ON department.emp_id = Employee.emp_id

order by emp_id;
```

Output

```
1  Select * from department;
2  SELECT department.emp_id,dept_name,location,emp_fname, emp_lname
3  FROM Employee
4  Right JOIN department
5  ON department.emp_id = Employee.emp_id
6  order by emp_id;
7
```

Data Output | Messages | Notifications

	emp_id integer	dept_name character varying	location character varying (30)	emp_fname character varying	emp_lname character varying
1	1	ACCOUNTING	New York	John	Smith
2	2	SALES	Florida	Mia	Clark
3	3	RESEARCH	Chicago	Noah	Rodriguez
4	4	OPERATIONS	Houston	Ava	Gracia
5	5	HUMAN RESOURCES	Los Angeles	James	Luther

Note: We can say that the RIGHT JOIN selects all rows from the right table even if they do not have similar rows from the left table.

Table-aliasing with Right Join

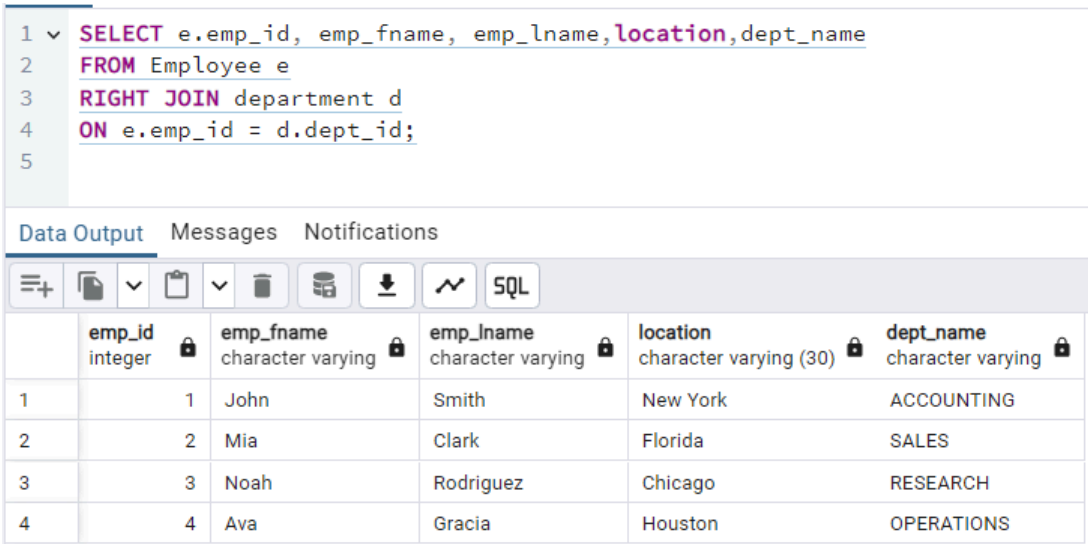
Generally, the tables we want to join will have columns with a similar name like **the emp_id** column.

We will use **table aliases** to assign the joined tables short names to make the command more understandable.

In the below command, we will use the table aliasing, and it returns a similar outcome as above:

```
SELECT e.emp_id, emp_fname, emp_lname,location,dept_name
FROM Employee e
RIGHT JOIN department d
ON e.emp_id = d.dept_id;
```

Output



```
SELECT e.emp_id, emp_fname, emp_lname,location,dept_name
FROM Employee e
RIGHT JOIN department d
ON e.emp_id = d.dept_id;
```

Data Output | Messages | Notifications

	emp_id integer	emp_fname character varying	emp_lname character varying	location character varying (30)	dept_name character varying
1	1	John	Smith	New York	ACCOUNTING
2	2	Mia	Clark	Florida	SALES
3	3	Noah	Rodriguez	Chicago	RESEARCH
4	4	Ava	Gracia	Houston	OPERATIONS

Right join with USING Clause

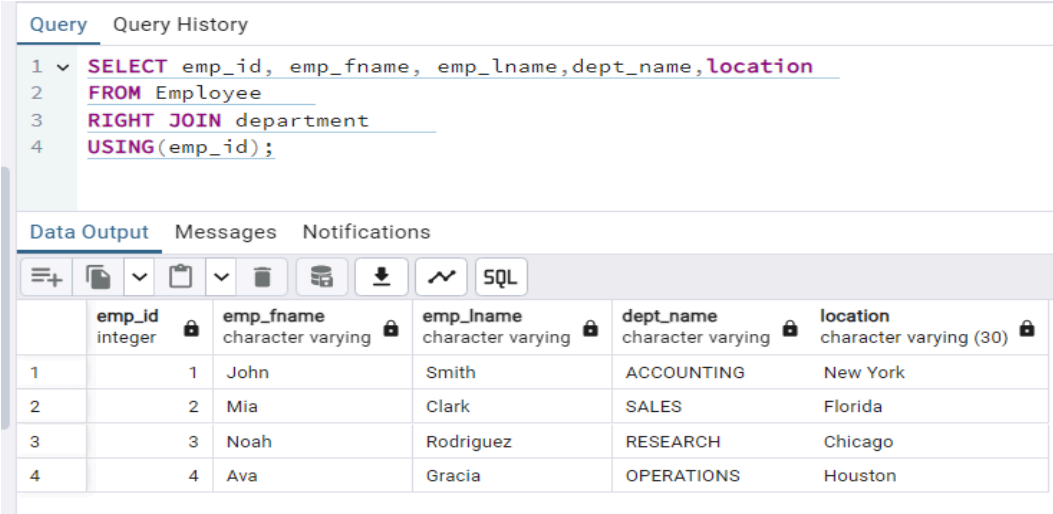
In this, we will see how the Right join works with the USING clause.

For example, the above tables ***Employee and department*** contain a similar column, which is emp_id; Thus, in those cases, we are using the USING clause to get the values from the tables.

In the following command, we are using the **USING** clause in the **Right join**, which returns the values **emp_id, emp_fname, emp_lname, dept_name, and location** as both tables have a similar column: **emp_id**.

```
SELECT emp_id, emp_fname, emp_lname,dept_name,location
FROM Employee
RIGHT JOIN department
USING(emp_id);
```

Output



Query | Query History

```
1 SELECT emp_id, emp_fname, emp_lname,dept_name,location
2 FROM Employee
3 RIGHT JOIN department
4 USING(emp_id);
```

Data Output | Messages | Notifications

	emp_id integer	emp_fname character varying	emp_lname character varying	dept_name character varying	location character varying (30)
1	1	John	Smith	ACCOUNTING	New York
2	2	Mia	Clark	SALES	Florida
3	3	Noah	Rodriguez	RESEARCH	Chicago
4	4	Ava	Gracia	OPERATIONS	Houston

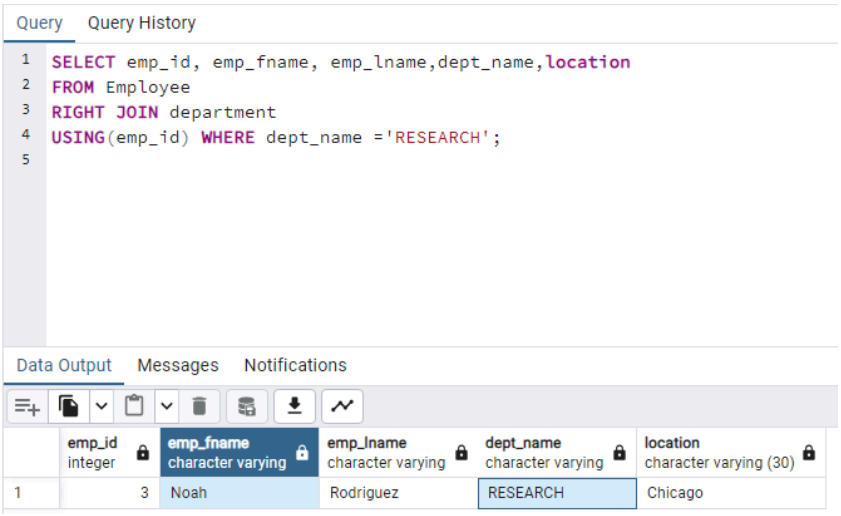
Right Join using WHERE clause

If we want to identify the rows from the right table(**department**) that does not have any matching rows in the left table (**Employee**), we can use the WHERE condition with the **Right join**.

As we can see in below command, we are selecting the rows from both tables **Employee** and ***departmen***t where **dept_name** is equal to **'RESEARCH'**:

```
SELECT emp_id, emp_fname, emp_lname,dept_name,location
FROM Employee
RIGHT JOIN department
USING(emp_id) WHERE dept_name ='RESEARCH';
```

Output



To join multiple tables using Right Join

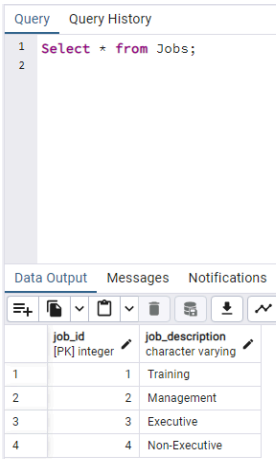
In the above section, we have two tables as ***Employee and department*** now, if we want to join more than two tables and get the records from that particular table. In that case, we will use the Right join.

For example, here we will take the **Jobs** table, which we created in the PostgreSQL Inner Join section of the tutorial.

To see the **Jobs** table's values, we will use the **SELECT** clause as follows:

```
Select * from Jobs;
```

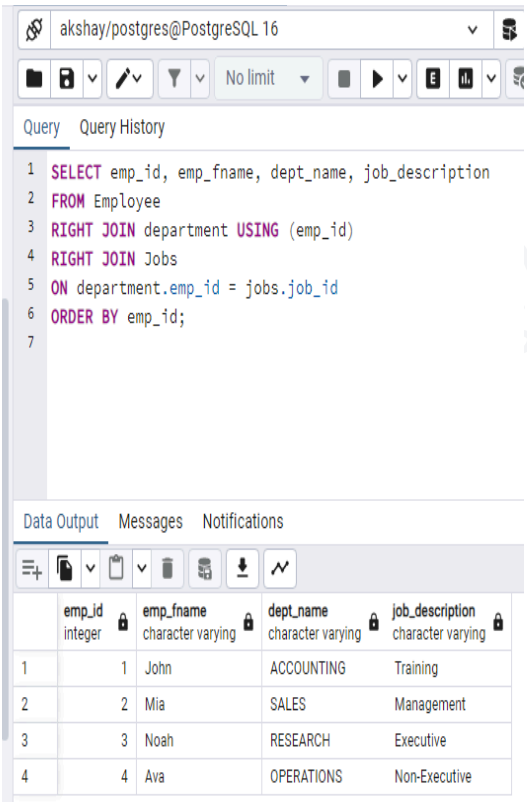
Table3: Jobs



We will join three tables such as ***Employee, department, and Jobs*** with the help of **PostgreSQL Right Join** as we can see in the following command:

```
SELECT emp_id, emp_fname, dept_name, job_description

FROM Employee

RIGHT JOIN department USING (emp_id)

RIGHT JOIN Jobs

ON department.emp_id = jobs.job_id

ORDER BY emp_id;
```

Output



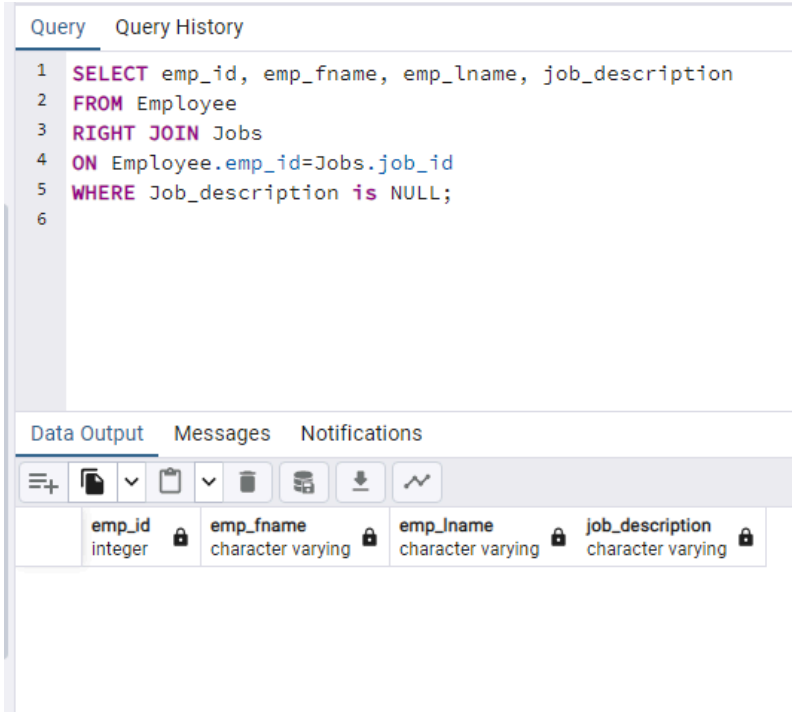
To get unmatched records by using of Right JOIN clause

If we want to get the data from the table, which does not contain any similar row of data from other tables, so in those cases, we will use the PostgreSQL Right Join clause.

As we can see in the below example, the Right join clause is used to identify an **employee** whose **Job_description** is **Null**:

```
SELECT emp_id, emp_fname, emp_lname, job_description
FROM Employee
RIGHT JOIN Jobs
ON Employee.emp_id=Jobs.job_id
WHERE Job_description is NULL;
```

Output



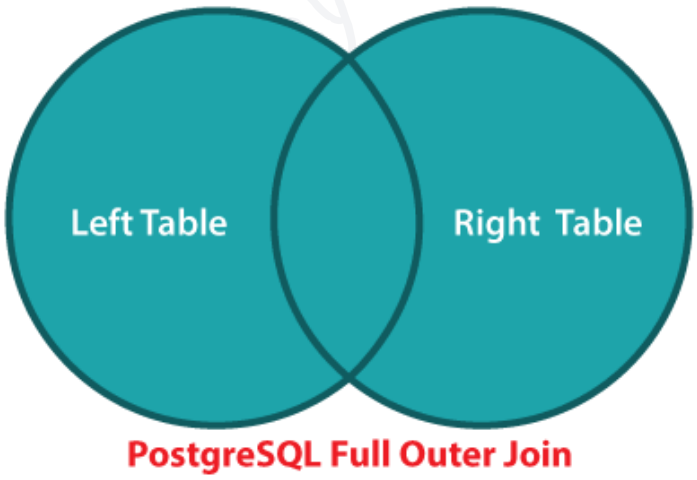
PostgreSQL Full Join

we are going to understand the working of **Full join**, which is used to return all records when there is a match in the **left table** or **right table** records. We also learn how to use **table-aliasing, WHERE clause** with the help of the **Full Outer join clause**.

What is the Full Join or Full Outer Join clause?

The PostgreSQL **Full Join** or **Full Outer Join** is used to return all records when there is a match in the **left table** or **right table** records. The main objective of a **Full Outer Join** is that it will combine the outcome of Left Join and Right Join clauses and returns all similar or unmatched rows from the tables on both sides of the join clause.

The following diagram displays the **PostgreSQL Full Outer Join** where we can easily understand that the **Full Outer Join** returns all the data from both the **Left table** and **Right table**:



Syntax of Full Outer Join

The syntax for Full Outer Join or Full Join is as following:

```
SELECT columns
FROM table1
```

```
FULL [OUTER] JOIN table2
ON table1.column = table2.column;
```

In the above syntax, The **Full Outer Join** keyword is used with the SELECT command and must be written after the **FROM** Keyword, and the **OUTER** keyword is optional.

We will follow the below steps to combine the **Left and Right tables** with the help of the **Full Join or Full Outer Join** condition:

- Firstly, we will define the **column list** from both tables, where we want to select data in the **SELECT** condition.
- Then, we will specify the **Right table**, which is **table 2** in the **FROM** clause.
- And lastly, we will describe the Left table, which is **table 1** in the **Full Outer Join** clause, and write the join condition after the **ON** keyword.

Example of Full Join

Let us see an example to understand how the **PostgreSQL Full Outer join** works:

To join two tables by using Full Outer Join

For this, we will create two tables named ***Summer_fruits*** and ***Winter_fruits*** table with the help of the CREATE command and insert some values using the INSERT command.

Firstly, we are going to create ***Summer_fruits*** and ***Winter_fruits*** tables by using the CREATE command:

```
CREATE TABLE Summer_fruits (
SF_ID INT PRIMARY KEY,
Summer_fruits_name VARCHAR (250) NOT NULL);
```

The below command is used to create the ***Winter_fruits*** table:

```
CREATE TABLE Winter_fruits (
WF_ID INT PRIMARY KEY,
Winter_fruits_name VARCHAR (250) NOT NULL);
```

The ***Summer_fruits*** and ***Winter_fruits*** tables have been successfully created on executing the above commands.

Once both the tables have been generated, we are ready to insert some values into it by using the INSERT command as follows:

```
INSERT INTO Summer_fruits (SF_ID, Summer_fruits_name)
VALUES
(1,'Mango'),
(2,'Watermelon'),
(3,'Apples'),
(4,'Guava'),
(5,'Pineapple'),
(6,'Musk Melon');
```


In the below command, we are inserting the values in the ***Winter_fruits*** table:

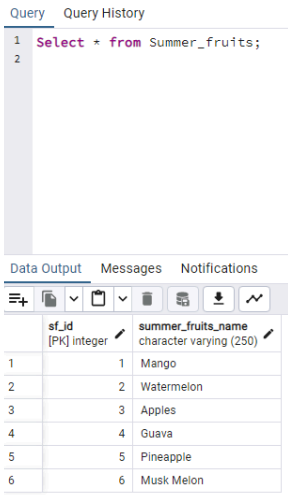
```
INSERT INTO Winter_fruits (WF_ID, Winter_fruits_name)
VALUES(1,'Grape'),
(2,'Apples'),
(3,'Mango'),
(4,'Pears'),
(5,'Pineapple'),
(6,'Cranberries'),
```

(7,'Bananas');

Table1: Summer_fruits

Select * from Summer_fruits;

Output



Query | Query History

```
Select * from Summer_fruits;
```

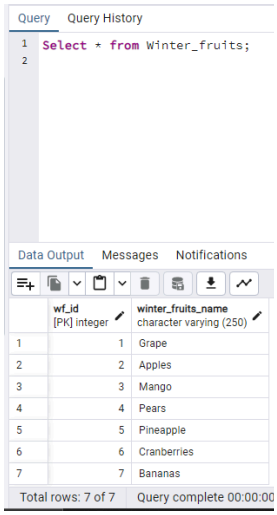
Data Output | Messages | Notifications

	sf_id [PK] integer	summer_fruits_name character varying (250)
1	1	Mango
2	2	Watermelon
3	3	Apples
4	4	Guava
5	5	Pineapple
6	6	Musk Melon

Table2: Winter_fruits

Select * from Winter_fruits;

Output



Query | Query History

```
Select * from Winter_fruits;
```

Data Output | Messages | Notifications

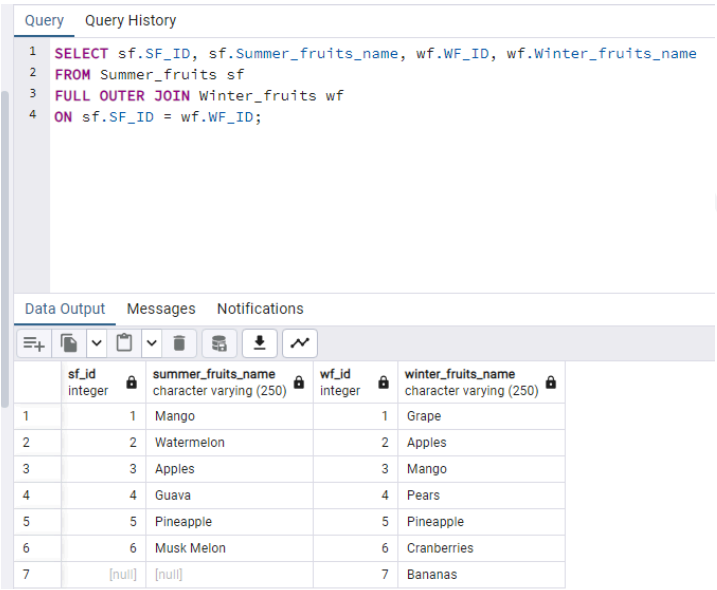
	wf_id [PK] integer	winter_fruits_name character varying (250)
1	1	Grape
2	2	Apples
3	3	Mango
4	4	Pears
5	5	Pineapple
6	6	Cranberries
7	7	Bananas

Total rows: 7 of 7 | Query complete 00:00:00

The below query is used to select records from both tables (**Summer_fruits** and **Winter_fruits**):

```
SELECT sf.SF_ID, sf.Summer_fruits_name, wf.WF_ID,
wf.Winter_fruits_name

FROM Summer_fruits sf

FULL OUTER JOIN Winter_fruits wf

ON sf.SF_ID = wf.WF_ID;
```

Output



	sf_id integer	summer_fruits_name character varying (250)	wf_id integer	winter_fruits_name character varying (250)
1	1	Mango	1	Grape
2	2	Watermelon	2	Apples
3	3	Apples	3	Mango
4	4	Guava	4	Pears
5	5	Pineapple	5	Pineapple
6	6	Musk Melon	6	Cranberries
7	[null]	[null]	7	Bananas

Working of PostgreSQL Full Outer Join

- It is used to get the records of both the **left table (Summer_fruits)** and the **right table (Winter_fruits)**.
- If a row from **Summer_fruits**, which is **Table1 or Left Table** matches a row in **Winter_fruits**, which is **Table2 or Right table**, then the result row will contain columns that came from columns of rows from both tables.

- If the rows in the joined tables are not similar, then the **Full Outer Join** places **NULL values** for every column of the table.

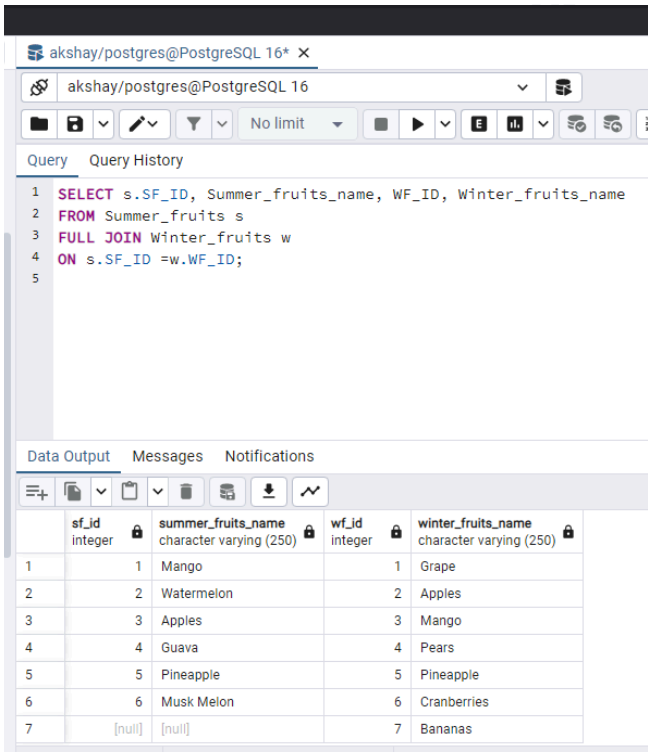
Table-aliasing with Full Join

We will use **table aliases** to assign the joined tables short names to make the command more understandable because sometimes writing the complete table lead us to tedious processes.

In the below command, we will use the table aliasing, and it returns a similar outcome:

```
SELECT s.SF_ID, Summer_fruits_name, WF_ID, Winter_fruits_name
FROM Summer_fruits s
FULL JOIN Winter_fruits w
ON s.SF_ID =w.WF_ID;
```

Output



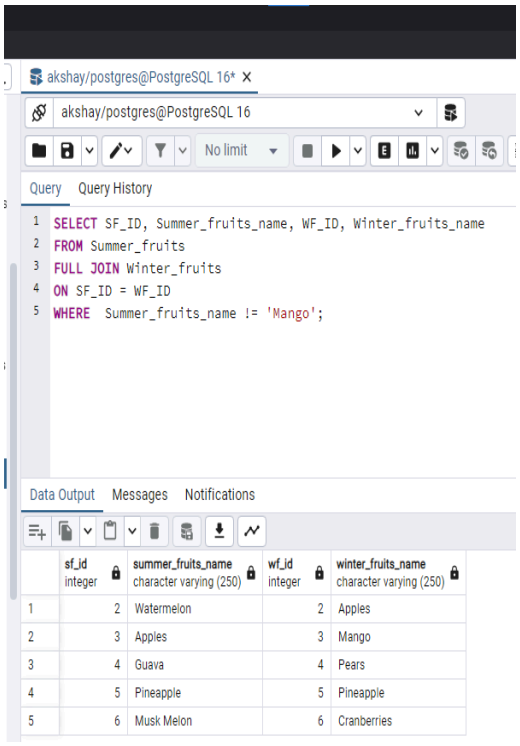
Full Join using where clause

We can also use the **Full join** with a WHERE condition. The WHERE clause allows us to return the filter outcome.

In the below example, we will select rows from both tables ***Summer_fruits*** and ***Winter_fruits*** where **Summer_fruits_names** is not equal to **Mango**:

```
SELECT SF_ID, Summer_fruits_name, WF_ID, Winter_fruits_name

FROM Summer_fruits

FULL JOIN Winter_fruits

ON SF_ID = WF_ID

WHERE  Summer_fruits_name != 'Mango';
```

Output



PostgreSQL Cross Join

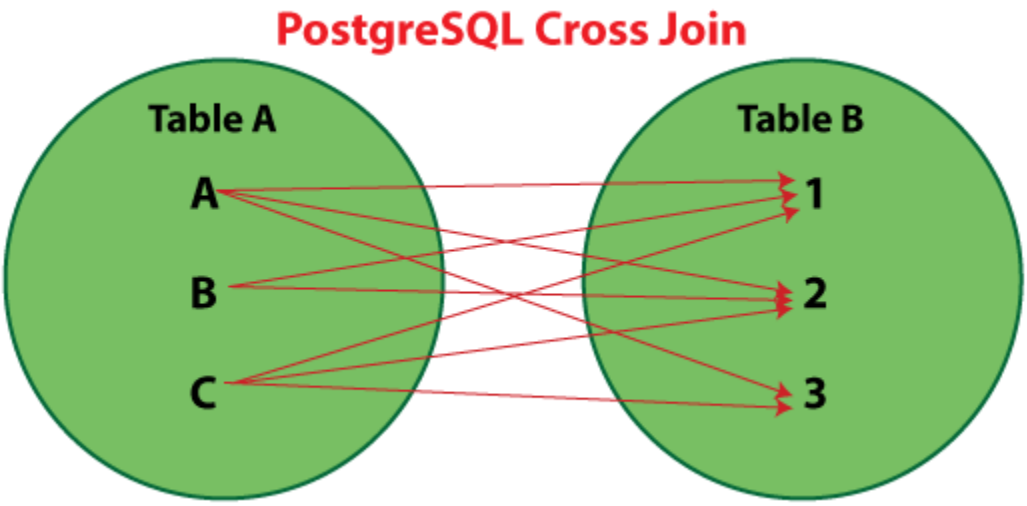
in this section, we are going to understand the working of **Cross join**, which allows us to create a Cartesian Product of rows in two or more tables. We also learn how to use **table-aliasing, WHERE clause, and join multiple tables** with the help of the **Cross Join clause**.

What is Cross Join?

The **Cross Join** is used to combine all possibilities of the multiple tables and returns the output, which contains each row from all the selected tables. The **CROSS JOIN**, further known as **CARTESIAN JOIN** that allows us to produce the Cartesian product of all related tables.

The **Cartesian product** can be described as all existing rows in the first table multiplied by all rows in the second table. It is parallel to the **Inner Join**, where the join condition is not existing with this clause.

The following diagram displays the **Cross Join,** where we can easily understand that the **Cross Join** returns all the records from **Table1 and Table2**, and each row is the grouping of rows from both tables.



Cross Join Syntax

The **Cross-Join** keyword is used with the SELECT command and must be written after the **FROM** Keyword. The below syntaxes are used to get all the data from both associated tables:

Syntax1

```
SELECT column-lists

FROM Table1

CROSS JOIN Table2;
```

Syntax2

The below syntax is similar to the above syntax as we did not use the Cross Join keyword:

```
SELECT [column_list|*]

FROM Table1, Table2;
```

Syntax3

Here, we can use an INNER JOIN clause with the condition that always analyzes toward exact duplicate of the cross join:

```
SELECT *

FROM Table1

INNER JOIN Table2 ON true;
```

In the above syntax's, we have the following parameters:

In the above syntax's, we have the following parameters:

Parameter	Description
Column-lists	The column-list is used to specify the name of the column or field, which we want to return.
Table1 and Table2	These are the table names from which we get the records.

Example of Cross join

Let us see an example to understand how the **Cross join** works:

To join two tables by using Cross Join

For this, we will use the ***Summer_fruits and Winter_fruits*** table, which we created in the Full join section

Table1: Summer_fruits

```
Select * from Summer_fruits;
```

Output

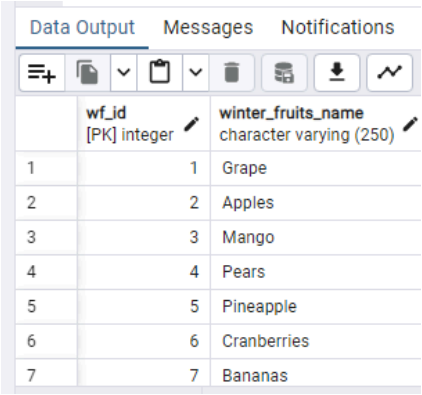
Data Output | Messages | Notifications

	sf_id [PK] integer	summer_fruits_name character varying (250)
1	1	Mango
2	2	Watermelon
3	3	Apples
4	4	Guava
5	5	Pineapple
6	6	Musk Melon

Table2: Winter_fruits

```
Select * from Winter_fruits;
```

Output



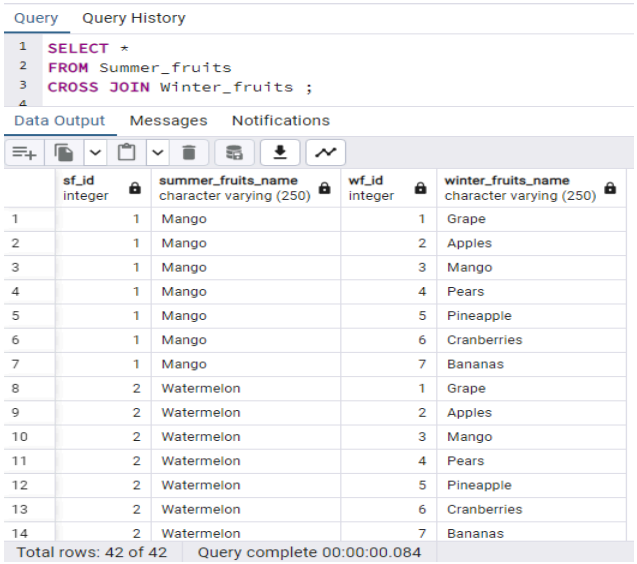
Data Output | Messages | Notifications

	wf_id [PK] integer	winter_fruits_name character varying (250)
1	1	Grape
2	2	Apples
3	3	Mango
4	4	Pears
5	5	Pineapple
6	6	Cranberries
7	7	Bananas

We will execute the below command to get all records from both tables (***Summer_fruits and Winter_fruits***):

```
SELECT *

FROM Summer_fruits

CROSS JOIN Winter_fruits ;
```

Output



Query | Query History

```
SELECT *
FROM Summer_fruits
CROSS JOIN Winter_fruits ;
```

Data Output | Messages | Notifications

	sf_id integer	summer_fruits_name character varying (250)	wf_id integer	winter_fruits_name character varying (250)
1	1	Mango	1	Grape
2	1	Mango	2	Apples
3	1	Mango	3	Mango
4	1	Mango	4	Pears
5	1	Mango	5	Pineapple
6	1	Mango	6	Cranberries
7	1	Mango	7	Bananas
8	2	Watermelon	1	Grape
9	2	Watermelon	2	Apples
10	2	Watermelon	3	Mango
11	2	Watermelon	4	Pears
12	2	Watermelon	5	Pineapple
13	2	Watermelon	6	Cranberries
14	2	Watermelon	7	Bananas

Total rows: 42 of 42 | Query complete 00:00:00.084

When the **CROSS-JOIN** command is executed, we will see that it shows 42 rows, which implies that the **Six** rows from the ***Summer_fruites*** table multiply by the **Seven** rows from the ***Winter_fruits*** table.

Note: It is suggested to use separate column names in its place of SELECT * command to avoid the output of repetitive columns two times.

Uncertain Columns problem in CROSS JOIN

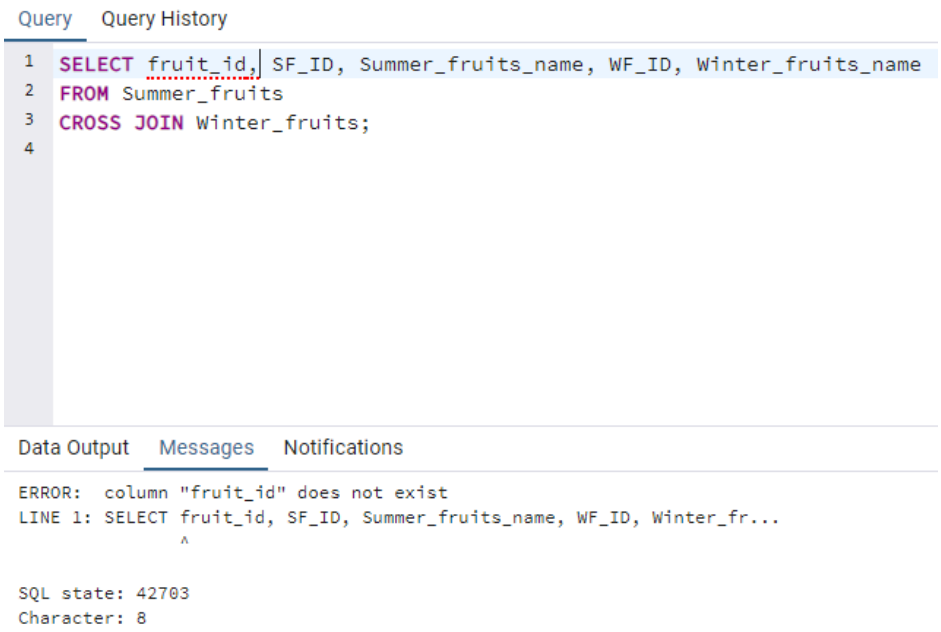
Sometimes, we need to get the selected column records from more than two tables. And these tables can have some matching column names.

Let see one **example** to understand this type of case, suppose the ***Summer_fruits and Winter_fruits*** table contain one similar column that's is: **fruit_id** as we can see in the below command:

```
SELECT fruit_id, SF_ID, Summer_fruits_name, WF_ID,
Winter_fruits_name

FROM Summer_fruits

CROSS JOIN Winter_fruits;
```

Output

On executing the above command, the PostgreSQL CROSS JOIN command throws an error, which is **The column name is ambiguous**, and it implies that the name of the column exists in both tables. PostgreSQL becomes unclear about which column we want to display.



Therefore, to solve the above error, we will specify the table name before the column name as we can see in the below command:

Create tables

-- Create Summer_fruits table

CREATE TABLE Summer_fruits (

fruit_id INTEGER,

SF_ID INTEGER,

Summer_fruits_name VARCHAR(250)

);

-- Insert data into Summer_fruits table

INSERT INTO Summer_fruits (fruit_id, SF_ID, Summer_fruits_name)

VALUES

(1, 1, 'Mango'),

```
  (2, 1, 'Mango'),
  (3, 1, 'Mango'),
  (4, 1, 'Mango'),
  (5, 1, 'Mango'),
  (6, 1, 'Mango'),
  (7, 1, 'Mango'),
  (8, 2, 'Mango'),
  (9, 2, 'Watermelon'),
  (10, 2, 'Watermelon'),
  (11, 2, 'Watermelon');

-- Create Winter_fruits table
CREATE TABLE Winter_fruits (
  WF_ID INTEGER,
  Winter_fruits_name VARCHAR(250)
);

-- Insert data into Winter_fruits table
INSERT INTO Winter_fruits (WF_ID, Winter_fruits_name)
VALUES
  (1, 'Grape'),
  (2, 'Apples'),
  (3, 'Mango'),
```

(4, 'Pears'),

(5, 'Pineapple'),

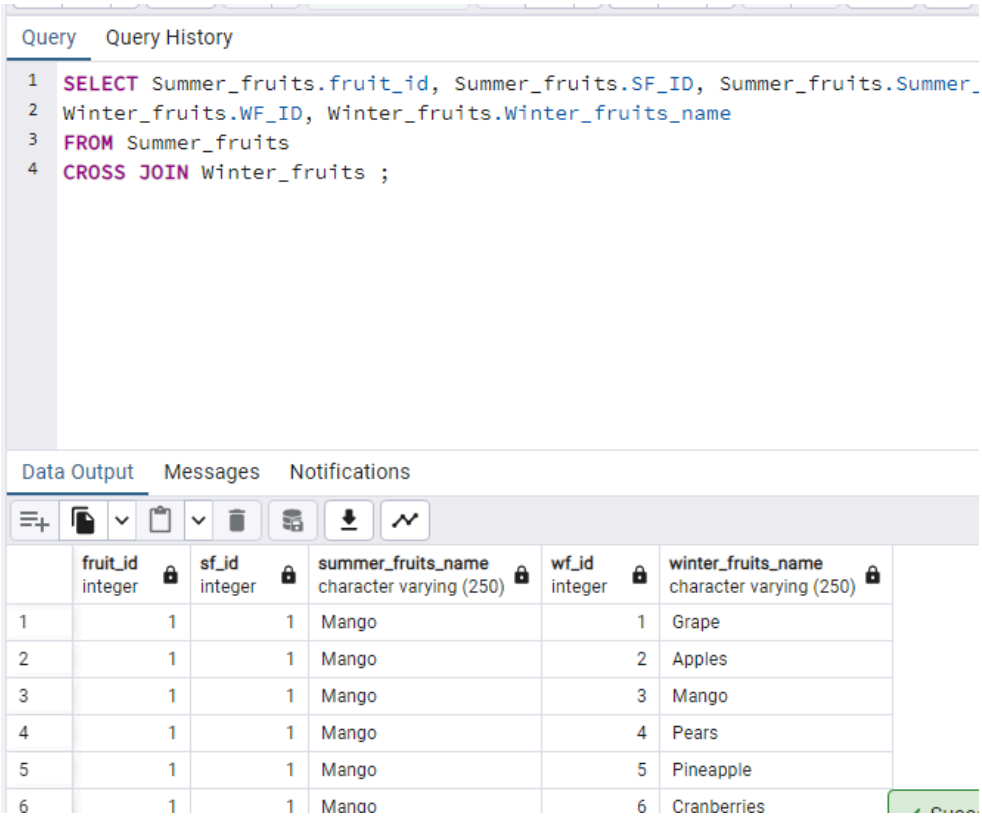
(6, 'Cranberries'),

(7, 'Bananas');

solve the above error

```
SELECT Summer_fruits.fruit_id, Summer_fruits.SF_ID,
Summer_fruits.Summer_fruits_name,

Winter_fruits.WF_ID, Winter_fruits.Winter_fruits_name

FROM Summer_fruits

CROSS JOIN Winter_fruits ;
```

Output



	fruit_id integer	sf_id integer	summer_fruits_name character varying (250)	wf_id integer	winter_fruits_name character varying (250)
1	1	1	Mango	1	Grape
2	1	1	Mango	2	Apples
3	1	1	Mango	3	Mango
4	1	1	Mango	4	Pears
5	1	1	Mango	5	Pineapple
6	1	1	Mango	6	Cranberries

Table-aliasing with Cross Join

Generally, the tables we want to join will have columns with a similar name like **the fruit_id** column.

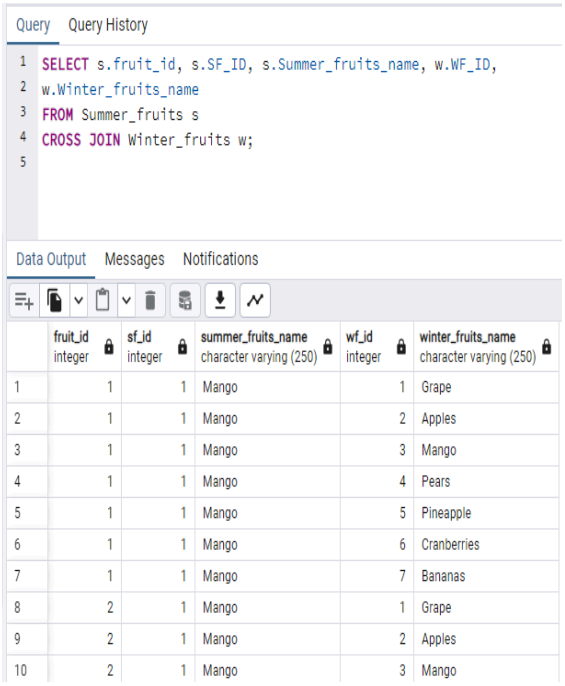
Instead of using the complete table name, we can use **table aliases** to assign the joined tables short names to make the command more understandable. Sometimes, writing a **full table name** is a tedious process.

Thus, we will use the **table aliasing,** and it returns a similar outcome as above as we can see in the below command:

```
SELECT s.fruit_id, s.SF_ID, s.Summer_fruits_name, w.WF_ID,
w.Winter_fruits_name
FROM Summer_fruits s
CROSS JOIN Winter_fruits w;
```

Output

Once we implemented the above command, we will get the below output:



Query | Query History

```
1  SELECT s.fruit_id, s.SF_ID, s.Summer_fruits_name, w.WF_ID,
2  w.Winter_fruits_name
3  FROM Summer_fruits s
4  CROSS JOIN Winter_fruits w;
5
```

Data Output | Messages | Notifications

	fruit_id integer	sf_id integer	summer_fruits_name character varying (250)	wf_id integer	winter_fruits_name character varying (250)
1	1	1	Mango	1	Grape
2	1	1	Mango	2	Apples
3	1	1	Mango	3	Mango
4	1	1	Mango	4	Pears
5	1	1	Mango	5	Pineapple
6	1	1	Mango	6	Cranberries
7	1	1	Mango	7	Bananas
8	2	1	Mango	1	Grape
9	2	1	Mango	2	Apples
10	2	1	Mango	3	Mango

Cross Join using WHERE Clause

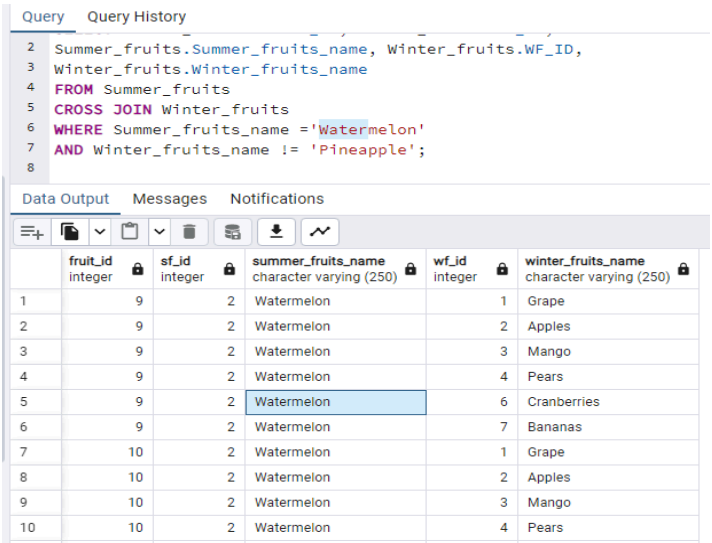
If we want to identify the rows from Table1 (**Summer_fruits**) that do not have any matching rows in Table2 (**Winter_fruits**), we can use the WHERE condition with the **Cross Join**.

As we can see in the below command, we are selecting the rows from both tables **Summer_fruits and Winter_fruits** where **Summer_fruits_name** is equal to **Watermelon** and **Winter_fruits_name** is not equal to **Pineapple**.

```
SELECT Summer_fruits.fruit_id, Summer_fruits.SF_ID,
Summer_fruits.Summer_fruits_name, Winter_fruits.WF_ID,
Winter_fruits.Winter_fruits_name
FROM Summer_fruits
CROSS JOIN Winter_fruits
WHERE Summer_fruits_name ='Watermelon'
AND Winter_fruits_name != 'Pineapple';
```

Output

On executing the above command, we will get the following result:



Query | Query History

```
Summer_fruits.Summer_fruits_name, Winter_fruits.WF_ID,
Winter_fruits.Winter_fruits_name
FROM Summer_fruits
CROSS JOIN Winter_fruits
WHERE Summer_fruits_name ='Watermelon'
AND Winter_fruits_name != 'Pineapple';
```

Data Output | Messages | Notifications

	fruit_id integer	sf_id integer	summer_fruits_name character varying (250)	wf_id integer	winter_fruits_name character varying (250)
1	9	2	Watermelon	1	Grape
2	9	2	Watermelon	2	Apples
3	9	2	Watermelon	3	Mango
4	9	2	Watermelon	4	Pears
5	9	2	Watermelon	6	Cranberries
6	9	2	Watermelon	7	Bananas
7	10	2	Watermelon	1	Grape
8	10	2	Watermelon	2	Apples
9	10	2	Watermelon	3	Mango
10	10	2	Watermelon	4	Pears

To join multiple tables using Cross JOIN

In the above section, we have two tables as ***Summer_fruits and Winter_fruits*** now, if we want to join more than two tables and get the records from that particular table. In that case, we will use the **Cross join**.

For example, we will create **Fruite_sales** table by using Create Clause as we can see in the following command:

```
CREATE TABLE Fruit_sales (
Fruit_id int primary key,
Sales_id int,
Fruits_name varchar not null
);
```

To see the **Fruit_sales** table's values, we will use the **SELECT** clause as follows:

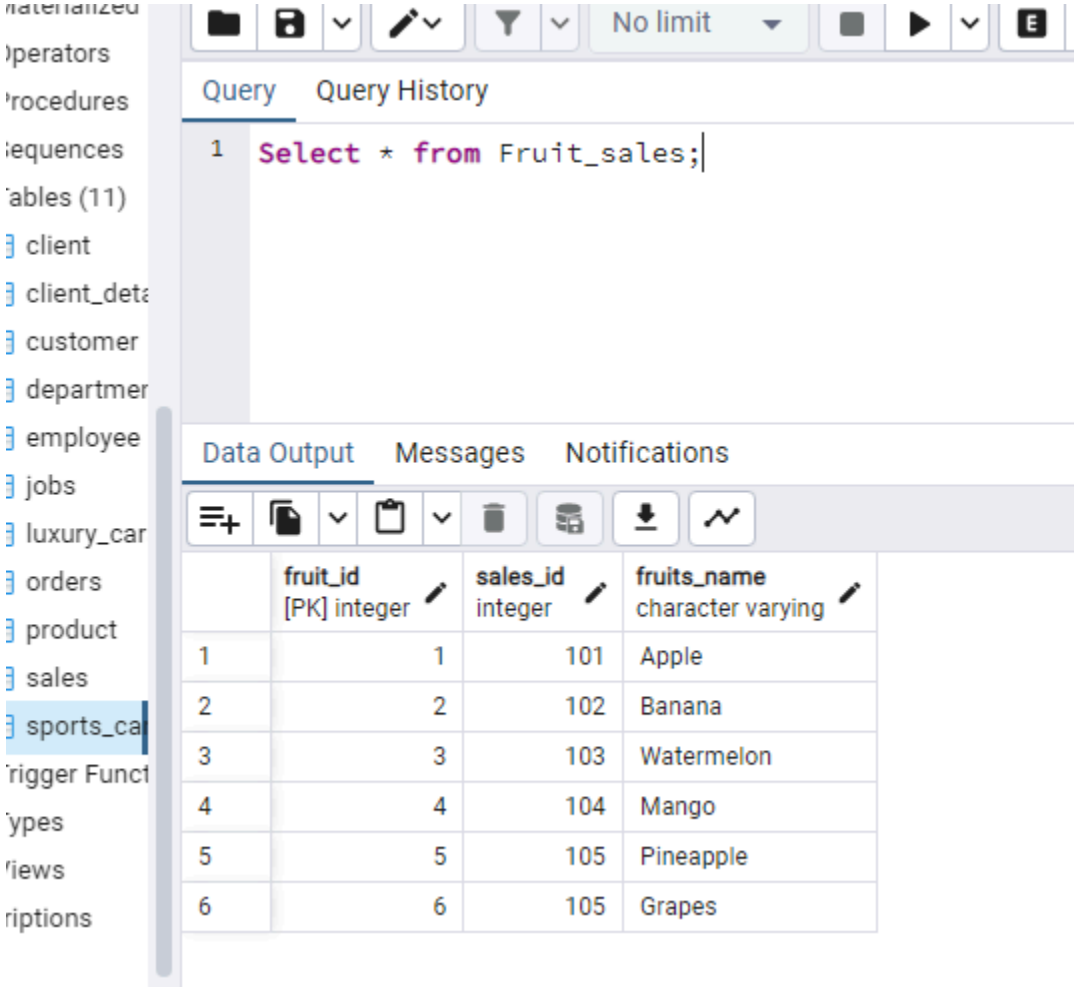
Once the ***Fruit_sales*** table has been created successfully, we will insert some values into it with the help of INSERT command as we can see in the following command:

```
INSERT INTO Fruit_sales (fruit_id, Sales_id, Fruits_name)
VALUES (1, 101,'Apple'),
(2, 102,'Banana'),
(3, 103,'Watermelon'),
(4, 104,'Mango'),
(5, 105,'Pineapple'),
(6, 105,'Grapes');
```

After **creating and inserting** the values in the **Fruit_sales** table, we will get the following output on executing the below command:

```
Select * from Fruit_sales;
```

Table3: Fruit_sales



Now, we will join multiple tables such as ***Summer_fruits, Winter_fruits, and Fruit_sales*** with the help of **Cross Join** as we can see in the following statement:

```
SELECT * FROM Summer_fruits
LEFT JOIN (Winter_fruits CROSS JOIN Fruit_sales)
ON Summer_fruits.fruit_id= Fruit_sales.fruit_id
ORDER BY Fruits_name;
```

Output

Query | Query History | Scratch Pad

```
SELECT * FROM Summer_fruits
LEFT JOIN (Winter_fruits CROSS JOIN Fruit_sales)
ON Summer_fruits.fruit_id= Fruit_sales.fruit_id
ORDER BY Fruits_name;
```

Data Output | Messages | Notifications

	fruit_id integer	sf_id integer	summer_fruits_name character varying (250)	wf_id integer	winter_fruits_name character varying (250)	fruit_id integer	sales_id integer	fruits_name character varying
1	1	1	Mango	1	Grape	1	101	Apple
2	1	1	Mango	2	Apples	1	101	Apple
3	1	1	Mango	3	Mango	1	101	Apple
4	1	1	Mango	4	Pears	1	101	Apple
5	1	1	Mango	5	Pineapple	1	101	Apple
6	1	1	Mango	6	Cranberries	1	101	Apple
7	1	1	Mango	7	Bananas	1	101	Apple
8	2	1	Mango	1	Grape	2	102	Banana
9	2	1	Mango	2	Apples	2	102	Banana
10	2	1	Mango	3				

Successfully run. Total query runtime: 84 msec. 47 rows affected

PostgreSQL Self Join

we are going to understand the working of **Self joins**, which is used to relate rows within the same table. We also learned how to **get the hierarchical data from a similar table** with the help of the **Self join clause**.

What is Self Join?

In, we have one particular type of join, which is known as **Self Join**. The **" Self Join** is used to set the different names of a similar table completely, and we can use the aliases also.

To proceed a self-join, we will define a similar table two times with different table aliases and give the Join predicate after the **ON** keyword.

In real-time, we use a self-join to **compare rows within the same table** (because the comparison of similar table name is not allowed in), and to fetch the hierarchical data.

Note: There is no such keyword as Self Join; however, we can use the Inner Join, Left Join, Right Join with the help of aliases.

Self Join Syntax

we have different syntaxes for Self-Join, which are as follows:

Syntax1

In the below syntax, we use an **Inner Join Keyword,** which combines the table to itself:

```
SELECT column_list

FROM table_name Table1

INNER JOIN table_name Table2 ON join_predicate;
```

In the above syntax, the **table_name** is combined with the help of the INNER JOIN clause.

Syntax2

In the below syntax, we use the **Left Join Keyword**, which combines the table to itself:

```
SELECT column_list

FROM table_name Table1

LEFT JOIN table_name Table2 ON join_predicate;
```

In the above syntax, the **table_name** is combined itself with the help of the LEFT JOIN clause.

Syntax3

In the below syntax, we use the **Right Join Keyword**, which combines the table to itself:

```
SELECT column_list

FROM table_name Table1

RIGHT JOIN Table_name Table2 ON join_predicate;
```

In the above syntax, the **table_name** is combined itself with the help of the RIGHT JOIN clause.

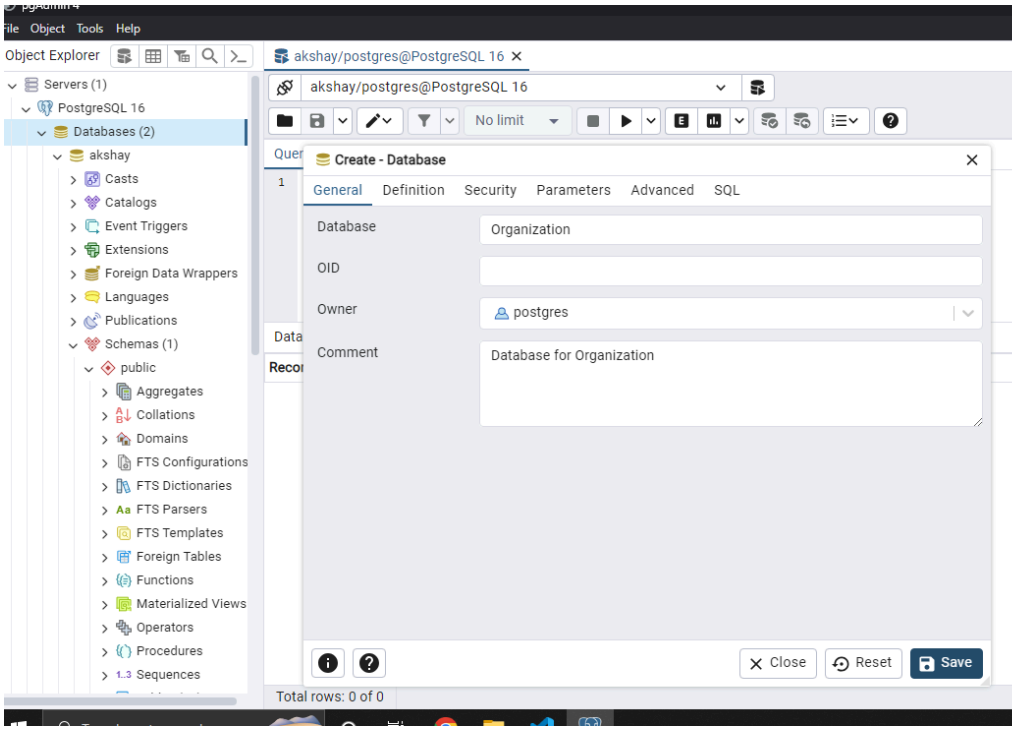
Example of Self join

Let us see an example to understand how the **Self join** works:

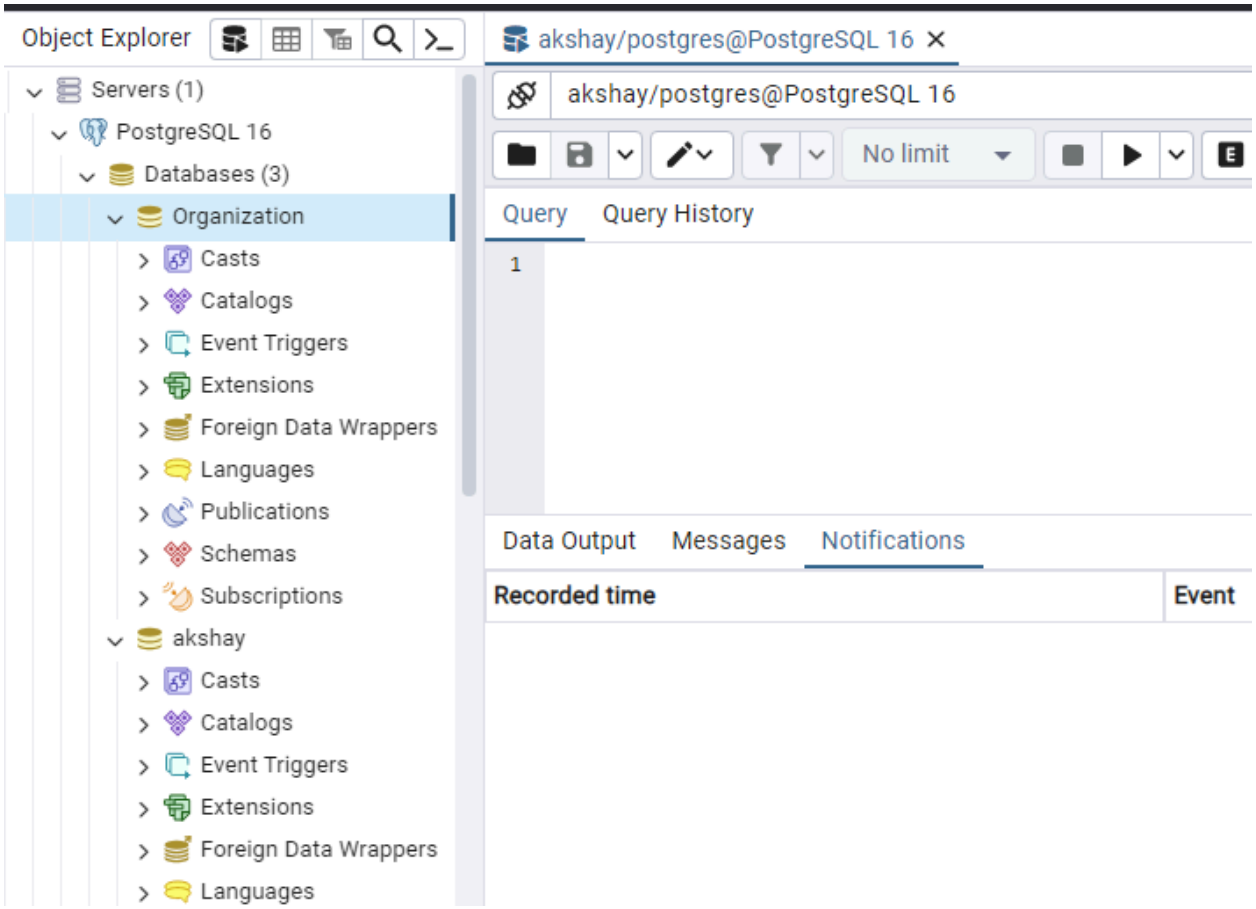
Example of fetching the Hierarchical Records from a table

Databases → Create → Database

- After that, the **create database** window will open where we need to provide some necessary details (**Database name, Comment**) for creating a database and then click on the **Save** button.



- The **Organization** database has been created successfully and display in Object tree as we can see in the below screenshot:



After the successful creation **Origination database**, we will create the **Customer** table by using the CREATE command as we can see in the below statement:

```
CREATE TABLE Customer (

Customer_id INT PRIMARY KEY,

First_name VARCHAR NOT NULL,

Last_name VARCHAR NOT NULL,

Order_id INT,

FOREIGN KEY (Order_id) REFERENCES Customer (Customer_id)

ON DELETE CASCADE

);
```

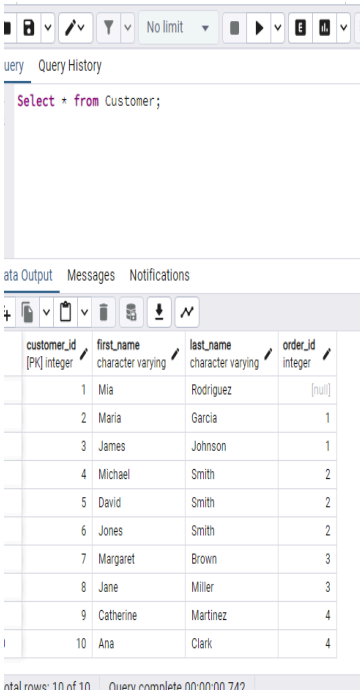
Now, we will insert some customer records into the **Customer** table with the help of INSERT as we can see in the below statement:

```
INSERT INTO Customer(Customer_id,First_name, Last_name,

Order_id )VALUES

(1, 'Mia', 'Rodriguez', NULL),

(2, 'Maria', 'Garcia', 1),

(3, 'James ', 'Johnson', 1),

(4, 'Michael', 'Smith', 2),

(5, 'David', 'Smith', 2),

(6, 'Jones', 'Smith', 2),

(7, 'Margaret', 'Brown', 3),

(8, 'Jane', 'Miller', 3),

(9, 'Catherine', 'Martinez', 4),

(10, 'Ana', 'Clark', 4);
```

After creating and inserting the values in the ***Customer*** table, we will get the following output on executing the below command:

```
Select * from Customer;
```

Output



```
Select * from Customer;
```

ata Output | Messages | Notifications

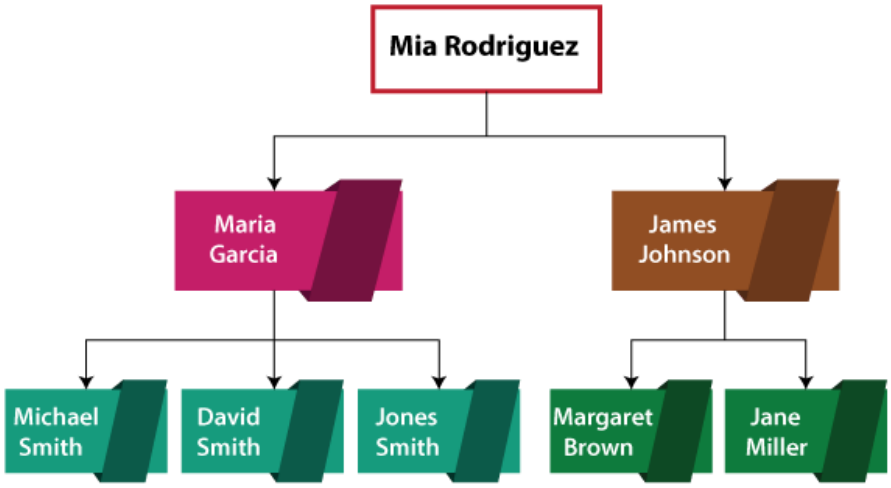
customer_id [PK] integer	first_name character varying	last_name character varying	order_id integer
1	Mia	Rodriguez	[null]
2	Maria	Garcia	1
3	James	Johnson	1
4	Michael	Smith	2
5	David	Smith	2
6	Jones	Smith	2
7	Margaret	Brown	3
8	Jane	Miller	3
9	Catherine	Martinez	4
10	Ana	Clark	4

otal rows: 10 of 10 Query complete 00:00:00.742

In the **Customer** table, the **Order_id** column references the **Customer_id** column. The value in the **Order_id** column displays the Order that is purchased by the Customer.

The customer does not purchase anything if the value is **Null** in the **Order_id** column.

As we can see in the following image that the overall hierarchy looks like this:

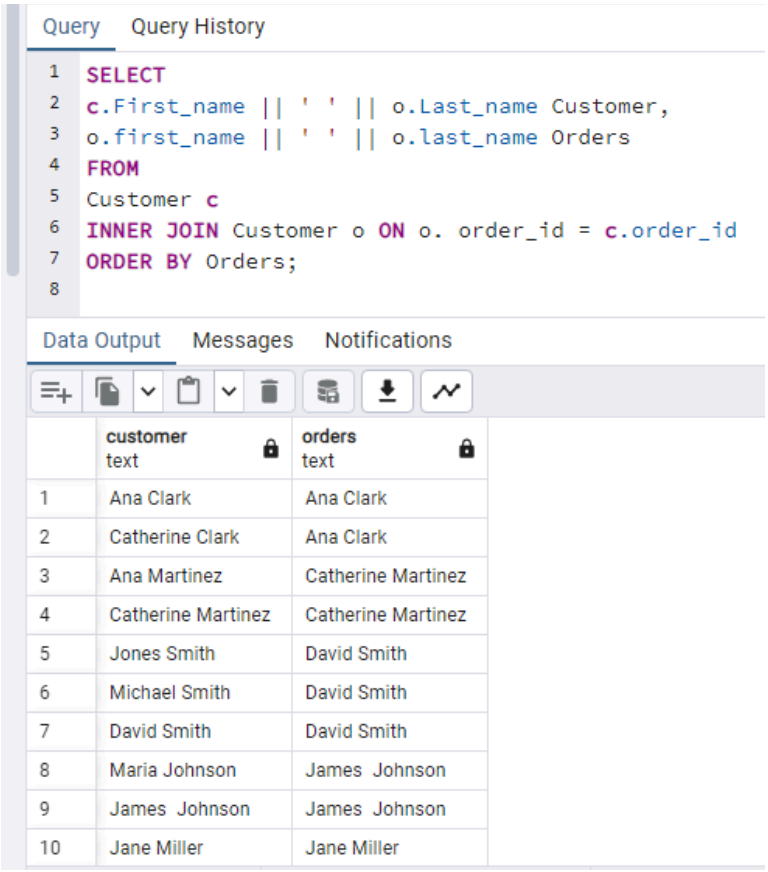


In the below example, we will fetch the data of who purchase with the help of Self Join in the **Customer** table:

```
SELECT

c.First_name || ' ' || o.Last_name Customer,

o.first_name || ' ' || o.last_name Orders

FROM

Customer c

INNER JOIN Customer o ON o. order_id = c.order_id

ORDER BY Orders;
```

Output

After implementing the above command, we will get the following result:



	customer text	orders text
1	Ana Clark	Ana Clark
2	Catherine Clark	Ana Clark
3	Ana Martinez	Catherine Martinez
4	Catherine Martinez	Catherine Martinez
5	Jones Smith	David Smith
6	Michael Smith	David Smith
7	David Smith	David Smith
8	Maria Johnson	James Johnson
9	James Johnson	James Johnson
10	Jane Miller	Jane Miller

As we can observe in the above output table, the ***Customer*** table is executed two times, one as the **Customer**, and another as the **Orders**.

In the above command, we use the **table aliases** such as **c** for the **customer** and **o** for the **Orders**.

And the **join predicate** identifies **Customer/Orders** pair with the help of similar values in the **Customer_id and Orders_id** columns.

Note: The customer (Mia Rodriguez) whose Order_id is Null does not appear on the outcome.

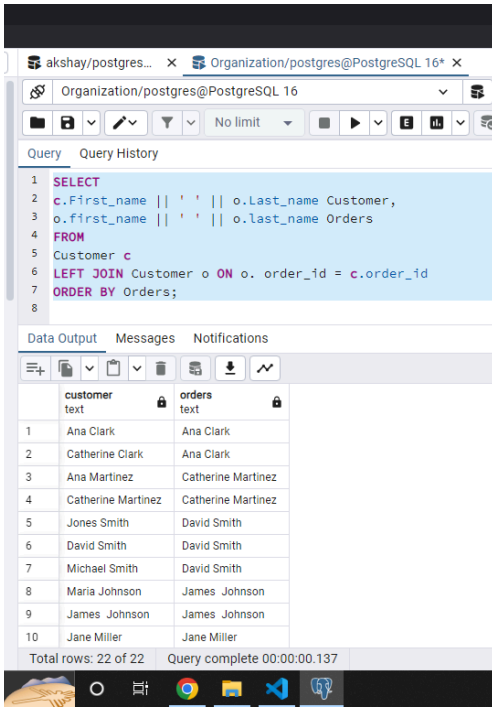
We will use the LEFT JOIN in its place of INNER JOIN clause for including the top **Order** in the Output table using the following command:

```
SELECT
c.First_name || ' ' || o.Last_name Customer,
o.first_name || ' ' || o.last_name Orders
```

```
FROM
Customer c
LEFT JOIN Customer o ON o. order_id = c.order_id
ORDER BY Orders;
```

Output

After executing the above command, we will get the following output:



PostgreSQL Natural Join

In this section, we are going to understand the working of **Natural join**, which is used to join two or more than two tables.

What is the Natural Join clause?

The **natural join** is where multiple tables are combined, and as an output, we will get the new rows, which is intended to join the columns for each of the tables. And it is also used to combine the tables, which creates an implicit join depend on similar column names in the combined tables.

In other words, we can say that the **Natural Join** clause essentially creates a temporary table for a set of rows to work on several (two or more) tables.

And those tables will be specified in the join condition, and at least has one mutual column, and these standard columns should have a relation between them

By default, will use the INNER JOIN operation. And it can be used with LEFT JOIN, INNER JOIN or RIGHT JOIN, but the type of join must be defined in the joining or sql will use the INNER JOIN operation by default.

Syntax of Natural Join

The Natural Join keyword is used with the SELECT command and must be written after the **FROM** Keyword.

```
SELECT [Column_list |*]

FROM Table1

NATURAL [INNER, LEFT, RIGHT] JOIN Table2;
```

Note: In the above syntax, we can also use an asterisk (*) in place of Column-list as the asterisk will generate the output that contain the below fields:

- When both tables have a **unique column**, which contains ***different* column** names.
- When a table has a **common field** and where **both columns** have a same name.

Example of Natural join

Let us see an example to understand how the **PostgreSQL Natural join** works:

To join two tables by using Natural Join

For this, we will create two tables named ***Course_categories and Course*** tables with the help of the **CREATE** command and insert some values using the **INSERT** command.

Firstly, we are going to create ***Course_categories and Course*** tables by using the **CREATE** command:

The following statement is used to create the ***Course_categories*** table:

```
CREATE TABLE Course_categories (
Course_category_id serial PRIMARY KEY,
Course_category VARCHAR NOT NULL,
Course_id INT NOT NULL,
FOREIGN KEY (Course_id) REFERENCES Course(Course_id)
);
```

The below command is used to create the ***Course*** table:

```
CREATE TABLE Course (
Course_id serial PRIMARY KEY,
Course_name VARCHAR NOT NULL
);
```

The ***Course_categories and Course*** tables have been successfully created after executing the above commands.

In the above tables, all the **Course categories** have zero or several **courses**, but here all the courses are linked to unique **Course categories**.

In the ***Course_catagories*** table, the **Cousre_id** column is the **Foreign key**, which is referred to as the **Primary key** of the ***Course*** table.

We will use to perform the **PostgreSQL Natural Join** as the **Course_id** is the standard column in both tables.

Once both the tables have been generated, we are ready to insert some values into it by using the **INSERT** command as follows:

In the below command, we are inserting the values in the ***Course_catagories*** table:

```
INSERT INTO Course_categories (Course_category_id, Course_id)
VALUES
('Adobe Photoshop', 1),
('Adobe Illustrator', 1),
('JavaScript', 2),
('Advance CSS', 2),
('Machine Learning', 2),
('AWS', 3),
('CCNA', 3),
('Kubernetes', 3),
```

('Social Media Marketing', 4),

('Digital Marketing', 4);

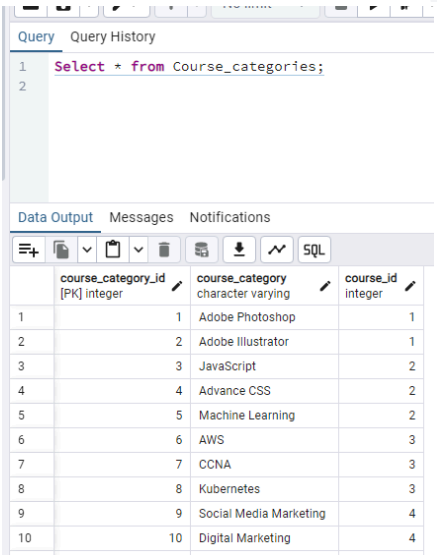
In the below command, we are inserting the values in the ***Course*** table:

```
INSERT INTO Course (Course_name)

VALUES

('Design'),

('Development'),

('IT & Software') ,

('Marketing');
```

After creating and inserting the values in the ***Course_categories and Course*** table, we will use the **SELECT** command to see existing records on the particular tables:

Table1: *Course_categories*

```
Select * from Course_categories;
```



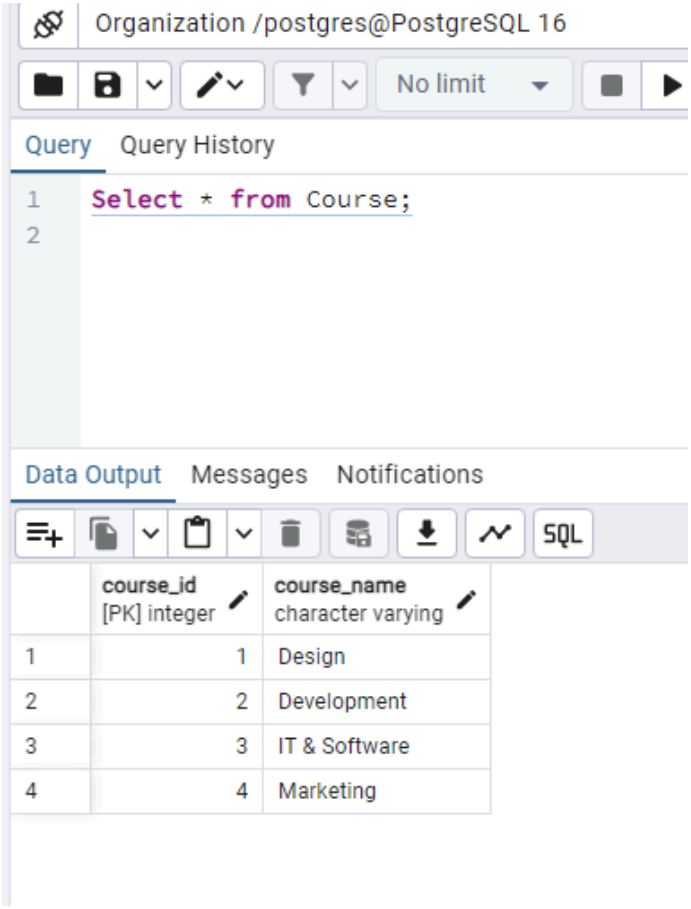
	course_category_id [PK] integer	course_category character varying	course_id integer
1	1	Adobe Photoshop	1
2	2	Adobe Illustrator	1
3	3	JavaScript	2
4	4	Advance CSS	2
5	5	Machine Learning	2
6	6	AWS	3
7	7	CCNA	3
8	8	Kubernetes	3
9	9	Social Media Marketing	4
10	10	Digital Marketing	4

Table2: Course

```
Select * from Course;
```

Output

After executing the above command, we will get the following records from the ***Course*** table:



	course_id [PK] integer	course_name character varying
1	1	Design
2	2	Development
3	3	IT & Software
4	4	Marketing

The below query uses **PostgreSQL Natural Join** clause to combine records from ***Course and Course_categories tables***.

```
SELECT *

FROM Course_categories

NATURAL JOIN Course;
```

Output

Query Query History

```
1 SELECT *
2 FROM Course_categories
3 NATURAL JOIN Course;
4
```

Data Output Messages Notifications

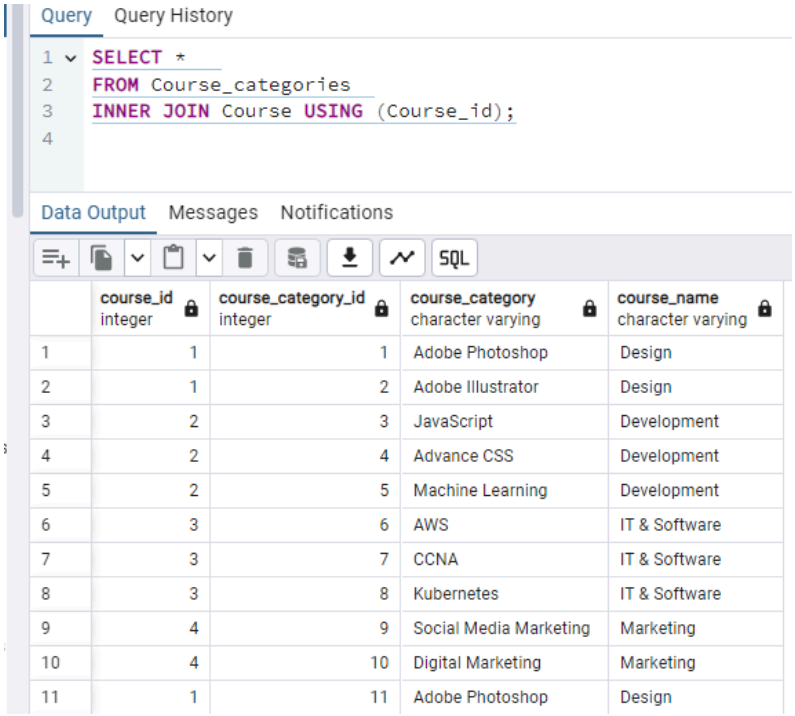
	course_id integer	course_category_id integer	course_category character varying	course_name character varying
1	1	1	Adobe Photoshop	Design
2	1	2	Adobe Illustrator	Design
3	2	3	JavaScript	Development
4	2	4	Advance CSS	Development
5	2	5	Machine Learning	Development
6	3	6	AWS	IT & Software
7	3	7	CCNA	IT & Software
8	3	8	Kubernetes	IT & Software
9	4	9	Social Media Marketing	Marketing
10	4	10	Digital Marketing	Marketing
11	1	11	Adobe Photoshop	Design
12	1	12	Adobe Illustrator	Design

The above command is similar to the below command, where we use the **INNER JOIN** clause instead of the **Natural Join Keyword**.

```
SELECT *

FROM Course_categories

INNER JOIN Course USING (Course_id);
```

Output

After executing the above command, we will get the below command:



	course_id integer	course_category_id integer	course_category character varying	course_name character varying
1	1	1	Adobe Photoshop	Design
2	1	2	Adobe Illustrator	Design
3	2	3	JavaScript	Development
4	2	4	Advance CSS	Development
5	2	5	Machine Learning	Development
6	3	6	AWS	IT & Software
7	3	7	CCNA	IT & Software
8	3	8	Kubernetes	IT & Software
9	4	9	Social Media Marketing	Marketing
10	4	10	Digital Marketing	Marketing
11	1	11	Adobe Photoshop	Design

In **PostgreSQL Natural Join**, it does not seem necessary to describe the Join clause as it uses an **implicit join** condition depends on the Common column.

But we should ignore using the **Natural Join** whenever it possible because sometimes it may cause an unpredicted outcome.

Let us see one example if we have two standard columns in both tables. So, for this we will take ***employee and department*** tables:

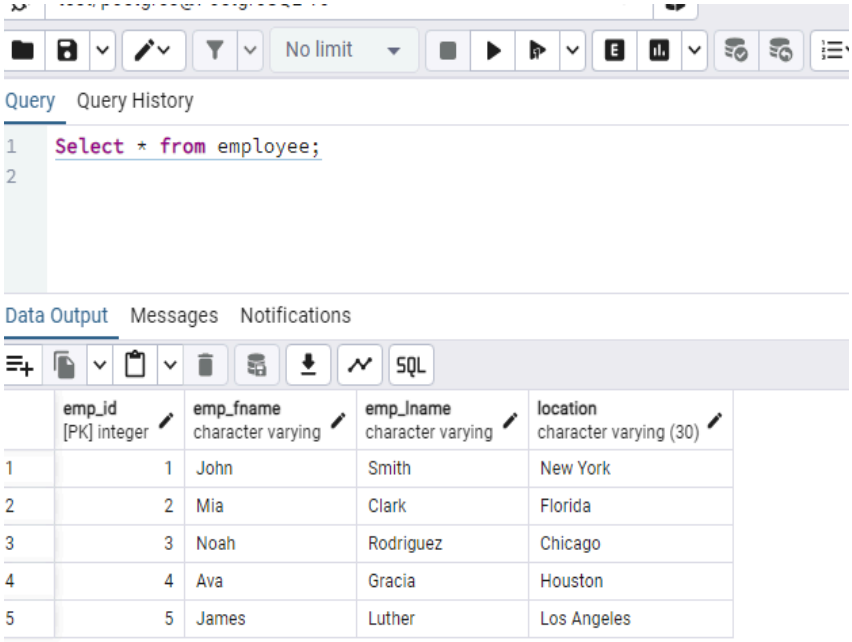
Structure of the employee table

We will get to see the structure of employee table by using the Select command as follows:

```
Select * from employee;
```

Output

After implementing the above statement, we will get the below result:



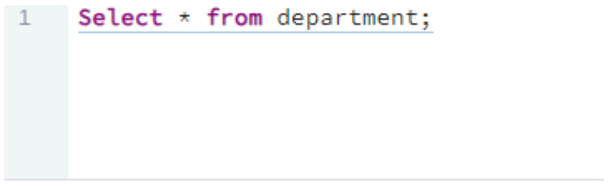
	emp_id [PK] integer	emp_fname character varying	emp_lname character varying	location character varying (30)
1	1	John	Smith	New York
2	2	Mia	Clark	Florida
3	3	Noah	Rodriguez	Chicago
4	4	Ava	Gracia	Houston
5	5	James	Luther	Los Angeles

Structure of the department table

We will get to see the structure of the ***department*** table by using the Select command as follows:

```
Select * from department;
```

Output :



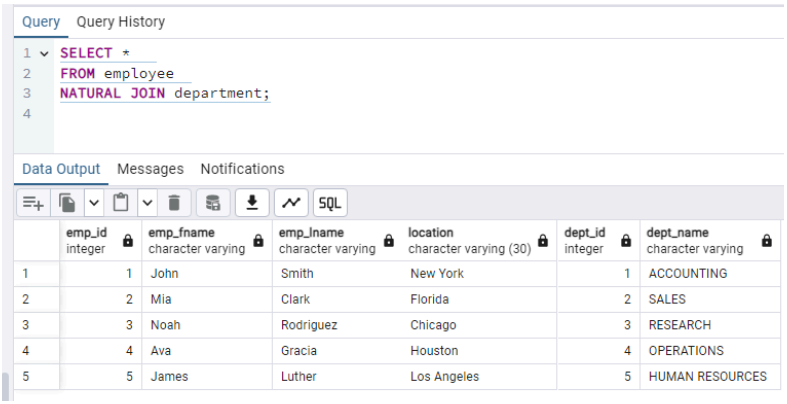
	emp_id [PK] integer	dept_id integer	dept_name character varying
1	1	1	ACCOUNTING
2	2	2	SALES
3	3	3	RESEARCH
4	4	4	OPERATIONS
5	5	5	HUMAN RESOURCES

As we can observe in the above screenshots that the ***employee and department*** tables have the same **emp_id** column, so we can use the **Natural Join clause** to combine these tables in the below command:

```
SELECT *

FROM employee

NATURAL JOIN department;
```

Output

On executing the above command, we will get the following output:



	emp_id integer	emp_fname character varying	emp_lname character varying	location character varying (30)	dept_id integer	dept_name character varying
1	1	John	Smith	New York	1	ACCOUNTING
2	2	Mia	Clark	Florida	2	SALES
3	3	Noah	Rodriguez	Chicago	3	RESEARCH
4	4	Ava	Gracia	Houston	4	OPERATIONS
5	5	James	Luther	Los Angeles	5	HUMAN RESOURCES

In the above output, we will get the **empty table** because both tables also have another common column called **emp_fname**, and which cannot be used for the PostgreSQL **Natural Join**. But the **Natural Join** condition just uses the **emp_fname** column.

Download Complete SQL & PostgreSQL Notes + Practice Files

You can access the full set of **SQL/PostgreSQL notes** along with practice datasets and queries from this GitHub repository:(https://github.com/akshay-dhage/SQL-resources-and-tutorials)

👉 **SQL-resources-and-tutorials by akshay-dhage**