PostgreSQL Cheat Sheet

Section 1 - PostgreSQL Queries Cheat Sheet: Querying Data

Let's discuss different commands or queries for querying data in the PostgreSQL database.

1. Select

This commonly used query allows you to retrieve information from the tables. It's also one of the most complex statements, as it comes with several clauses. Thus, we have divided it into several sections for better understanding.

Syntax:

```
SELECT
   select_list
FROM
   table_name;
```

Where:

Select_list specifies the number of columns from a specific table that you want to retrieve. If you mention multiple columns, you must separate them by commas. Also, you can use (*) for retrieving data from all columns from a particular table.

The **table_name** specifies the name of the table from which you want to retrieve the data.

- **Querying data from one column**

```
SELECT first_name FROM customer;
```

Here, the semicolon specifies the PostgreSQL statement's end.

- **Querying data from multiple columns**

```
SELECT
   first_name,
   last_name,
   email
FROM
   customer;
```

- **Querying data from all columns of a table.**

```
SELECT * FROM customer;
```

We don't recommend using (*), as it impacts database and application performance. So, explicitly specify the names of all columns.

- **Select statement with expression.**

```
SELECT
   first_name || ' ' || last_name,
   email
FROM
   customer;
```

(||) is the concatenation operator.

You can also simply use the SELECT statement without the conjunction of any clauses:

```
SELECT 5 * 3;
```

2. Column Alias

You can use the column alias to assign a temporary name for the column or expression. This alias exists temporarily during the query's execution.

Syntax:

```
SELECT column_name AS alias_name
FROM table_name;
```

You can also omit the AS from the above syntax:

```
SELECT column_name alias_name
FROM table_name;
```

Or

```
SELECT expression AS alias_name
FROM table_name;
```

- **Assigning column alias**

Let's say we need to retrieve all the customers' first and last names from the table. We use the following query:

```
SELECT
   first_name,
   last_name
FROM customer;
```

Now, we'll change the last_name with an alias:

```
SELECT
   first_name,
   last_name AS surname
FROM customer;
```

The column name 'last_name' changes to 'surname'.

You can also use the below query, providing the same result (omitting the as).

```
SELECT
   first_name,
   last_name surname
FROM customer;
```

- **Assigning column alias to an expression.**

This will retrieve all the customers' first and last names with a space in between them.

```
SELECT
   first_name || ' ' || last_name
FROM
   customer;
```

The above query will display a table without any specific column name.

Now, let us assign an alias for the concatenated column as "full_name:"

```
SELECT
    first_name || ' ' || last_name AS full_name
FROM
    customer;
```

You'll get a table with the column name "full_name."

- **Assign column alias with spaces**

column_name AS "column alias"

For example:

```
SELECT
    first_name || ' ' || last_name "full name"
FROM
    customer;
```

The above query will display a table with column name 'full_name'.

3. Order By

“Select” query results aren’t organized. So, you can organize it in ascending or descending order using the “order by” clause.

Syntax:

```
SELECT
      select_list
FROM
      table_name
ORDER BY
      sort_expression1 [ASC | DESC],
        ...
      sort_expressionN [ASC | DESC];
```

In the above syntax, the sort expression can be a column or an expression that you want to sort after the ORDER BY keywords. For sorting data based on multiple columns or expressions, you must place a comma (,) between two columns or expressions to separate them. Then, you must specify ascending or descending.

Note: The ‘order by’ clause must be be the last in any select query.

- **Sorting rows by one column.**

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

This query will display a table with a single column having the first names of all customers in ascending order.

Also, ascending is the default if you do not specify the (asc or desc) with the order by clause

Therefore, the following query will display the same output:

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

- **Sorting rows by multiple columns.**

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

The above query will display a table with two columns, first_name and last_name. The first_name column will have the customers' first names in the ascending order, while the last_name column has the last names of customers in the descending order.

- **Sorting rows by expression.**

```
SELECT
      first_name,
      LENGTH(first_name) len
FROM
      customer
ORDER BY
      len DESC;
```

This query will display a table with two columns, first_name and their lengths. It sorts the rows in descending order based on the first name's length.

- **Sorting rows with null values.**

Null represents the missing or unknown data. You can even sort the rows with null values.

```
ORDER BY sort_expresssion [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

-- create a new table

```
CREATE TABLE sort_demo(
      num INT
);
```

-- insert some data

```
INSERT INTO sort_demo(num)
VALUES(1),(2),(3),(null);
```

- **Sorting rows with null values.**

```
SELECT num
FROM sort_demo
ORDER BY num;
```

	num integer
1	1
2	2
3	3
4	[null]

As you can see, Null values will be last by default.

The following query will provide the same result.

```
SELECT num
FROM sort_demo
ORDER BY num NULLS LAST;
```

But to get the null valued rows in the first place, you must mention it explicitly:

```
SELECT num
FROM sort_demo
ORDER BY num NULLS FIRST;
```

You will get the following output:

	num integer
1	[null]
2	1
3	2
4	3

4. Select Distinct

This removes duplicate rows from a result set. You can apply the DISTINCT clause to one or more columns in the select list of the SELECT statement.

Syntax:

```
SELECT
   DISTINCT column1
FROM
   table_name;
```

Or:

```
SELECT
   DISTINCT column1, column2
FROM
   table_name;
```

Or:

```
SELECT
   DISTINCT ON (column1) column_alias,
   column2
FROM
   table_name
ORDER BY
   column1,
   column2;
```

For example, we will create a table distinct_demo using the following query and insert some data.

```
CREATE TABLE distinct_demo (
      id serial NOT NULL PRIMARY KEY,
      bcolor VARCHAR,
      fcolor VARCHAR
);
```

Insert rows using the following query.

```
INSERT INTO distinct_demo (bcolor, fcolor)
VALUES
      ('red', 'red'),
      ('red', 'red'),
      ('red', NULL),
      (NULL, 'red'),
      ('red', 'green'),
      ('red', 'blue'),
      ('green', 'red'),
      ('green', 'blue'),
      ('green', 'green'),
      ('blue', 'red'),
      ('blue', 'green'),
      ('blue', 'blue');
```

- **Without using distinct clause**

```
SELECT
      id,
      bcolor,
      fcolor
FROM
      distinct_demo ;
```

Output:

	id integer	**bcolor** character varying	**fcolor** character varying
1	1	red	red

2	2	red	red
3	3	red	[null]
4	4	[null]	red
5	5	red	green
6	6	red	blue
7	7	green	red
8	8	green	blue
9	9	green	green
10	10	blue	red
11	11	blue	red
12	12	blue	blue

- **Distinct clause on one column.**

```
SELECT
      DISTINCT bcolor
FROM
      distinct_demo
ORDER BY
      bcolor;
```

Output:

	bcolor character varying
1	blue
2	green
3	red
4	[null]

- **Distinct on multiple columns.**

```
SELECT
```

```
      DISTINCT bcolor,
      fcolor
FROM
      distinct_demo
ORDER BY
      bcolor,
      fcolor;
```

Output:

	bcolor character varying	**fcolor** character varying
1	blue	blue
2	blue	green
3	blue	red
4	green	blue
5	green	green
6	green	red
7	red	blue
8	red	green
9	red	red
10	red	[null]
11	[null]	red

- **Distinct on example.**

```
SELECT
      DISTINCT ON (bcolor) bcolor,
      fcolor
FROM
      distinct_demo
ORDER BY
      bcolor,
      fcolor;
```

Output:

	bcolor character varying	**fcolor** character varying
1	blue	blue
2	green	blue
3	red	blue
4	[null]	red

Section 2 - Filtering Data

Now, let’s see the different classes we can use with the SELECT statement to filter and retrieve data from database tables.

1. Where Clause

It is used to retrieve rows based on a specific condition mentioned.

Syntax:

```
SELECT select_list
FROM table_name
WHERE condition
ORDER BY sort_expression
```

The condition must evaluate as true, false, or unknown. It can be a boolean expression or a combination of boolean expressions using the AND and OR operators. Only the rows satisfying the condition will be returned.

In the where clause, you can use logical and comparison operators:

Operator	**Description**
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal

<=	Less than or equal
<> or !=	Not equal
AND	Logical operator AND
OR	Logical operator OR
IN	Return true if a value matches any value in a list
BETWEEN	Return true if a value is between a range of values
LIKE	Return true if a value matches a pattern
IS NULL	Return true if a value is NULL
NOT	Negate the result of other operators

- **Using (=) operator.**

```
SELECT
        last_name,
        first_name
FROM
        customer
WHERE
        first_name = 'Jamie';
```

The above query returns a table with two columns, last_name and first_name, where the values on the first_name column only consist of Jamie.

- **Using AND operator.**

```
SELECT
      last_name,
      first_name
FROM
      customer
WHERE
      first_name = 'Jamie' AND
        last_name = 'Rice';
```

The above returns the table containing the values in the last_name column as Rice and in the first_name column as Jamie.

- **Using OR operator.**

```
SELECT
```

```
      first_name,
      last_name
FROM
      customer
WHERE
      last_name = 'Rodriguez' OR
      first_name = 'Adam';
```

This query returns a table containing values whose last_name is Rodriguez, or the first_name is Adam.

- **Using IN operator.**

```
SELECT
      first_name,
      last_name
FROM
      customer
WHERE
      first_name IN ('Ann','Anne','Annie');
```

It returns a table containing the values whose first name is Ann, Anne, and Annie.

- **Using LIKE operator.**

```
SELECT
      first_name,
      last_name
FROM
      customer
WHERE
      first_name LIKE 'Ann%'
```

This query returns a table whose first_name column values start with ‘Ann’.

- **Using BETWEEN operator.**

```
SELECT
      first_name,
      LENGTH(first_name) name_length
FROM
      customer
```

```
WHERE
      first_name LIKE 'A%' AND
      LENGTH(first_name) BETWEEN 3 AND 5
ORDER BY
      name_length;
```

You will get a table containing all the first names, whose length varies between 3 and 5.

- **Using not equal (<>) operator.**

```
SELECT
      first_name,
      last_name
FROM
      customer
WHERE
      first_name LIKE 'Bra%' AND
      last_name <> 'Motley';
```

The above query returns a table containing the values that start with ‘Bra’ in the first_name column and all other values except ‘Motley’ in the last_name column.

2. LIMIT

This clause will constrain the number of rows returned by the query.

Syntax:

```
SELECT select_list
FROM table_name
ORDER BY sort_expression
LIMIT row_count
```

The statement returns row_count rows generated by the query. If row_count is zero, the query returns an empty set. In case row_count is NULL, the query returns the same result set as it does not have the LIMIT clause.

Use the OFFSET clause to skip several rows before returning the row_count rows.

```
SELECT select_list
FROM table_name
LIMIT row_count OFFSET row_to_skip;
```

- **Using LIMIT for limiting the number of rows.**

```
SELECT
      film_id,
      title,
      release_year
FROM
      film
ORDER BY
      film_id
LIMIT 5;
```

This query returns a table of five rows containing film_id, title, and release_year, where the film_id is ordered in ascending order.

- **Using OFFSET.**

```
SELECT
      film_id,
      title,
      release_year
FROM
      film
ORDER BY
      film_id
LIMIT 4 OFFSET 3;
```

This query returns a table of four rows containing film_id, title, and release_year, where the film_id is ordered in ascending order. It starts the values from film_id = 4 and not from the beginning.

- **Using OFFSET for getting top/bottom rows.**

```
SELECT
      film_id,
      title,
      rental_rate
FROM
      film
ORDER BY
      rental_rate DESC
LIMIT 10;
```

The table with film_id, title, and rental_rate columns contains ten rows, where rental_rate is in the descending order.

3. FETCH

PostgreSQL supports the FETCH clause to retrieve several rows returned by a query.

Syntax:

```
OFFSET start { ROW | ROWS }
FETCH { FIRST | NEXT } [ row_count ] { ROW | ROWS } ONLY
```

The FETCH clause is functionally equivalent to the LIMIT clause. If you plan to make your application compatible with other database systems, you should use the FETCH clause because it follows the standard SQL.

4. IN

Used with the “where” clause, this clause will check whether a value matches any value in a list of values.

```
value IN (value1,value2,...)
```

You can also use the select query in place of the values:

```
value IN (SELECT column_name FROM table_name);
```

For example:

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

We have 1 and 2 customer IDs, and the above query will return a table containing data of customers with customer ID 1 or 2.

- **Using NOT IN**

```
SELECT
```

```
      customer_id,
      rental_id,
      return_date
FROM
      rental
WHERE
      customer_id NOT IN (1, 2);
```

The above query will return a table containing data of all the customers, except the data of customers with customer ID 1 or 2

5. BETWEEN

This clause will match a value against a range of values.

Syntax:

```
value BETWEEN low AND high;
```

value >= low and value <= high

```
value NOT BETWEEN low AND high;
```

value < low OR value > high

- **Using BETWEEN operators**

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

The above query returns the table with customer_id, payment_id, and amount, where the amount ranges between 8 and 9.

- **Using NOT BETWEEN**

```
SELECT
```

```
      customer_id,
      payment_id,
      amount
FROM
      payment
WHERE
      amount NOT BETWEEN 8 AND 9;
```

The above query returns the table with customer_id, payment_id, and amount, except for the amount ranging between 8 and 9.

- **Using BETWEEN**

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

The above query returns the table containing customer_id, payment_id, amount, and payment_date, where the date ranges between 2007-02-07 and 2007-02-15.

6. LIKE

This operator will match the first name of the customer with a string like this query:

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

You will get a table with first_name and last_name columns, where the values in the first_name column start with ‘Jen’.

You can match a pattern by combining literal values with wildcard characters and using the LIKE or NOT LIKE operator to find the matches. PostgreSQL provides you with two wildcards:

- Percent sign (%) matches any sequence of zero or more characters.

- Underscore sign (_) matches any single character.

For example:

```
SELECT
	'foo' LIKE 'foo', -- true
	'foo' LIKE 'f%', -- true
	'foo' LIKE '_o_', -- true
	'bar' LIKE 'b_'; -- false
```

- **Using (%) and (_)**

```
SELECT
      first_name,
      last_name
FROM
      customer
WHERE
      first_name LIKE '_her%'
ORDER BY
        First_name;
```

This returns a table with first_name and last_name columns, where the values in the first_name column start with any letter followed by 'her'.

- **Using NOT LIKE**

```
SELECT
      first_name,
      last_name
FROM
      customer
WHERE
      first_name NOT LIKE 'Jen%'
ORDER BY
        first_name
```

You will get a table with first_name and last_name columns, where the first_name column contains values that do not start with 'Jen.'

- **Using ILIKE (checks case sensitivity)**

```
SELECT
      first_name,
      last_name
FROM
      customer
WHERE
      first_name ILIKE 'BAR%';
```

This query returns a table with first_name and last_name columns, where the values in first_name start with BAR.

7. IS NULL

The IS NULL condition is used to test for the NULL values in SELECT, INSERT, UPDATE, and DELETE statements.

Syntax:

```
expression IS NULL;
```

If the expression is NULL, the condition evaluates to true. Otherwise, the condition evaluates as false.

- **IS NULL with SELECT.**

```
SELECT *
FROM employees
WHERE first_number is NULL;
```

The above query returns a table containing records from the employee table whose fisst_number is NULL.

- **IS NULL with INSERT**

```
INSERT INTO contacts
(first_name, last_name)
SELECT first_name, last_name
FROM employees
WHERE employee_number IS NULL;
```

The above query inserts new data into the contacts table whose employee number has a NULL value.

- **IS NULL with UPDATE**

```
UPDATE employees
SET status = 'Not Active'
WHERE last_name IS NULL;
```

The above query updates the records in the employees table whose last name holds a NULL value.

- **IS NULL with DELETE**

```
DELETE FROM employees
WHERE employee_number IS NULL;
```

The above query will delete all the records employees table whose employee number is NULL.

Section 3 - Joining Multiple Tables

Let’s discuss ‘JOIN’ in PostgreSQL:

JOINS

You can combine columns from one (self-join) or more tables based on the common column values between the related tables. The common columns are typically the first table’s primary key columns and the second table’s foreign key columns.

To explain the concept, we’ll use two tables where we perform different types of joins.

The following query creates the table ‘basket_a:’

```
CREATE TABLE basket_a (
    a INT PRIMARY KEY,
    fruit_a VARCHAR (100) NOT NULL
);
```

The following query creates the table ‘basket_b:’

```
CREATE TABLE basket_b (
    b INT PRIMARY KEY,
    fruit_b VARCHAR (100) NOT NULL
);
```

Use the following query to insert data into ‘basket_a:’

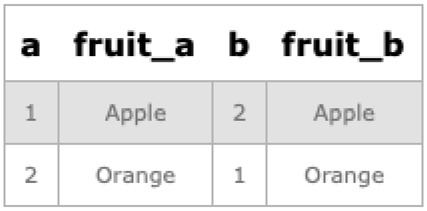
```
INSERT INTO basket_a (a, fruit_a)
VALUES
    (1, 'Apple'),
    (2, 'Orange'),
    (3, 'Banana'),
    (4, 'Cucumber');
```

The following query inserts data into ‘basket_b:’

```
INSERT INTO basket_b (b, fruit_b)
VALUES
    (1, 'Orange'),
    (2, 'Apple'),
    (3, 'Watermelon'),
    (4, 'Pear');
```

- **Inner join**

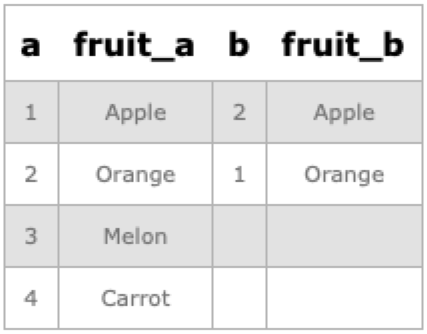
```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
INNER JOIN basket_b
    ON fruit_a = fruit_b;
```



a	fruit_a	b	fruit_b
1	Apple	2	Apple
2	Orange	1	Orange

- **Left join**

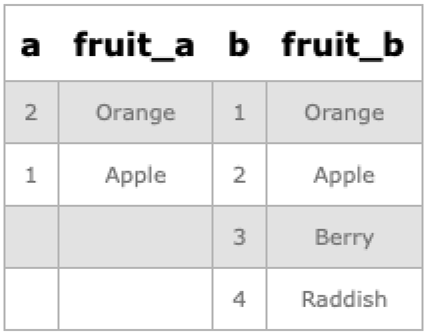
```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    table_a
LEFT JOIN table_b
   ON fruit_a = fruit_b;
```



a	fruit_a	b	fruit_b
1	Apple	2	Apple
2	Orange	1	Orange
3	Melon		
4	Carrot		

- **Right join**

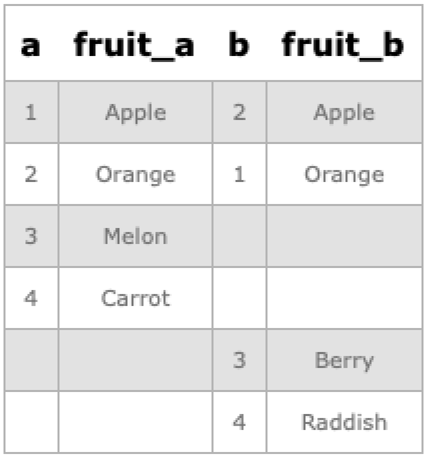
```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    table_a
RIGHT JOIN table_b ON fruit_a = fruit_b;
```



a	fruit_a	b	fruit_b
2	Orange	1	Orange
1	Apple	2	Apple
		3	Berry
		4	Raddish

- **Full outer join**

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    table_a
FULL OUTER JOIN table_b
    ON fruit_a = fruit_b;
```



a	fruit_a	b	fruit_b
1	Apple	2	Apple
2	Orange	1	Orange
3	Melon		
4	Carrot		
		3	Berry
		4	Raddish

Table Alias

These temporarily assign tables new names during the execution of a query.

```
table_name AS alias_name;
```

Self Join

A self-join is a regular join that joins a table to itself. To form a self-join, you specify the same table twice with different table aliases and provide the join predicate after the ON keyword.

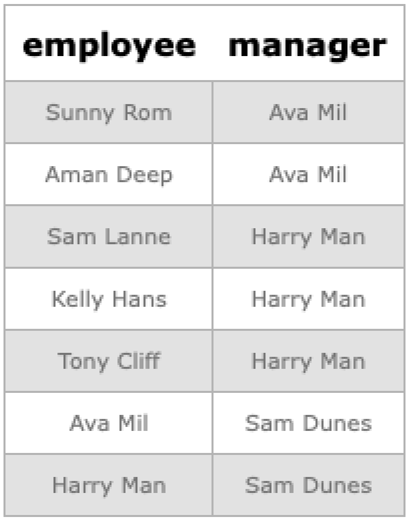
For example, here’s an employee table with some inserted data:

```
CREATE TABLE employee_data (
      emp_id INT PRIMARY KEY,
      f_name VARCHAR (255) NOT NULL,
      l_name VARCHAR (255) NOT NULL,
      manager_id INT,
      FOREIGN KEY (manager_id)
      REFERENCES employee (emp_id)
      ON DELETE CASCADE
);
```

```
INSERT INTO employee (
      emp_id,
      f_name,
      l_name,
      manager_id
)
VALUES
      (1, 'Sam', 'Dunes', NULL),
      (2, 'Ava', 'Mil', 1),
      (3, 'Harry', 'Man, 1),
      (4, 'Aman', 'Deep', 2),
      (5, 'Sunny', 'Rom', 2),
      (6, 'Kelly', 'Hans', 3),
      (7, 'Tony', 'Cliff, 3),
      (8, 'Sam', 'Lanne', 3);
```

- **Using Self Join**

```
SELECT
    e.f_name || ' ' || e.l_name employee,
    m .f_name || ' ' || m .l_name manager
FROM
    employee_data e
INNER JOIN employee_data m ON m .emp_id = e.manager_id
ORDER BY manager;
```



employee	manager
Sunny Rom	Ava Mil
Aman Deep	Ava Mil
Sam Lanne	Harry Man
Kelly Hans	Harry Man
Tony Cliff	Harry Man
Ava Mil	Sam Dunes
Harry Man	Sam Dunes

Cross Join

This join allows you to produce a Cartesian Product of rows in two or more tables. Unlike LEFT JOIN or INNER JOIN, the CROSS JOIN clause does not have a join predicate:

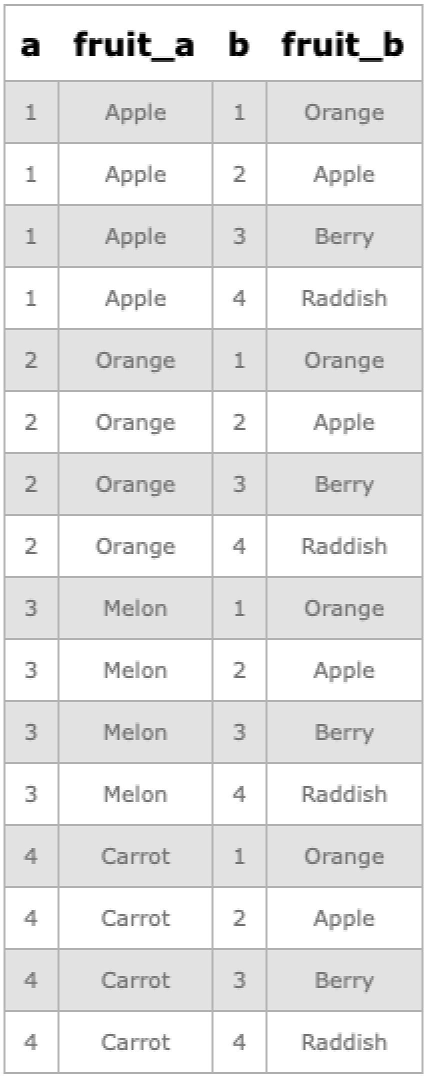
```
CREATE TABLE table_a (
    a INT PRIMARY KEY,
    fruit_a VARCHAR (100) NOT NULL
);
```

```
CREATE TABLE table_b (
    b INT PRIMARY KEY,
    fruit_b VARCHAR (100) NOT NULL
);
```

```
INSERT INTO table_a (a, fruit_a)
VALUES
    (1, 'Apple'),
    (2, 'Orange'),
    (3, 'Melon'),
    (4, 'Carrot');
```

```
INSERT INTO table_b (b, fruit_b)
VALUES
    (1, 'Orange'),
    (2, 'Apple'),
    (3, 'Berry'),
    (4, 'Raddish');
```

```
SELECT * FROM table_a cross join table_b;
```



a	fruit_a	b	fruit_b
1	Apple	1	Orange
1	Apple	2	Apple
1	Apple	3	Berry
1	Apple	4	Raddish
2	Orange	1	Orange
2	Orange	2	Apple
2	Orange	3	Berry
2	Orange	4	Raddish
3	Melon	1	Orange
3	Melon	2	Apple
3	Melon	3	Berry
3	Melon	4	Raddish
4	Carrot	1	Orange
4	Carrot	2	Apple
4	Carrot	3	Berry
4	Carrot	4	Raddish

Natural Join

This creates an implicit join based on the same column names in the joined tables.

Syntax:

```
SELECT select_list
FROM T1
NATURAL [INNER, LEFT, RIGHT] JOIN T2;
```

For example, we have created a table and inserted some rows:

```
DROP TABLE IF EXISTS categories;
```

```sql
CREATE TABLE categories (
      cat_id serial PRIMARY KEY,
      cat_name VARCHAR (255) NOT NULL
);
```

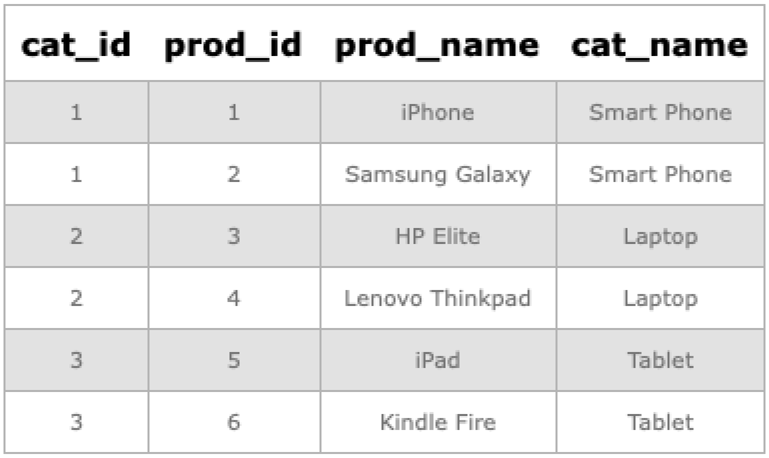
```sql
DROP TABLE IF EXISTS products;
CREATE TABLE prod (
      prod_id serial PRIMARY KEY,
      prod_name VARCHAR (255) NOT NULL,
      cat_id INT NOT NULL,
      FOREIGN KEY (cat_id) REFERENCES categories (cat_id)
);
```

```sql
INSERT INTO categories (cat_name)
VALUES
      ('Smart Phone'),
      ('Laptop'),
      ('Tablet');
```

```sql
INSERT INTO prod (prod_name, cat_id)
VALUES
      ('iPhone', 1),
      ('Samsung Galaxy', 1),
      ('HP Elite', 2),
      ('Lenovo Thinkpad', 2),
      ('iPad', 3),
      ('Kindle Fire', 3);
```

- **Using natural join**

```sql
SELECT * FROM prod
NATURAL JOIN categories;
```



cat_id	prod_id	prod_name	cat_name
1	1	iPhone	Smart Phone
1	2	Samsung Galaxy	Smart Phone
2	3	HP Elite	Laptop
2	4	Lenovo Thinkpad	Laptop
3	5	iPad	Tablet
3	6	Kindle Fire	Tablet

Section 4 - Grouping Data

Group by Clause

This divides the rows returned from the SELECT statement into groups. For each group, you can apply an aggregate function.

Syntax:

```
SELECT
   column_1,
   column_2,
   ...,
   aggregate_function(column_3)
FROM
   table_name
GROUP BY
   column_1,
   column_2,
   ...;
```

PostgreSQL evaluates the GROUP BY clause after the FROM and WHERE clauses and before the HAVING SELECT, DISTINCT, ORDER BY, and LIMIT clauses.

```
DROP TABLE IF EXISTS employee_data;
```

```
CREATE TABLE employee_data (
      emp_id INT PRIMARY KEY,
      f_name VARCHAR (255) NOT NULL,
      l_name VARCHAR (255) NOT NULL,
      manager_id INT,
      Salary int,
      FOREIGN KEY (manager_id)
      REFERENCES employee_data (emp_id)
      ON DELETE CASCADE
);
```

```
INSERT INTO employee_data (
      emp_id,
      f_name,
      L_name,
      salary,
      manager_id
)
VALUES
      (1, 'Sam', 'Dunes', 1000,NULL),
      (2, 'Ava', 'Mil', 2000, 1),
      (3, 'Harry', 'Man', 2400, 1),
      (4, 'Aman', 'Deep', 4500, 2),
      (5, 'Sunny', 'Rom', 3455, 2),
      (6, 'Kelly', 'Hans', 6733,  3),
      (7, 'Tony', 'Cliff', 4577, 3),
      (8, 'Sam', 'Lanne', 4533, 3);
```

- **Using group by without aggregate function**

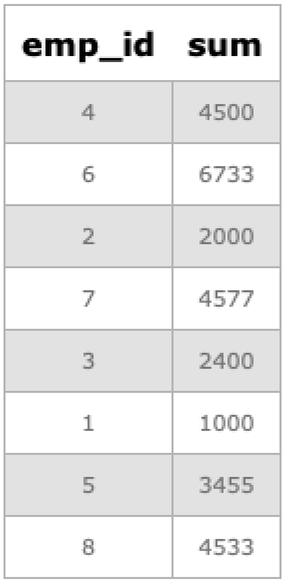
```
SELECT
   emp_id
FROM
   employee_data
GROUP BY
   Manager_id;
```

You will get an error:

column "employee_data.emp_id" must appear in the GROUP BY clause or be used in an aggregate function.

- **Group by with sum() function**

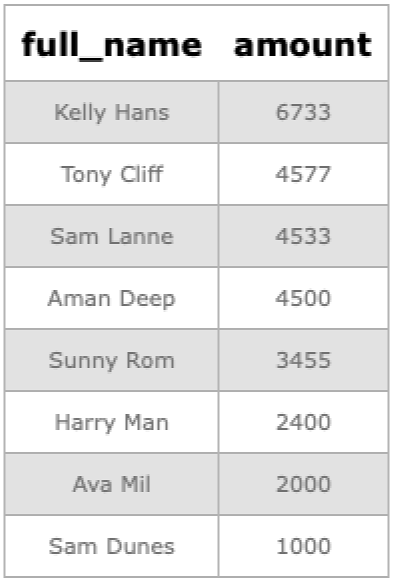
```
SELECT
      emp_id,
      SUM (salary)
FROM
      employee_data
GROUP BY
      emp_id;
```



emp_id	sum
4	4500
6	6733
2	2000
7	4577
3	2400
1	1000
5	3455
8	4533

- **Using group by with join clause**

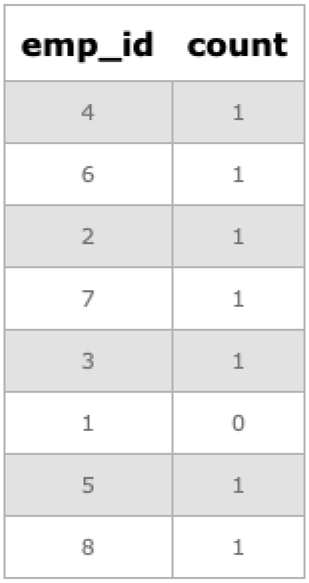
```
SELECT
      f_name || ' ' || l_name full_name,
      SUM (salary) amount
FROM
      employee_data
GROUP BY
      full_name, employee_data.salary
ORDER BY salary DESC;
```



full_name	amount
Kelly Hans	6733
Tony Cliff	4577
Sam Lanne	4533
Aman Deep	4500
Sunny Rom	3455
Harry Man	2400
Ava Mil	2000
Sam Dunes	1000

- **Group by with count() function**

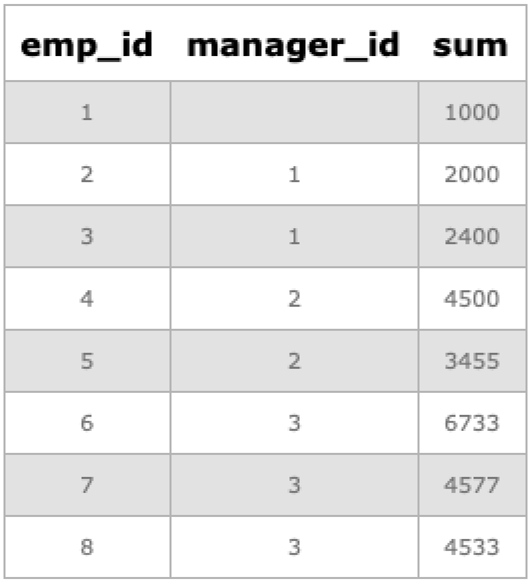
```
SELECT
      emp_id,
      COUNT (manager_id)
FROM
      employee_data
GROUP BY
      manager_id;
```



emp_id	count
4	1
6	1
2	1
7	1
3	1
1	0
5	1
8	1

- **Group by with multiple columns**

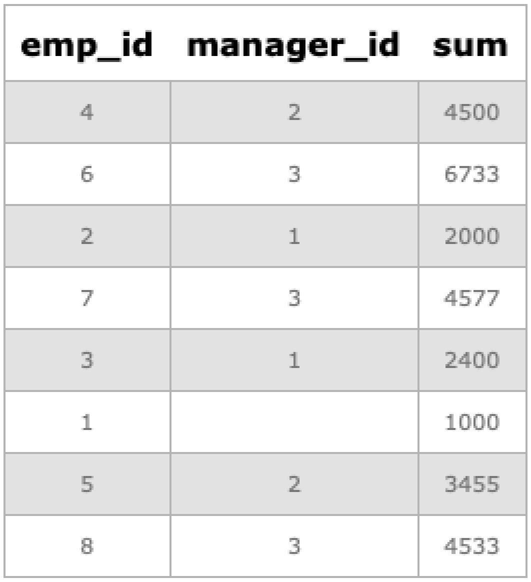
```
SELECT
      emp_id,
      manager_id,
      SUM(salary)
FROM
      employee_data
GROUP BY
      manager_id,
      emp_id
ORDER BY
    emp_id;
```



emp_id	manager_id	sum
1		1000
2	1	2000
3	1	2400
4	2	4500
5	2	3455
6	3	6733
7	3	4577
8	3	4533

- **Group by with date column**

```
SELECT
            emp_id, manager_id, SUM(salary) sum
FROM
      employee_data
GROUP BY
      emp_id, manager_id,salary ;
```



emp_id	manager_id	sum
4	2	4500
6	3	6733
2	1	2000
7	3	4577
3	1	2400
1		1000
5	2	3455
8	3	4533

Having Clause

This specifies a search condition for a group or an aggregate. The HAVING clause is often used with the GROUP BY clause to filter groups or aggregates based on a specified condition.

Syntax:

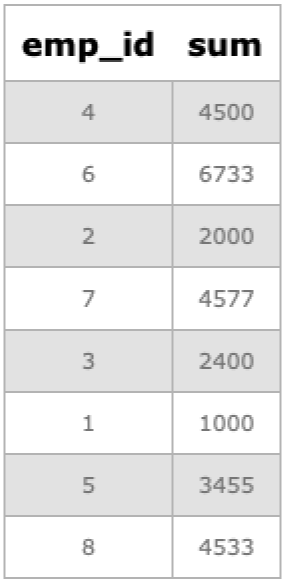
```
SELECT
      column1,
      aggregate_function (column2)
FROM
      table_name
GROUP BY
      column1
HAVING
      condition;
```

PostgreSQL evaluates the HAVING clause after the FROM, WHERE, GROUP BY, and before the SELECT, DISTINCT, ORDER BY, and LIMIT clauses.

- **Using having with a sum function**

```
SELECT
      EMP_id,
      SUM (SALARY)
FROM
      EMPLOYEE_DATA
```

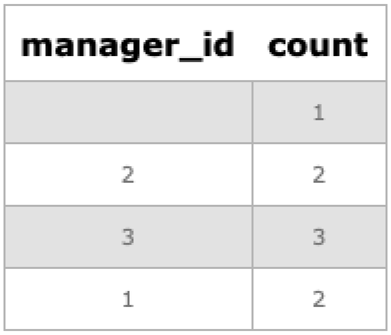
```
GROUP BY
      emp_id;
```



emp_id	sum
4	4500
6	6733
2	2000
7	4577
3	2400
1	1000
5	3455
8	4533

- **Using the having clause with the count function**

```
SELECT
      manager_id,
      COUNT (emp_id)
FROM
      employee_data
GROUP BY
      manager_id;
```



manager_id	count
	1
2	2
3	3
1	2

Section 5 - Set Operations

This section will walk you through different set operations supported by PostgreSQL.

UNION

It combines result sets of two or more SELECT statements into a single result set.

Syntax:

```
SELECT select_list_1
FROM table_expresssion_1
UNION
SELECT select_list_2
FROM table_expression_2
```

For example, we have created two tables and inserted data in them to perform union.

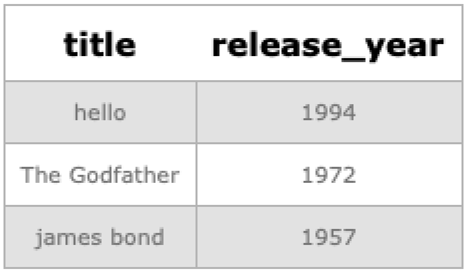
```
DROP TABLE IF EXISTS top_films;
CREATE TABLE top_films(
      title VARCHAR NOT NULL,
      release_year SMALLINT
);
```

```
DROP TABLE IF EXISTS popular_films;
CREATE TABLE popular_films(
      title VARCHAR NOT NULL,
      release_year SMALLINT
);
```

```
INSERT INTO
   top_films(title,release_year)
VALUES
   ('hello',1994),
   ('The Godfather',1972),
   ('james bond',1957);
```

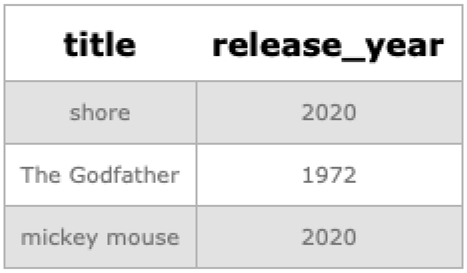
```
INSERT INTO
   popular_films(title,release_year)
VALUES
   ('shore',2020),
   ('The Godfather',1972),
   ('mickey mouse,2020);
```

```
SELECT * FROM top_films;
```



title	release_year
hello	1994
The Godfather	1972
james bond	1957

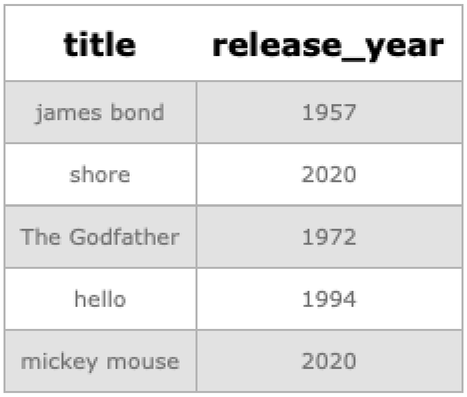
```
SELECT * FROM popular_films;
```



title	release_year
shore	2020
The Godfather	1972
mickey mouse	2020

- **Union example**

```
SELECT * FROM top_films
UNION
SELECT * FROM popular_films;
```

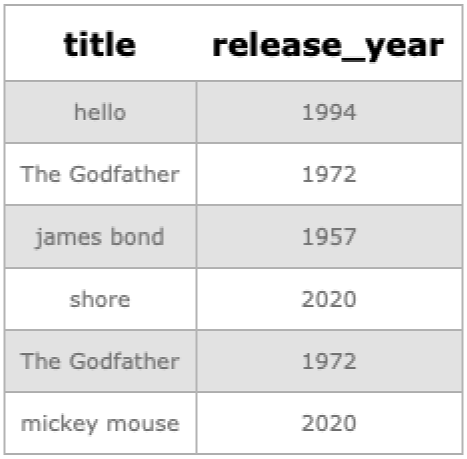


title	release_year
james bond	1957
shore	2020
The Godfather	1972
hello	1994
mickey mouse	2020

- **Union all example**

```
SELECT * FROM top_films
```

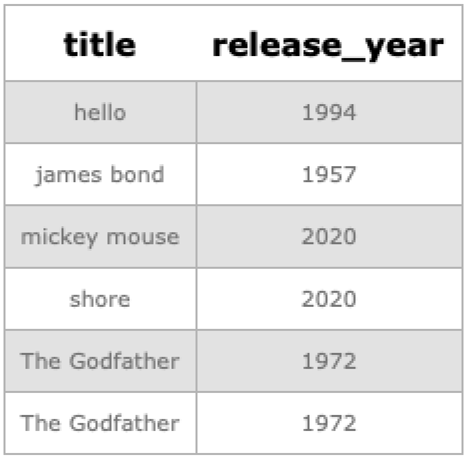
```
UNION ALL
SELECT * FROM popular_films;
```



title	release_year
hello	1994
The Godfather	1972
james bond	1957
shore	2020
The Godfather	1972
mickey mouse	2020

- **Union all with order by**

```
SELECT * FROM top_films
UNION ALL
SELECT * FROM popular_films
ORDER BY title;
```



title	release_year
hello	1994
james bond	1957
mickey mouse	2020
shore	2020
The Godfather	1972
The Godfather	1972

INTERSECT

This operator combines result sets of two or more SELECT statements into a single result set.

Syntax:

```
SELECT select_list
FROM A
INTERSECT
SELECT select_list
FROM B;
```

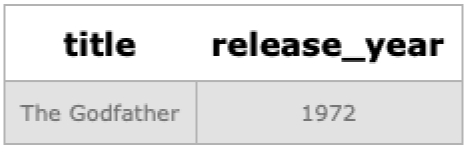
Make sure the number of columns is the same and have compatible data types to be merged.

- **Intersect with an order by clause**

```
SELECT select_list
FROM A
INTERSECT
SELECT select_list
FROM B
ORDER BY sort_expression;
```

For example, we will intersect the two tables: popular_films and top_films:

```
SELECT *
FROM popular_films
INTERSECT
SELECT *
FROM top_films;
```



title	release_year
The Godfather	1972

Section 6 - Grouping Sets, Cube, and Rollup

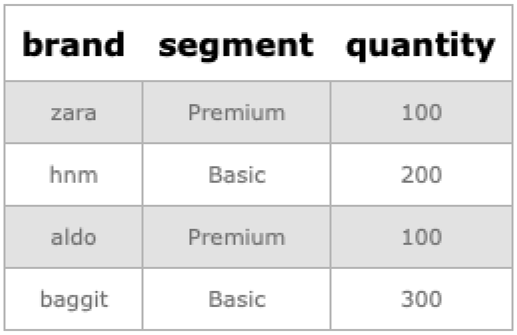
Grouping Sets

This can generate multiple grouping sets in a query. To explain, we will create the sales table:

```
DROP TABLE IF EXISTS sales;
CREATE TABLE sales (
    brand VARCHAR NOT NULL,
    segment VARCHAR NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (brand, segment)
);
```

```
INSERT INTO sales (brand, segment, quantity)
VALUES
    ('zara', 'Premium', 100),
    ('hnm', 'Basic', 200),
    ('aldo', 'Premium', 100),
    ('baggit', 'Basic', 300);
```

```
SELECT * FROM sales;
```



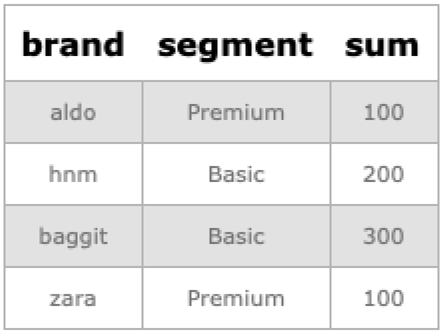
brand	segment	quantity
zara	Premium	100
hnm	Basic	200
aldo	Premium	100
baggit	Basic	300

A grouping set is a set of columns that you group using the GROUP BY clause. The grouping sets' syntax consists of multiple columns enclosed in parentheses, separated by commas.

Syntax:

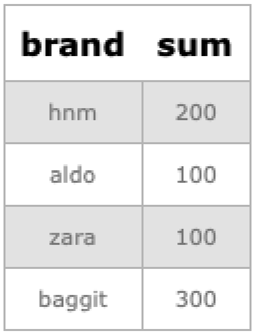
```
(column1, column2, ...)
```

```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    brand,
    segment;
```



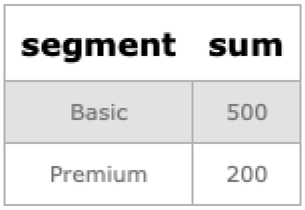
brand	segment	sum
aldo	Premium	100
hnm	Basic	200
baggit	Basic	300
zara	Premium	100

```
SELECT
    brand,
    SUM (quantity)
FROM
    sales
GROUP BY
    brand;
```



brand	sum
hnm	200
aldo	100
zara	100
baggit	300

```
SELECT
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    segment;
```



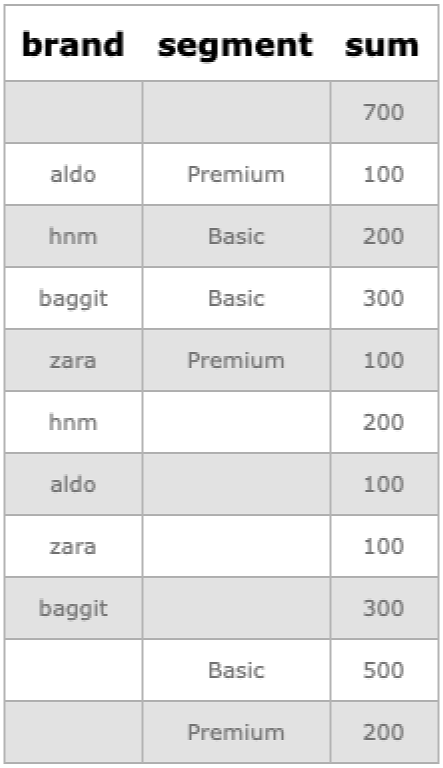
segment	sum
Basic	500
Premium	200

General syntax of grouping sets

```
SELECT
    c1,
    c2,
    aggregate_function(c3)
FROM
    table_name
GROUP BY
    GROUPING SETS (
        (c1, c2),
        (c1),
        (c2),
        ()
);
```

For example:

```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    GROUPING SETS (
        (brand, segment),
        (brand),
        (segment),
        ()
    );
```



brand	segment	sum
		700
aldo	Premium	100
hnm	Basic	200
baggit	Basic	300
zara	Premium	100
hnm		200
aldo		100
zara		100
baggit		300
	Basic	500
	Premium	200

CUBE

CUBE is a subclause of the GROUP BY clause. The CUBE allows you to generate multiple grouping sets.

Syntax:

```
SELECT
    c1,
    c2,
    c3,
    aggregate (c4)
FROM
    table_name
GROUP BY
    CUBE (c1, c2, c3);
```

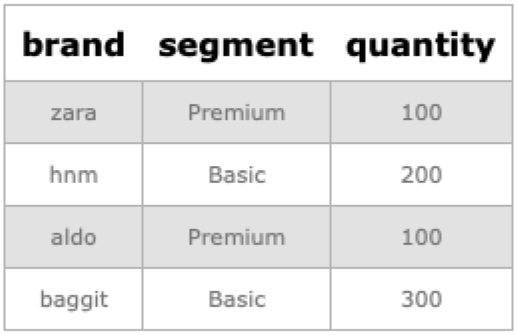
The CUBE subclause is a short way to define multiple grouping sets, so the following two queries are equivalent.

```
CUBE(c1,c2,c3)
```

```
GROUPING SETS (
    (c1,c2,c3),
    (c1,c2),
    (c1,c3),
    (c2,c3),
    (c1),
    (c2),
    (c3),
    ()
 )
```

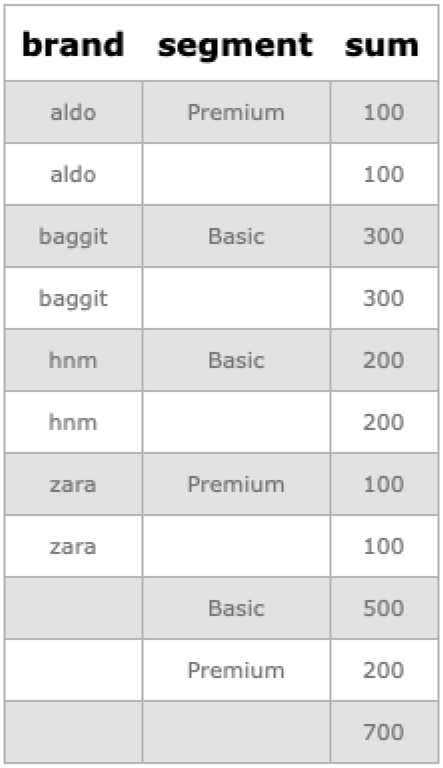
We will use the ‘sales’ table to understand the CUBE clause:

```
SELECT * FROM sales;
```



brand	segment	quantity
zara	Premium	100
hnm	Basic	200
aldo	Premium	100
baggit	Basic	300

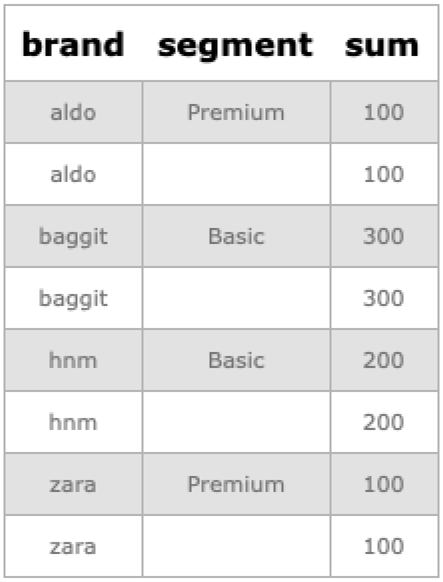
```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    CUBE (brand, segment)
ORDER BY
    brand,
    segment;
```



brand	segment	sum
aldo	Premium	100
aldo		100
baggit	Basic	300
baggit		300
hnm	Basic	200
hnm		200
zara	Premium	100
zara		100
	Basic	500
	Premium	200
		700

- **Partial cube**

```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    brand,
    CUBE (segment)
ORDER BY
    brand,
    segment;
```



brand	segment	sum
aldo	Premium	100
aldo		100
baggit	Basic	300
baggit		300
hnm	Basic	200
hnm		200
zara	Premium	100
zara		100

Roll-Up

ROLLUP is a subclause of the GROUP BY clause that offers a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not generate all possible grouping sets based on the specified columns. It just makes a subset.

Syntax:

```
SELECT
    c1,
    c2,
    c3,
    aggregate(c4)
FROM
    table_name
GROUP BY
    ROLLUP (c1, c2, c3);
```

- **Partial roll-up syntax**

```
SELECT
    c1,
    c2,
    c3,
    aggregate(c4)
FROM
```

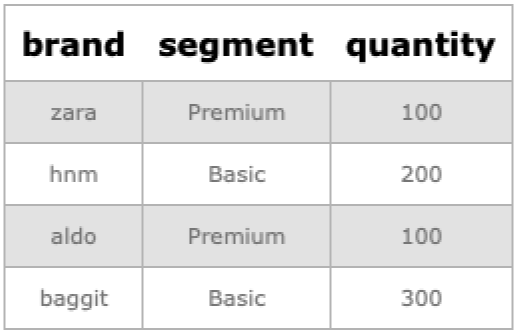
```
    table_name
GROUP BY
    c1,
    ROLLUP (c2, c3);
```

For example, we have created the ‘sales’ table and inserted data:

```
DROP TABLE IF EXISTS sales;
CREATE TABLE sales (
    brand VARCHAR NOT NULL,
    segment VARCHAR NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (brand, segment)
);
```

```
INSERT INTO sales (brand, segment, quantity)
VALUES
    ('zara', 'Premium', 100),
    ('hnm', 'Basic', 200),
    ('aldo', 'Premium', 100),
    ('baggit', 'Basic', 300);
```

SELECT * FROM sales:

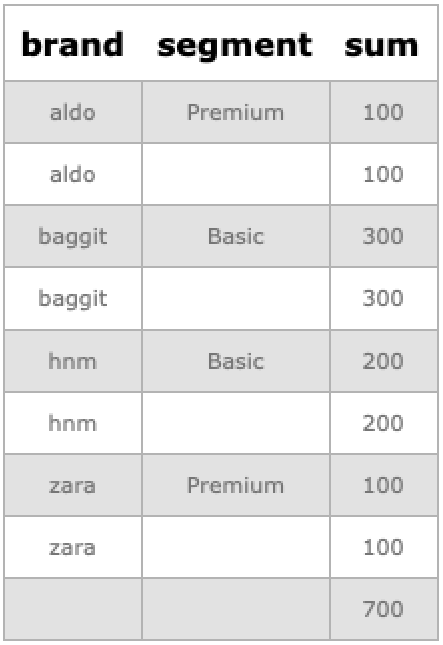


brand	segment	quantity
zara	Premium	100
hnm	Basic	200
aldo	Premium	100
baggit	Basic	300

- **Roll-up example**

```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
```

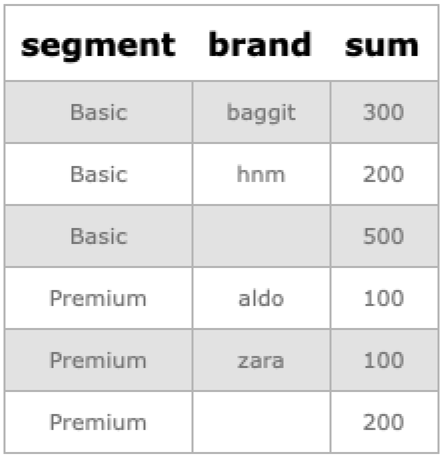
```
    sales
GROUP BY
    ROLLUP (brand, segment)
ORDER BY
    brand,
    segment;
```



brand	segment	sum
aldo	Premium	100
aldo		100
baggit	Basic	300
baggit		300
hnm	Basic	200
hnm		200
zara	Premium	100
zara		100
		700

- **Partial roll-up example**

```
SELECT
    segment,
    brand,
    SUM (quantity)
FROM
    sales
GROUP BY
    segment,
    ROLLUP (brand)
ORDER BY
    segment,
    brand;
```



segment	brand	sum
Basic	baggit	300
Basic	hnm	200
Basic		500
Premium	aldo	100
Premium	zara	100
Premium		200

Section 7 - Subquery

Subquery

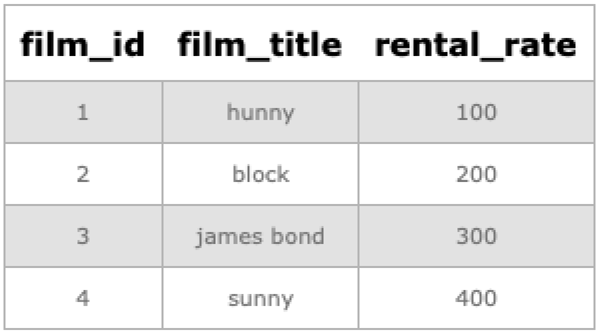
Retrieving a specific output sometimes requires running more than one query. To reduce the steps, we use the subquery — a query inside another query.

Suppose we want to know the movies whose rental rate is greater than the average. We will create the ‘film’ table and insert values into it:

```
DROP TABLE IF EXISTS film;
CREATE TABLE film (
    film_id INT NOT NULL,
    film_title VARCHAR NOT NULL,
    rental_rate INT NOT NULL,

    PRIMARY KEY (film_id)
);
```

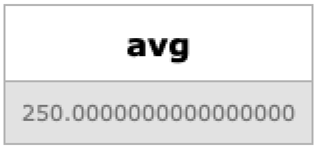
```
INSERT INTO film (film_id, film_title, rental_rate)
VALUES
    (1, 'hunny', 100),
    (2, 'block', 200),
    (3, 'james bond', 300),
    (4, 'sunny', 400);
```

```
SELECT * FROM film;
```



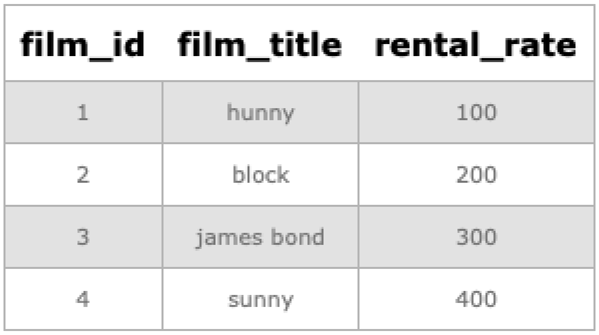
film_id	film_title	rental_rate
1	hunny	100
2	block	200
3	james bond	300
4	sunny	400

```
SELECT
      AVG (rental_rate)
FROM
      film;
```



avg
250.0000000000000000

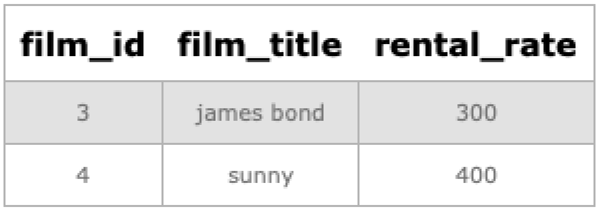
```
SELECT
      film_id,
      film_title,
      rental_rate
FROM
      film
WHERE
      rental_rate > 2.98;
```



film_id	film_title	rental_rate
1	hunny	100
2	block	200
3	james bond	300
4	sunny	400

We use the subquery:

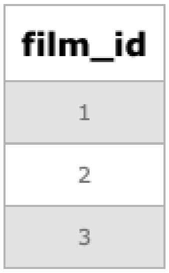
```
SELECT
      film_id,
      film_title,
      rental_rate
FROM
      film
WHERE
      rental_rate > (
            SELECT
                  AVG (rental_rate)
            FROM
                  film
      );
```



film_id	film_title	rental_rate
3	james bond	300
4	sunny	400

- **Subquery with IN operator**

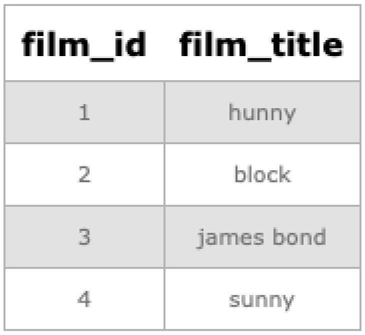
```
SELECT
      film_id
FROM
      film
WHERE
      salary BETWEEN '100'
AND '300';
```



film_id
1
2
3

The above example will only allow a single column output. To get multiple columns, use the subquery:

```
SELECT
      film_id,
      film_title
FROM
      film
WHERE
      film_id IN (
            SELECT
                  film_id
            FROM
                  film
                        WHERE
                  rental_rate BETWEEN '100'
            AND '400'
      );
```



film_id	film_title
1	hunny
2	block
3	james bond
4	sunny

ANY

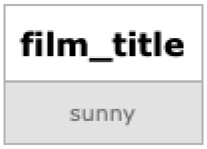
The ANY operator compares a value to a set of values returned by a subquery. It returns true if any subquery value meets the condition; otherwise, it returns false.

Syntax:

```
expression operator ANY(subquery)
```

For example, find the films whose lengths are greater than or equal to the maximum length of any film category:

```
SELECT film_title
FROM film
WHERE rental_rate >= ANY(
    SELECT MAX( rental_rate )
    FROM film
       );
```



film_title
sunny

ALL

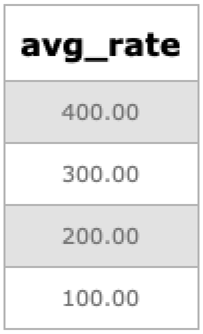
The ALL operator allows you to query data by comparing a value with a list of values returned by a subquery.

Syntax:

```
comparison_operator ALL (subquery)
```

To find the average lengths of all films grouped by film rating, run the following query:

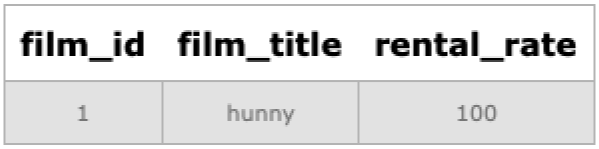
```
SELECT
    ROUND(AVG(rental_rate), 2) avg_rate
FROM
    film
GROUP BY
    film_id
ORDER BY
    avg_rate DESC;
```



avg_rate
400.00
300.00
200.00
100.00

Find all films whose lengths are greater than the list of the average lengths above:

```
SELECT
    film_id,
    film_title,
    rental_rate
FROM
    film
WHERE
    rental_rate <= ALL (
            SELECT
                ROUND(AVG (rental_rate),2)
            FROM
                film
            GROUP BY
                film_id
    )
ORDER BY
    rental_rate;
```



film_id	film_title	rental_rate
1	hunny	100

EXISTS

This is a boolean operator that tests for the existence of rows in a subquery.

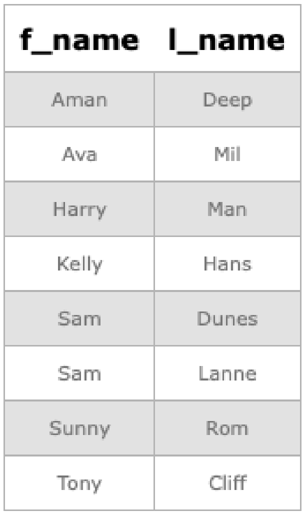
Syntax:

```
SELECT
    column1
FROM
    table_1
WHERE
    EXISTS( SELECT
                1
            FROM
                table_2
            WHERE
                column_2 = table_1.column_1);
```

It will accept the subquery as an argument. If the subquery returns at least one row, the result of EXISTS is true. In case the subquery returns no row, the result of EXISTS is false. The EXISTS operator is often used with the correlated subquery.

- **Customers with at least one payment whose amount is greater than 11**

```
SELECT f_name,
       l_name
FROM employee_data
WHERE EXISTS
    (SELECT 1
     FROM employee_data
     WHERE salary > 2000 )
ORDER BY f_name,
         l_name;
```



f_name	l_name
Aman	Deep
Ava	Mil
Harry	Man
Kelly	Hans
Sam	Dunes
Sam	Lanne
Sunny	Rom
Tony	Cliff

- **NOT EXISTS**

```
SELECT f_name,
       l_name
FROM employee_data
WHERE NOT EXISTS
    (SELECT 1
     FROM employee_data
     WHERE salary < 3000)
ORDER BY f_name,
         l_name;
```

No result set.

Section 8 - Common Table Expressions

A common table expression is a temporary result set that you can reference within another SQL statement, including SELECT, INSERT, UPDATE, or DELETE. Common table expressions are temporary as they only exist during the query's execution.

Syntax:

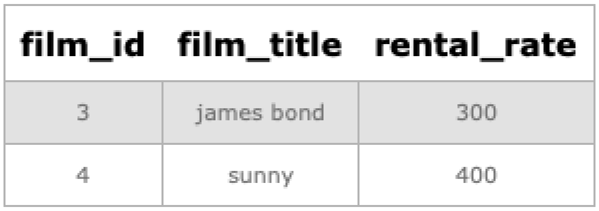
```
WITH cte_name (column_list) AS (
    CTE_query_definition
)
statement;
```

For example:

```
WITH cte_film AS (
    SELECT
        film_id,
        Film_title, rental_rate
          FROM
        film
)
```

```
SELECT
    film_id,
    film_title,
    rental_rate
```

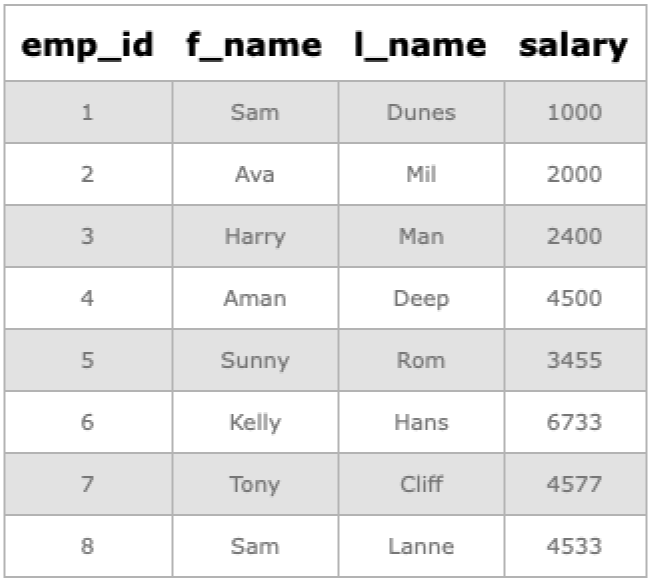
```
FROM
    cte_film
WHERE
  rental_rate>200
ORDER BY
    film_title;
```



film_id	film_title	rental_rate
3	james bond	300
4	sunny	400

- **Joining CTE with table**

```
WITH cte_emp AS (
    SELECT emp_id,
        COUNT(salary) count
    FROM   employee_data
    GROUP  BY manager_id, emp_id
)
SELECT emp_id,
    f_name,
    l_name,
    salary
FROM employee_data;
```



emp_id	f_name	l_name	salary
1	Sam	Dunes	1000
2	Ava	Mil	2000
3	Harry	Man	2400
4	Aman	Deep	4500
5	Sunny	Rom	3455
6	Kelly	Hans	6733
7	Tony	Cliff	4577
8	Sam	Lanne	4533

Recursive Query

A recursive query refers to a recursive CTE. They're useful in querying hierarchical data like organizational structure, bill of materials, etc.

Syntax:

```
WITH RECURSIVE cte_name AS(
    CTE_query_definition -- non-recursive term
    UNION [ALL]
    CTE_query definion  -- recursive term
) SELECT * FROM cte_name;
```

For example, we have created the 'employees' table and inserted some data:

```
CREATE TABLE employees (
      employee_id serial PRIMARY KEY,
      full_name VARCHAR NOT NULL,
      manager_id INT
);
```

```
INSERT INTO employees (
      employee_id,
      full_name,
      manager_id
)
VALUES
      (1, 'Michael North', NULL),
      (2, 'Megan Berry', 1),
      (3, 'Sarah Berry', 1),
      (4, 'Zoe Black', 1),
      (5, 'Tim James', 1),
      (6, 'Bella Tucker', 2),
      (7, 'Ryan Metcalfe', 2),
      (8, 'Max Mills', 2),
      (9, 'Benjamin Glover', 2),
      (10, 'Carolyn Henderson', 3),
      (11, 'Nicola Kelly', 3),
      (12, 'Alexandra Climo', 3),
      (13, 'Dominic King', 3),
      (14, 'Leonard Gray', 4),
      (15, 'Eric Rampling', 4),
      (16, 'Piers Paige', 7),
      (17, 'Ryan Henderson', 7),
      (18, 'Frank Tucker', 8),
      (19, 'Nathan Ferguson', 8),
      (20, 'Kevin Rampling', 8);
```

To get all manager subordinates with the id 2:

```
WITH RECURSIVE subordinates AS (
      SELECT
            employee_id,
            manager_id,
            full_name
      FROM
            employees
      WHERE
            employee_id = 2
      UNION
            SELECT
                  e.employee_id,
                  e.manager_id,
                  e.full_name
            FROM
                  employees e
            INNER JOIN subordinates s ON s.employee_id = e.manager_id
) SELECT
      *
FROM
      subordinates;
```

The non-recursive term returns the base result set R0, the employee with the id 2.

```
 employee_id | manager_id |  full_name
-------------+------------+-------------
           2 |          1 | Megan Berry
```

The recursive term's first iteration returns the following result set:

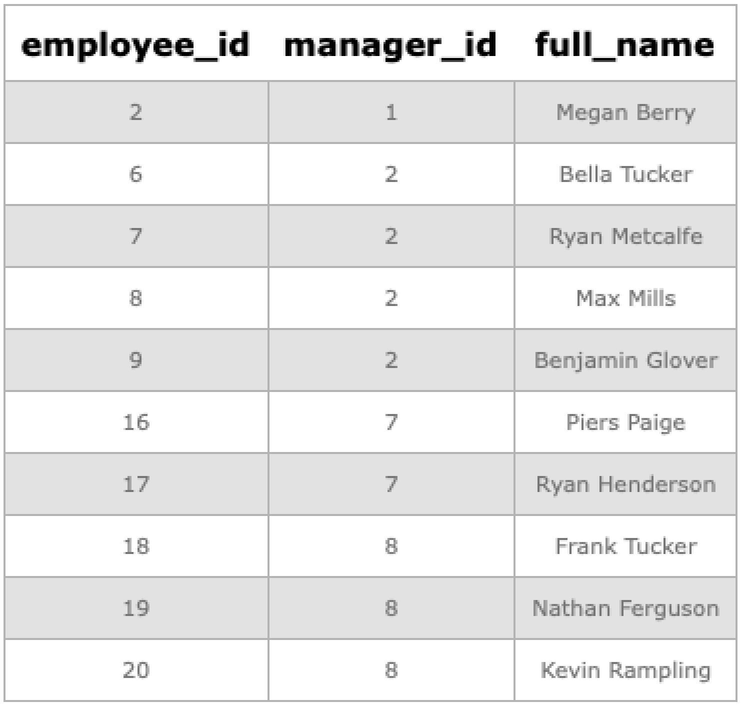
```
 employee_id | manager_id |    full_name
-------------+------------+-----------------
           6 |          2 | Bella Tucker
           7 |          2 | Ryan Metcalfe
           8 |          2 | Max Mills
           9 |          2 | Benjamin Glover
```

The second iteration of the recursive member uses the result set above step as the input value, and returns this result set.

```
 employee_id | manager_id |    full_name
-------------+------------+-----------------
          16 |          7 | Piers Paige
          17 |          7 | Ryan Henderson
          18 |          8 | Frank Tucker
          19 |          8 | Nathan Ferguson
          20 |          8 | Kevin Rampling
```

The third iteration returns an empty result set as no employee reports to the employee with id 16, 17, 18, 19, and 20.

The following is the final result set, which is the union of all result sets in the first and second iterations generated by the non-recursive and recursive terms.



employee_id	manager_id	full_name
2	1	Megan Berry
6	2	Bella Tucker
7	2	Ryan Metcalfe
8	2	Max Mills
9	2	Benjamin Glover
16	7	Piers Paige
17	7	Ryan Henderson
18	8	Frank Tucker
19	8	Nathan Ferguson
20	8	Kevin Rampling

Section 9 - Modifying Data

In this section, you will learn how to:

- insert data into a table with the INSERT statement
- modify existing data with the UPDATE statement
- remove data with the DELETE statement
- merge data with the UPSERT statement

Insert

This statement allows you to insert a new row into a table.

Syntax:

```
INSERT INTO table_name(column1, column2, ...)
VALUES (value1, value2, ...);
```

- **Returning clause**

If you want to return the entire inserted row, you use an asterisk (*) after the RETURNING keyword.

```
INSERT INTO table_name(column1, column2, ...)
VALUES (value1, value2, ...)
RETURNING *;
```

To return some information about the inserted row, you can specify one or more columns after the RETURNING clause.

```
INSERT INTO table_name(column1, column2, ...)
VALUES (value1, value2, ...)
RETURNING id;
```

- **Inserting single row**

```
INSERT INTO links (url, name)
VALUES('https://www.xyz.com', 'data');
```

- **Inserting character strings containing singgleute**

```
INSERT INTO links (url, name)
VALUES('http://www.xyz.com','O''XYZ');
```

- **Insert date value**

```
INSERT INTO links (url, name, last_update)
VALUES('https://www.google.com','Google','2013-06-01');
```

- **Getting the last insert id**

```
INSERT INTO links (url, name)
VALUES('http://www.xyz.org','PostgreSQL')
RETURNING id;
```

INSERT Multiple Rows

Use the following syntax to insert multiple rows.

```
INSERT INTO table_name (column_list)
VALUES
    (value_list_1),
    (value_list_2),
    ...
    (value_list_n);
```

To insert and return multiple rows, use the returning clause:

```
INSERT INTO table_name (column_list)
VALUES
    (value_list_1),
    (value_list_2),
    ...
    (value_list_n)
RETURNING * | output_expression;
```

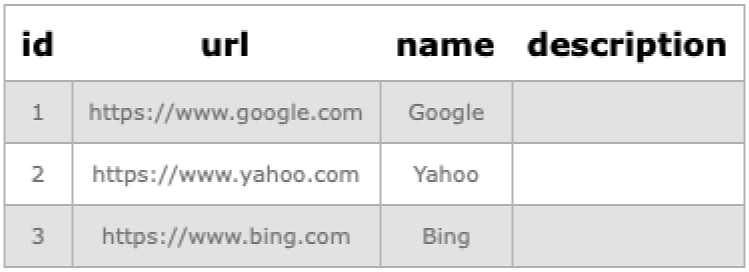
For example, we will create the ‘links’ table and insert data:

```
DROP TABLE IF EXISTS links;

CREATE TABLE links (
    id SERIAL PRIMARY KEY,
    url VARCHAR(255) NOT NULL,
    name VARCHAR(255) NOT NULL,
    description VARCHAR(255)
);
```

- **Insert multiple rows using the below query**

```
INSERT INTO
    links (url, name)
VALUES
    ('https://www.google.com','Google'),
    ('https://www.yahoo.com','Yahoo'),
    ('https://www.bing.com','Bing');
```

```
SELECT * FROM links;
```

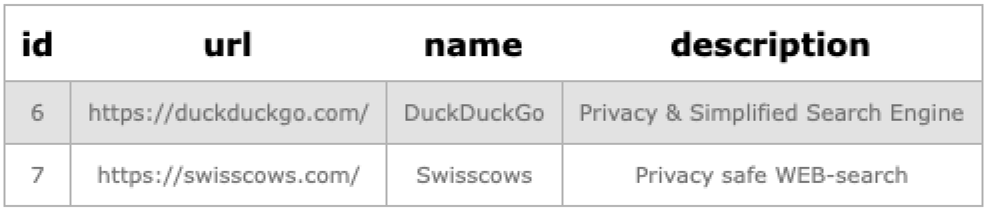


id	url	name	description
1	https://www.google.com	Google	
2	https://www.yahoo.com	Yahoo	
3	https://www.bing.com	Bing	

- **Insert and return multiple columns**

```
INSERT INTO
    links(url,name, description)
VALUES
    ('https://duckduckgo.com/','DuckDuckGo','Privacy & Simplified Search
Engine'),
    ('https://swisscows.com/','Swisscows','Privacy safe WEB-search')
RETURNING *;
```

```
SELECT * FROM links;
```



id	url	name	description
6	https://duckduckgo.com/	DuckDuckGo	Privacy & Simplified Search Engine
7	https://swisscows.com/	Swisscows	Privacy safe WEB-search

Update

UPDATE allows you to modify data in a table.

Syntax:

```
UPDATE table_name
SET column1 = value1,
    column2 = value2,
    ...
WHERE condition;
```

- **Return updated rows**

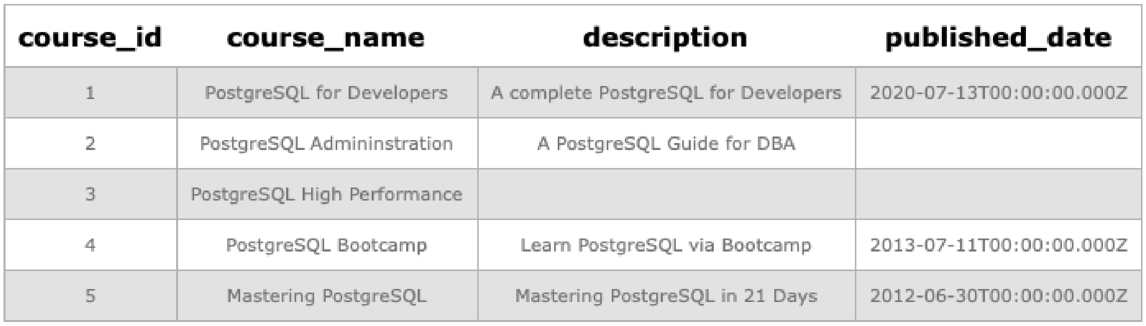
```
UPDATE table_name
SET column1 = value1,
    column2 = value2,
    ...
WHERE condition
RETURNING * | output_expression AS output_name;
```

To explain, we will create the ‘courses’ table and insert data:

```
DROP TABLE IF EXISTS courses;

CREATE TABLE courses(
      course_id serial primary key,
      course_name VARCHAR(255) NOT NULL,
      description VARCHAR(500),
      published_date date
);
```

```
INSERT INTO
      courses(course_name, description, published_date)
VALUES
      ('PostgreSQL for Developers','A complete PostgreSQL for
Developers','2020-07-13'),
      ('PostgreSQL Admininstration','A PostgreSQL Guide for DBA',NULL),
      ('PostgreSQL High Performance',NULL,NULL),
      ('PostgreSQL Bootcamp','Learn PostgreSQL via Bootcamp','2013-07-11'),
      ('Mastering PostgreSQL','Mastering PostgreSQL in 21
Days','2012-06-30');
```

SELECT * FROM courses:

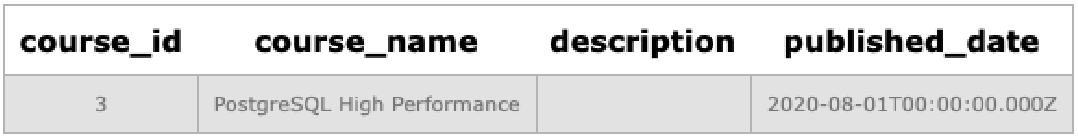


course_id	course_name	description	published_date
1	PostgreSQL for Developers	A complete PostgreSQL for Developers	2020-07-13T00:00:00.000Z
2	PostgreSQL Admininstration	A PostgreSQL Guide for DBA	
3	PostgreSQL High Performance		
4	PostgreSQL Bootcamp	Learn PostgreSQL via Bootcamp	2013-07-11T00:00:00.000Z
5	Mastering PostgreSQL	Mastering PostgreSQL in 21 Days	2012-06-30T00:00:00.000Z

- **Update a single row**

```
UPDATE courses
SET published_date = '2020-08-01'
WHERE course_id = 3;
```

```
SELECT * FROM COURSES WHERE COURSE_ID=3;
```



course_id	course_name	description	published_date
3	PostgreSQL High Performance		2020-08-01T00:00:00.000Z

Update Join

Use this statement to update data in a table based on values in another table.

Syntax:

```
UPDATE t1
SET t1.c1 = new_value
FROM t2
WHERE t1.c2 = t2.c2;
```

For example, we have created two tables and inserted data into it.

–Table 1

```
CREATE TABLE prod (
    id SERIAL PRIMARY KEY,
    segment VARCHAR NOT NULL,
    discount NUMERIC (4, 2)
);
```

```
INSERT INTO
    Prod (segment, discount)
VALUES
    ('Grand Luxury', 0.05),
    ('Luxury', 0.06),
    ('Mass', 0.1);
```

–Table 2

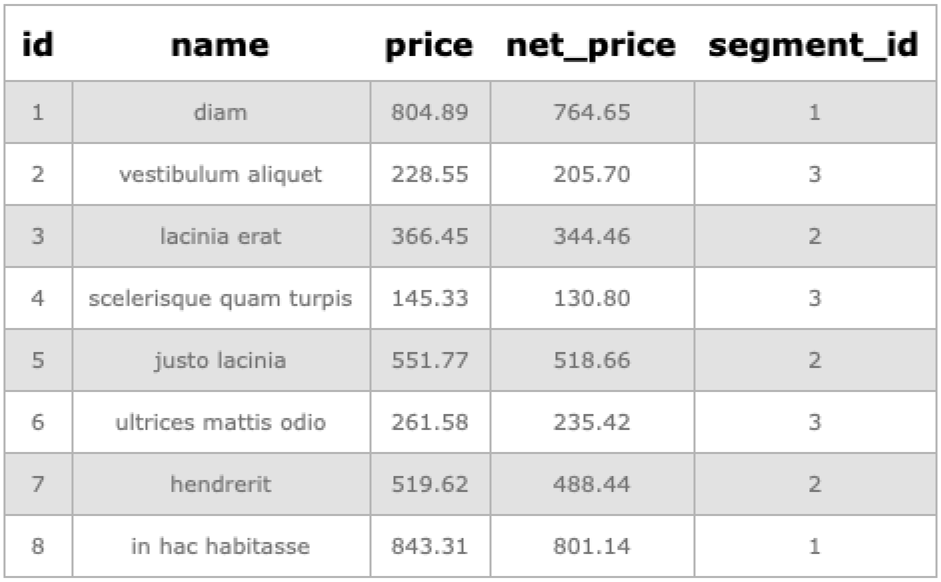
```
DROP TABLE IF EXISTS product;
CREATE TABLE product(
    id SERIAL PRIMARY KEY,
    name VARCHAR NOT NULL,
    price NUMERIC(10,2),
    net_price NUMERIC(10,2),
    segment_id INT NOT NULL,
    FOREIGN KEY(segment_id) REFERENCES prod(id)
);
```

```
INSERT INTO
    product (name, price, segment_id)
VALUES
    ('diam', 804.89, 1),
    ('vestibulum aliquet', 228.55, 3),
    ('lacinia erat', 366.45, 2),
    ('scelerisque quam turpis', 145.33, 3),
    ('justo lacinia', 551.77, 2),
    ('ultrices mattis odio', 261.58, 3),
    ('hendrerit', 519.62, 2),
    ('in hac habitasse', 843.31, 1)
   ;
```

Now, we'll run an update join query:

```
UPDATE product
SET net_price = price - price * discount
FROM prod
WHERE product.segment_id = prod.id;
```

```
SELECT * FROM product;
```



id	name	price	net_price	segment_id
1	diam	804.89	764.65	1
2	vestibulum aliquet	228.55	205.70	3
3	lacinia erat	366.45	344.46	2
4	scelerisque quam turpis	145.33	130.80	3
5	justo lacinia	551.77	518.66	2
6	ultrices mattis odio	261.58	235.42	3
7	hendrerit	519.62	488.44	2
8	in hac habitasse	843.31	801.14	1

Delete

This statement allows you to delete one or more rows from a table.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Use the RETURNING clause to return the deleted row(s) to the client:

```
DELETE FROM table_name
WHERE condition
RETURNING (select_list | *)
```

- **Deleting one row**

```
DELETE FROM links
WHERE id = 8;
```

- **Deleting and returning one row.**

```
DELETE FROM links
WHERE id = 7
RETURNING *;
```

- **Deleting multiple rows**

```
DELETE FROM links
WHERE id IN (6,5)
RETURNING *;
```

UPSERT

This statement will insert or update data into an existing row. The idea is that when you insert a new row into the table, PostgreSQL will update the row if it already exists; otherwise, it will insert the new row. That is why we call the action upsert.

To use the upsert feature in PostgreSQL, use the INSERT ON CONFLICT statement.

```
INSERT INTO table_name(column_list)
VALUES(value_list)
ON CONFLICT target action;
```

For example, we will create the ‘customers’ table and insert data into it:

```
DROP TABLE IF EXISTS customers;
```

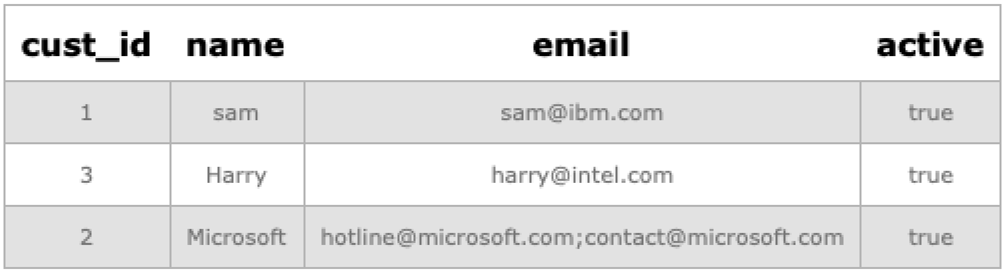
```
CREATE TABLE customers (
      cust_id serial PRIMARY KEY,
      name VARCHAR UNIQUE,
      email VARCHAR NOT NULL,
      active bool NOT NULL DEFAULT TRUE
);
```

```
INSERT INTO
    customers (name, email)
VALUES
    ('sam', 'sam@ibm.com'),
    ('Microsoft', 'contact@microsoft.com'),
    ('Harry', 'harry@intel.com');
```

Now we can use the following query to change the email:

```
INSERT INTO customers (name, email)
VALUES('Microsoft','hotline@microsoft.com')
ON CONFLICT (name)
DO
```

```
   UPDATE SET email = EXCLUDED.email || ';' || customers.email;
```



cust_id	name	email	active
1	sam	sam@ibm.com	true
3	Harry	harry@intel.com	true
2	Microsoft	hotline@microsoft.com;contact@microsoft.com	true

Section 10 - Transactions

A database transaction is a single unit of work with several operations. A PostgreSQL transaction is atomic, consistent, isolated, and durable, represented as ACID properties.

For example:

```
DROP TABLE IF EXISTS accounts;

CREATE TABLE accounts (
    id INT GENERATED BY DEFAULT AS IDENTITY,
    name VARCHAR(100) NOT NULL,
    balance DEC(15,2) NOT NULL,
    PRIMARY KEY(id)
);
```

- **Begin a transaction**

Use the BEGIN statement at the start of your statements.

```
BEGIN;

INSERT INTO accounts(name,balance)
VALUES('Alice',10000);
```

- **Commit a transaction.**

This makes your changes permanent, even if a crash happens.

-- Start a transaction.

```
BEGIN;
```

-- Insert a new row into the accounts table.

```
INSERT INTO accounts(name,balance)
VALUES('Alice',10000);
```

-- Commit the change (or roll it back later).

```
COMMIT;
```

- **Rollback a transaction.**

You can rollback any transaction until the last commit statement.

-- Begin the transaction.

```
BEGIN;
```

-- Deduct the amount from account 1.

```
UPDATE accounts
SET balance = balance - 1500
WHERE id = 1;
```

-- Add the amount from account 3 (instead of 2).

```
UPDATE accounts
SET balance = balance + 1500
WHERE id = 3;
```

-- Roll back the transaction.

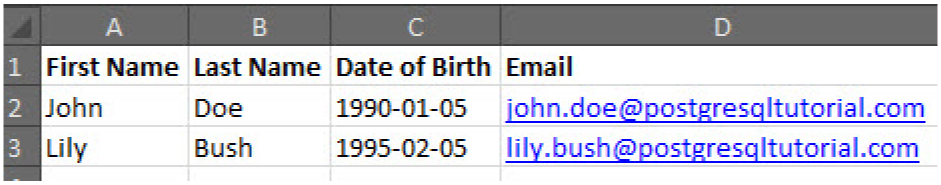
```
ROLLBACK;
```

Section 11 - Import CSV File Into PostgreSQL Table

We will now create the ‘persons’ table to understand how to import a CSV file into a PostgreSQL table.

```
CREATE TABLE persons (
  id SERIAL,
  f_name VARCHAR(50),
  l_name VARCHAR(50),
  dob DATE,
  email VARCHAR(255),
  PRIMARY KEY (id)
)
```

Create a CSV with the following format:

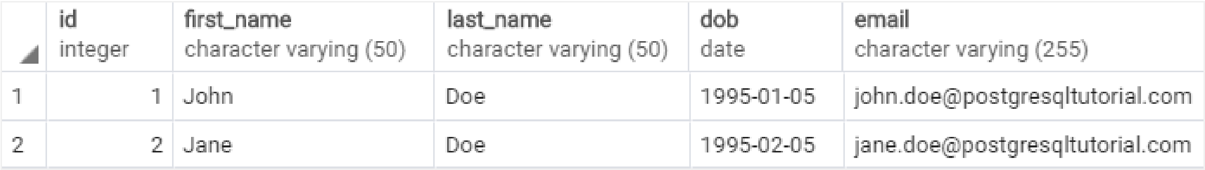


	A	B	C	D
1	**First Name**	**Last Name**	**Date of Birth**	**Email**
2	John	Doe	1990-01-05	john.doe@postgresqltutorial.com
3	Lily	Bush	1995-02-05	lily.bush@postgresqltutorial.com

- **Importing with the copy statement**

```
COPY persons(first_name, last_name, dob, email)
FROM 'C:\sampledb\persons.csv'
DELIMITER ','
CSV HEADER;
```

```
SELECT * FROM persons;
```

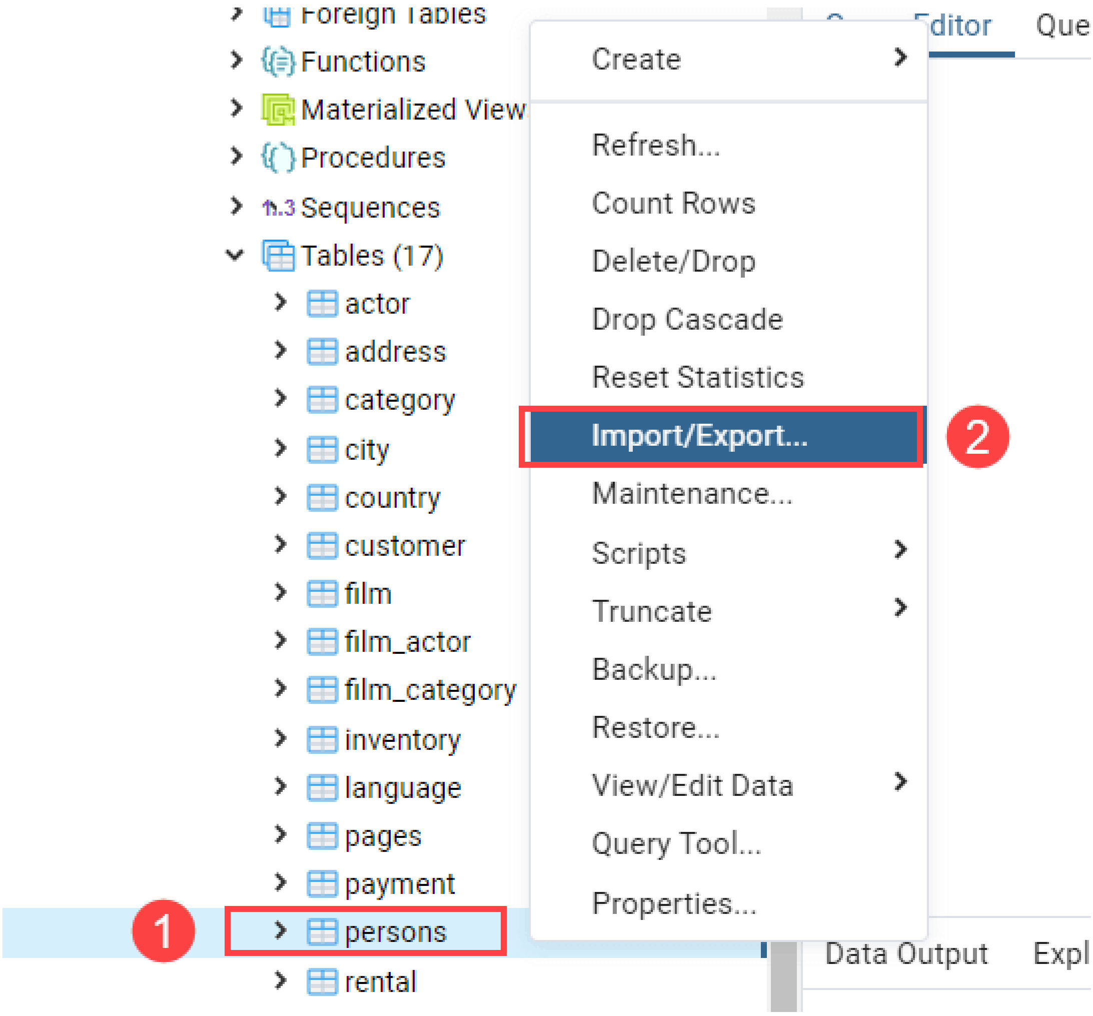


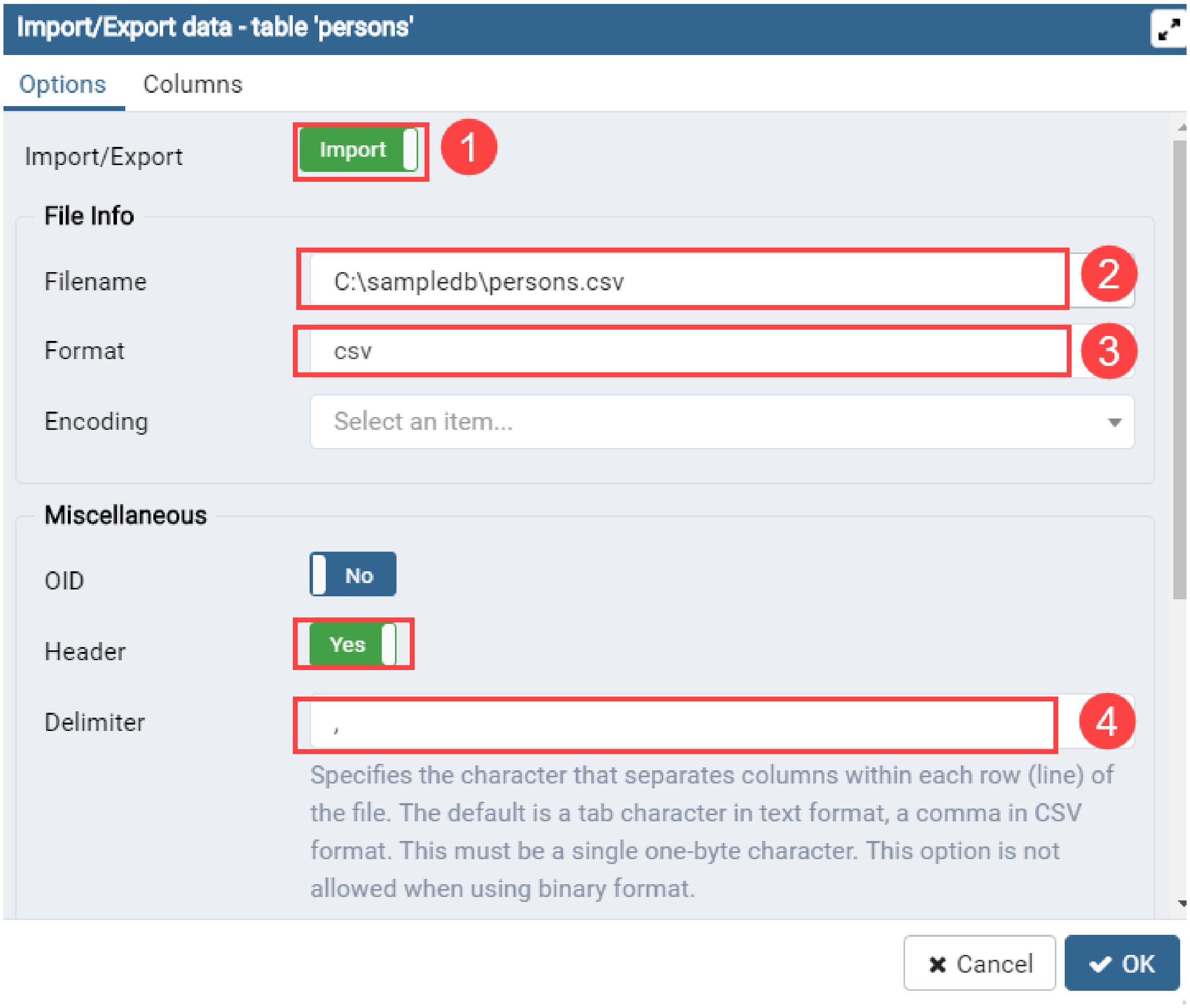
	id integer	first_name character varying (50)	last_name character varying (50)	dob date	email character varying (255)
1	1	John	Doe	1995-01-05	john.doe@postgresqltutorial.com
2	2	Jane	Doe	1995-02-05	jane.doe@postgresqltutorial.com

- **Using pgAdmin**

The following query will truncate the table to start again:

```
TRUNCATE TABLE persons
RESTART IDENTITY;
```


Foreign Tables
Functions
Materialized View
Procedures
Sequences
Tables (17)
actor
address
category
city
country
customer
film
film_actor
film_category
inventory
language
pages
payment
persons
rental
1
Editor
Que
Create
Refresh...
Count Rows
Delete/Drop
Drop Cascade
Reset Statistics
Import/Export...
2
Maintenance...
Scripts
Truncate
Backup...
Restore...
View/Edit Data
Query Tool...
Properties...
Data Output
Expl



Import/Export data - table 'persons'

Import/Export: Import (1)

File Info

Filename: C:\sampledb\persons.csv (2)

Format: csv (3)

Encoding: Select an item...

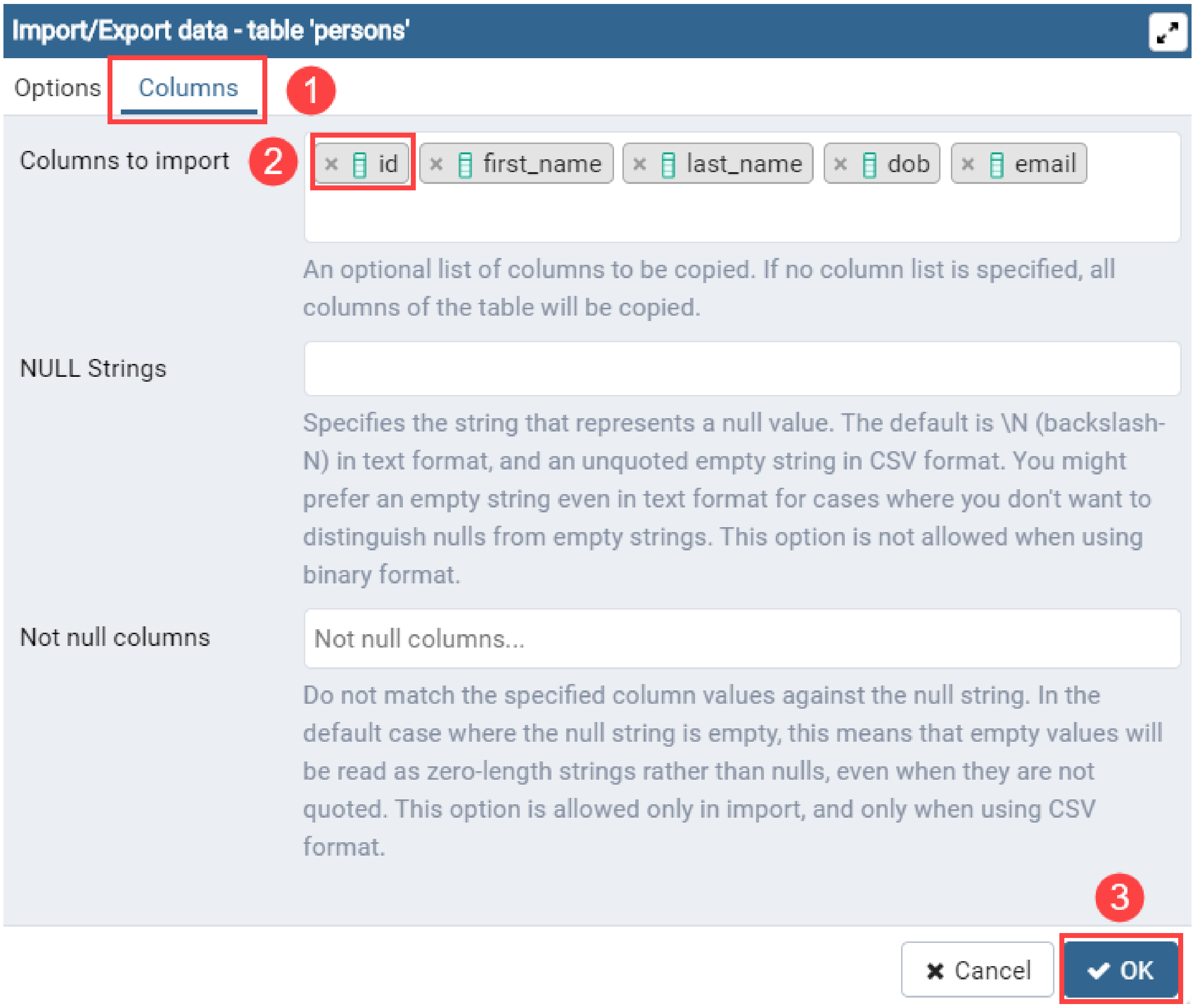
Miscellaneous

OID: No

Header: Yes

Delimiter: , (4)

Specifies the character that separates columns within each row (line) of the file. The default is a tab character in text format, a comma in CSV format. This must be a single one-byte character. This option is not allowed when using binary format.



Import/Export data - table 'persons'

Columns to import: id, first_name, last_name, dob, email

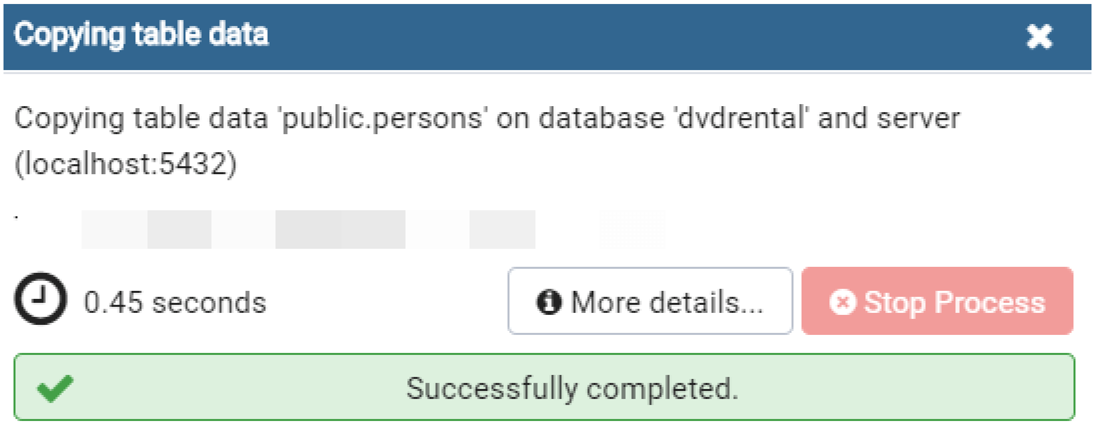
An optional list of columns to be copied. If no column list is specified, all columns of the table will be copied.

NULL Strings

Specifies the string that represents a null value. The default is \N (backslash-N) in text format, and an unquoted empty string in CSV format. You might prefer an empty string even in text format for cases where you don't want to distinguish nulls from empty strings. This option is not allowed when using binary format.

Not null columns: Not null columns...

Do not match the specified column values against the null string. In the default case where the null string is empty, this means that empty values will be read as zero-length strings rather than nulls, even when they are not quoted. This option is allowed only in import, and only when using CSV format.



Copying table data

Copying table data 'public.persons' on database 'dvdrental' and server (localhost:5432)

0.45 seconds

Successfully completed.

Section 12 - Managing Tables in PostgreSQL

Data Types

- **Boolean:** Can hold one of three possible values: true, false or null.
- **CHAR(n):** the fixed-length character with space padded.
- **ARCHAR(n**): variable-length character string and can store upto n characters.
- **TEXT**: variable-length character string.
- **Small integer (SMALLINT):** 2-byte signed integer that ranges from -32,768 to 32,767.
- **Integer (INT):** 4-byte integer that has a range from -2,147,483,648 to 2,147,483,647.
- **float(n):** floating-point number whose precision, at least, n, up to a maximum of 8 bytes.
- **realor float8:** 4-byte floating-point number.
- **numeric or numeric(p,s)**: a real number with p digits with s number after the decimal point. The numeric(p,s) is the exact number.
- **DATE**: stores the dates only.
- **TIME**: stores the time of day values.
- **TIMESTAMP**: stores both date and time values.
- **TIMESTAMPTZ**: timezone-aware timestamp data type. It is the abbreviation for timestamp with the time zone.
- **INTERVAL**: stores periods of time.

Create Table

Use the CREATE TABLE statement to create a new table:

Syntax:

```
CREATE TABLE [IF NOT EXISTS] table_name (
   column1 datatype(length) column_contraint,
   column2 datatype(length) column_contraint,
   column3 datatype(length) column_contraint,
   table_constraints
);
```

Select Into

This statement creates a new table and inserts data returned from a query into the table. The new table will have columns with the same names as the query's result set columns. Unlike a regular SELECT statement, the SELECT INTO statement does not return a result to the client.

Syntax:

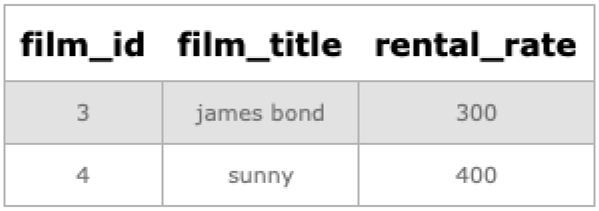
```
SELECT
    select_list
```

```
INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table_name
FROM
    table_name
WHERE
    search_condition;
```

For example:

```
SELECT
    film_id,
   film_ title,
    rental_rate
INTO TABLE film_r
FROM
    film
WHERE
   Rental_rate > 200
ORDER BY
   film_ title;
```

```
SELECT * FROM film_r;
```



film_id	film_title	rental_rate
3	james bond	300
4	sunny	400

Sequence

A sequence in PostgreSQL is a user-defined schema-bound object that generates a sequence of integers based on a specified specification.

```
CREATE SEQUENCE [ IF NOT EXISTS ] sequence_name
    [ AS { SMALLINT | INT | BIGINT } ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ CACHE cache ]
    [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

- **Creating an ascending sequence**

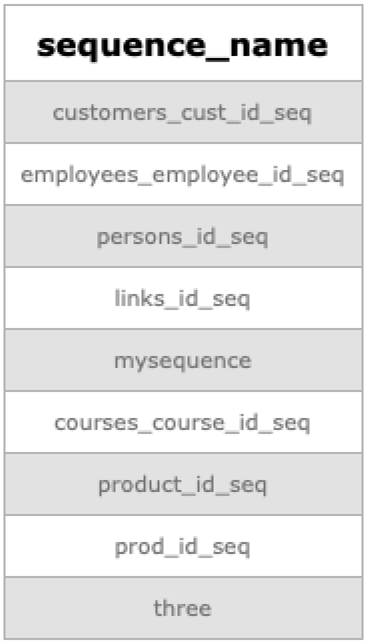
```
CREATE SEQUENCE mysequence
INCREMENT 5
START 100;
```

- **Creating a descending sequence**

```
CREATE SEQUENCE three
INCREMENT -1
MINVALUE 1
MAXVALUE 3
START 3
CYCLE;
```

- **Listing all sequences**

```
SELECT
    relname sequence_name
FROM
    pg_class
WHERE
    relkind = 'S';
```



sequence_name
customers_cust_id_seq
employees_employee_id_seq
persons_id_seq
links_id_seq
mysequence
courses_course_id_seq
product_id_seq
prod_id_seq
three

- **Deleting sequence**

```
DROP SEQUENCE [ IF EXISTS ] sequence_name [, ...]
[ CASCADE | RESTRICT ];
```

- **Drop sequence**

```
DROP TABLE order_details;
```

Identity Column

It allows you to automatically assign a unique number to a column. The GENERATED AS IDENTITY constraint is the SQL standard-conforming variant of the good old SERIAL column.

```
column_name type GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY[ (
sequence_option ) ]
```

- **Generated always examples**

```
CREATE TABLE color (
    color_id INT GENERATED ALWAYS AS IDENTITY,
    color_name VARCHAR NOT NULL
);
```

```
INSERT INTO color(color_name)
VALUES ('Red');
```

- **Generated by default as identity**

```
DROP TABLE color;

CREATE TABLE color (
    color_id INT GENERATED BY DEFAULT AS IDENTITY,
    color_name VARCHAR NOT NULL
);
```

```
INSERT INTO color (color_name)
VALUES ('White');
```

Alter Table

To change the structure of an existing table, use the PostgreSQL ALTER TABLE statement.

```
ALTER TABLE table_name action;
```

- **Adding a new column**

```
ALTER TABLE table_name
ADD COLUMN column_name datatype column_constraint;
```

- **Dropping a column**

```
ALTER TABLE table_name
ADD COLUMN column_name datatype column_constraint;
```

- **Renaming a column**

```
ALTER TABLE table_name
RENAME COLUMN column_name
TO new_column_name;
```

- **Changing default value of column**

```
ALTER TABLE table_name
ALTER COLUMN column_name
[SET DEFAULT value | DROP DEFAULT];
```

- **Changing not null constraint**

```
ALTER TABLE table_name
ALTER COLUMN column_name
[SET NOT NULL| DROP NOT NULL];
```

- **Adding check constraint**

```
ALTER TABLE table_name
ADD CHECK expression;
```

- **Renaming a table**

```
ALTER TABLE table_name
RENAME TO new_table_name;
```

Drop Table

Use the following syntax to drop a table:

```
DROP TABLE [IF EXISTS] table_name
[CASCADE | RESTRICT];
```

- The **CASCADE** option allows you to remove the table and its dependent objects.
- The **RESTRICT** option rejects the removal if there is any object, depending on the table. This is the default if you don't explicitly specify it in the DROP TABLE statement.

You can also drop multiple tables separated by commas:

```
DROP TABLE [IF EXISTS]
   table_name_1,
   table_name_2,
   ...
[CASCADE | RESTRICT];
```

- **Dropping table that does not exist**

```
DROP TABLE IF EXISTS author;
```

- **Dropping table with dependent objects**

```
CREATE TABLE authors (
      author_id INT PRIMARY KEY,
      firstname VARCHAR (50),
      lastname VARCHAR (50)
);
```

```
CREATE TABLE pages (
      page_id serial PRIMARY KEY,
      title VARCHAR (255) NOT NULL,
      contents TEXT,
      author_id INT NOT NULL,
      FOREIGN KEY (author_id)
          REFERENCES authors (author_id)
);
DROP TABLE IF EXISTS authors;
```

ERROR: cannot drop table authors because other objects depend on it
DETAIL: constraint pages_author_id_fkey on table pages depends on table authors
HINT: Use DROP ... CASCADE to drop the dependent objects too
SQL state: 2BP01

Run the following query:

```
DROP TABLE authors CASCADE;
```

- **Dropping multiple tables**

```
DROP TABLE tv shows, animes;
```

Truncate Table

To remove all data from a table, you use the DELETE statement. However, you might use the TRUNCATE TABLE statement for more efficiency.

```
TRUNCATE TABLE table_name;
```

- **Truncating multiple tables**

```
TRUNCATE TABLE
    table_name1,
    table_name2,
    ...;
```

- **Truncating table with foreign key references**

```
TRUNCATE TABLE table_name
```

```
CASCADE;
```

Temporary Table

A temporary table is a short-lived table that exists for the duration of a database session. PostgreSQL automatically drops the temporary tables at the end of a session or a transaction.

```
CREATE TEMPORARY TABLE temp_table_name(
   column_list
);
```

Section 13 - PostgreSQL Constraints

PostgreSQL includes the following column constraints:

- **NOT NULL:** Ensures that values in a column cannot be NULL.

```
CREATE TABLE table_name(
   ...
   column_name data_type NOT NULL,
   ...
);
```

- **UNIQUE:** Ensures values in a column are unique across the rows within the same table.

```
CREATE TABLE person (
      id SERIAL PRIMARY KEY,
      first_name VARCHAR (50),
      last_name VARCHAR (50),
      email VARCHAR (50) UNIQUE
);
```

- **PRIMARY KEY:** Identifies a table's primary key.

```
CREATE TABLE po_headers (
      po_no INTEGER PRIMARY KEY,
      vendor_no INTEGER,
      description TEXT,
      shipping_address TEXT
);
```

- **CHECK:** Ensures the data satisfies a boolean expression.

```
DROP TABLE IF EXISTS employees;
CREATE TABLE employees (
      id SERIAL PRIMARY KEY,
      first_name VARCHAR (50),
      last_name VARCHAR (50),
      birth_date DATE CHECK (birth_date > '1900-01-01'),
      joined_date DATE CHECK (joined_date > birth_date),
      salary numeric CHECK(salary > 0)
);
```

- **FOREIGN KEY:** Ensures values in a column or a group of columns from a table exists in a column or group of columns in another table. Unlike the primary key, a table can have many foreign keys.

```
[CONSTRAINT fk_name]
   FOREIGN KEY(fk_columns)
   REFERENCES parent_table(parent_key_columns)
   [ON DELETE delete_action]
   [ON UPDATE update_action]
```

Section 14 - Conditional Expressions & Operators

CASE

The CASE expression is the same as IF/ELSE statement in other programming languages. It allows you to add if-else logic to the query to form a powerful query.

```
CASE
      WHEN condition_1  THEN result_1
      WHEN condition_2  THEN result_2
      [WHEN ...]
      [ELSE else_result]
END
```

COALESCE

The COALESCE function accepts an unlimited number of arguments. It returns the first argument that is not null. If all arguments are null, the COALESCE function will return null.
This function evaluates arguments from left to right until it finds the first non-null argument. All the remaining arguments from the first non-null argument are not evaluated.

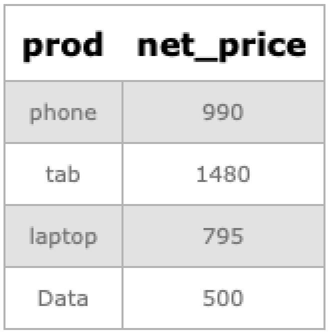
Syntax:

```
COALESCE (argument_1, argument_2, ...);
```

For example:

```
CREATE TABLE items (
      ID serial PRIMARY KEY,
      product VARCHAR (100) NOT NULL,
      price NUMERIC NOT NULL,
      discount NUMERIC
);
```

```
INSERT INTO items (product, price, discount)
VALUES
      ('A', 1000 ,10),
      ('B', 1500 ,20),
      ('C', 800 ,5),
      ('D', 500, NULL);

SELECT
      product,
      (price - COALESCE(discount,0)) AS net_price
FROM
      items;
```



prod	net_price
phone	990
tab	1480
laptop	795
Data	500

NULLIF

The NULLIF function is one of the most common conditional PostgreSQL expressions.

Syntax:

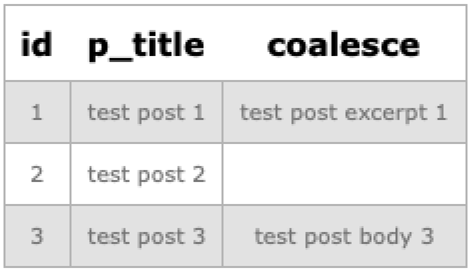
```
SELECT
      NULLIF (1, 1); -- return NULL

SELECT
      NULLIF (1, 0); -- return 1

SELECT
      NULLIF ('A', 'B'); -- return A
```

For example:

```
CREATE TABLE posts (
  id serial primary key,
      title VARCHAR (255) NOT NULL,
      excerpt VARCHAR (150),
      body TEXT,
      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
      updated_at TIMESTAMP
);
```

```
INSERT INTO posts (title, excerpt, body)
VALUES
      ('test post 1','test post excerpt 1','test post body 1'),
      ('test post 2','','test post body 2'),
      ('test post 3', null ,'test post body 3');
```

```
SELECT
      id,
      title,
      COALESCE (excerpt, LEFT(body, 40))
FROM
      posts;
```



id	p_title	coalesce
1	test post 1	test post excerpt 1
2	test post 2	
3	test post 3	test post body 3

CAST

There are many cases where you’d want to convert a value from one data type to another. PostgreSQL provides you with the CAST operator that allows you to do this.

Syntax:

```
CAST ( expression AS target_type );
```

- **Casting string to integer**

```
SELECT
      CAST ('100' AS INTEGER);
```

- **Casting a string to date**

```
SELECT
   CAST ('2015-01-01' AS DATE),
   CAST ('01-OCT-2015' AS DATE);
```

- **Casting string to double**

```
SELECT
      CAST ('10.2' AS DOUBLE);
```

- **Casting string to boolean**

```
SELECT
   CAST('true' AS BOOLEAN),
   CAST('false' as BOOLEAN),
   CAST('T' as BOOLEAN),
   CAST('F' as BOOLEAN);
```

- **Converting string to timestamp**

```
SELECT '2019-06-15 14:30:20'::timestamp;
```

- **Converting string to an interval**

```
SELECT '15 minute'::interval,
 '2 hour'::interval,
 '1 day'::interval,
 '2 week'::interval,
 '3 month'::interval;
```

Section 15 - Psql Commands Cheat Sheet

Here are the most commonly used Psql commands:

- **Connect to PostgreSQL**

```
psql -d database -U  user -W
```

- **Switch to a new database**

```
\c dbname username
```

- **List available databases**

```
\l
```

- **List available tables**

```
\dt
```

- **Describe table**

```
\d table_name
```

- **List available schema**

```
\dn
```

- **List available views**

```
\dv
```

- **List users and their roles**

```
\du
```

- **Execute previous commands**

```
SELECT version();
```

- **Command history**

```
\s
```

- **Execute psql from a file**

```
\i filename
```

- **Get help**

```
\?
```

- **Turn on query execution time**

```
dvdrental=# \timing
Timing is on.
dvdrental=# select count(*) from film;
 count
-------
  1000
(1 row)

Time: 1.495 ms
dvdrental=#
```

- **Quit plsql**

```
\q
```