PostgreSQL Functions

What is a Function in PostgreSQL?

- A **function** (or stored procedure) is a reusable block of code stored on the database server.
- It can contain **SQL statements + procedural code** (loops, conditions, variables).
- Functions save time because instead of running multiple queries, you call the function once.
- PostgreSQL supports functions in many languages: **SQL, PL/pgSQL, C, Python**, etc.

Note :Create a new database this work

CREATE DATABASE mydb;

CREATE FUNCTION Command

Syntax

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype
LANGUAGE plpgsql
AS $$
DECLARE
  -- variable declarations
BEGIN
  -- function logic
  RETURN value;
END;
$$;
```

Explanation of parts:

- **function_name** → Name of function.
- **OR REPLACE** → Updates existing function.
- **arguments** → Input parameters (can be none or many).
- **RETURNS** → The data type returned by the function.
- **LANGUAGE** → Programming language (e.g., plpgsql).
- **DECLARE** → Section to declare variables.
- **BEGIN … END** → Main logic of the function.
- **RETURN** → Final result of function.

Example – Function on Car Table

Suppose we have a table **Car** with a column **Car_price**.

We create a function to count cars within a price range:

```
CREATE FUNCTION get_car_Price(Price_from int, Price_to int)
RETURNS int
LANGUAGE plpgsql
AS $$
DECLARE
  Car_count integer;
BEGIN
  SELECT COUNT(*) INTO Car_count
  FROM Car
  WHERE Car_price BETWEEN Price_from AND Price_to;
```

```
  RETURN Car_count;
END;
$$;
```

How it works?

1. Function name → get_car_Price
2. Inputs → Price_from and Price_to
3. Inside function → it counts cars whose price is between given values.
4. Returns → number of cars (integer)

Structure of get_car_Price Function

1. **Header Section**
   - Function name → get_car_Price()
   - Parameters → Price_from INT, Price_to INT
   - Return type → INT
   - Language → plpgsql
2. **Function Body**
   - Written inside $$ … $$
   - **DECLARE** → variable Car_count
   - **BEGIN…END** →
     - SELECT INTO → counts cars between given price range
     - RETURN → returns Car_count

PostgreSQL – Creating Functions

1. Ways to Create a Function

- **pgAdmin** (GUI tool)
  - Open pgAdmin → connect DB → Query Tool → write function code → Execute → Refresh functions list.

- **SQL Shell (psql)** (command line)
  - Connect to DB: \c database_name
  - Write function code in shell.
  - List all functions: \df

Create Function in PostgreSQL using pgAdmin

1) Open pgAdmin

- Launch **pgAdmin 4** from your system.
- Log in with your PostgreSQL user (default: postgres).

2) Connect to the Server

- Expand **Servers** → **PostgreSQL 16 (or your version)**.
- Enter the password if asked.

3) Select / Create Database

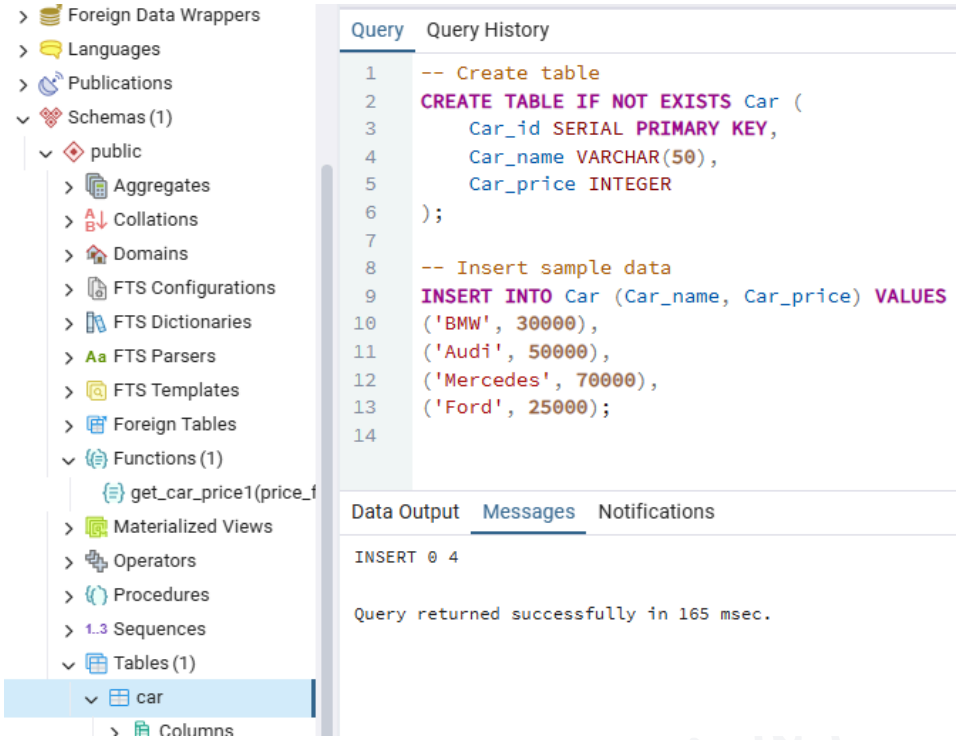
- Expand **Databases**.

- If you already created javatpoint (from earlier), right-click → **Connect**.
- If not:
    - Right-click **Databases** → **Create** → **Database...**
    - Enter name mydb→ **Save**.

4) Create Table Car (if not already created)

- Expand your database → **Schemas** → **public** → **Tables**.
- Right-click **Tables** → **Create** → **Table...**

```
-- Create table
CREATE TABLE IF NOT EXISTS Car (
    Car_id SERIAL PRIMARY KEY,
    Car_name VARCHAR(50),
    Car_price INTEGER
);
-- Insert sample data
INSERT INTO Car (Car_name, Car_price) VALUES
('BMW', 30000),
('Audi', 50000),
('Mercedes', 70000),
('Ford', 25000);
```



- Click **Save**.

5) Create the Function

- Expand **mydb**→ **Schemas** → **public** → **Functions**.
- Right-click **Functions** → **Create** → **Function...**

Now fill in:

General Tab

- Name: get_car_price1

Definition Tab

- Return type: integer
- Language: plpgsql

Code Tab

```
DECLARE
  Car_count INT;
BEGIN
  SELECT COUNT(*) INTO Car_count
  FROM Car
  WHERE Car_price BETWEEN Price_from AND Price_to;
  RETURN Car_count;
END;
```

Arguments Tab

Click **+** and add parameters:

1. Name: Price_from → Type: integer
2. Name: Price_to → Type: integer

Click **Save**

6) Verify Function

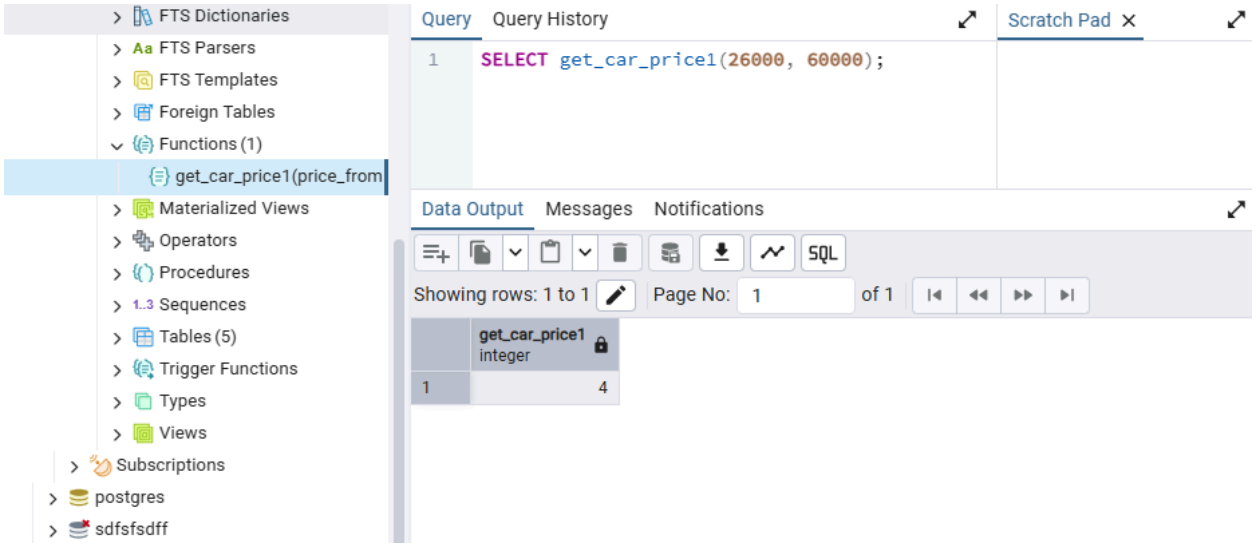
- Expand **Functions** → **get_car_price1** should now appear.
- Right-click → **Properties** to review.

7) Test the Function

- Open **Query Tool** (right-click DB → Query Tool).
- Run:

```
SELECT get_car_price1(26000, 60000);
```

Expected output:



8) Check Function List (Optional)

- In Query Tool:

\df

(or just see it under **Functions** tree in pgAdmin).

Create Function using SQL Shell (psql)

1) Open psql

Linux/macOS:

sudo -u postgres psql

or

psql -U postgres

Windows (SQL Shell):

we will open the psqlin our local system

(Enter the password if asked.)

2) (Optional) Create the database

If you need a fresh DB:

CREATE DATABASE mydb1;

Expected: CREATE DATABASE

3) Connect to the database

\c mydb1

```
mydb=# CREATE DATABASE mydb1;
CREATE DATABASE
mydb=# \c mydb1
You are now connected to database "mydb1" as user "postgres".
```

4) Create the sample table Car

```
CREATE TABLE Car1 (
  Car_id SERIAL PRIMARY KEY,
  Car_name VARCHAR(50),
  Car_price INT
);
```

```
mydb1=# CREATE TABLE Car1 (
mydb1(#   Car_id SERIAL PRIMARY KEY,
mydb1(#   Car_name VARCHAR(50),
mydb1(#   Car_price INT
mydb1(# );
CREATE TABLE
mydb1=#
```

5) Insert sample data

```
INSERT INTO Car1 (Car_name, Car_price) VALUES
('BMW', 30000),
('Audi', 50000),
('Mercedes', 70000),
('Ford', 25000);
```

```
        ^
mydb1=# INSERT INTO Car1 (Car_name, Car_price) VALUES
mydb1-# ('BMW', 30000),
mydb1-# ('Audi', 50000),
mydb1-# ('Mercedes', 70000),
mydb1-# ('Ford', 25000);
INSERT 0 4
mydb1=#
```

6) Verify data (optional)

```
SELECT * FROM Car1;
```

```
mydb1=# SELECT * FROM Car1;
 car_id | car_name | car_price
--------+----------+-----------
      1 | BMW      |     30000
      2 | Audi     |     50000
      3 | Mercedes |     70000
      4 | Ford     |     25000
(4 rows)


mydb1=# |
```

7) Create the function (corrected & safe)

```
CREATE OR REPLACE FUNCTION get_car_price1(Price_from INT, Price_to INT)

RETURNS INT

LANGUAGE plpgsql

AS $$

DECLARE

  Car_count INT;

BEGIN

  SELECT COUNT(*) INTO Car_count

  FROM Car1

  WHERE Car_price BETWEEN Price_from AND Price_to;



  RETURN Car_count;

END;

$$;
```

```
mydb1=# CREATE OR REPLACE FUNCTION get_car_price1(Price_from INT, Price_to INT)
mydb1-# RETURNS INT
mydb1-# LANGUAGE plpgsql
mydb1-# AS $$
mydb1$# DECLARE
mydb1$#   Car_count INT;
mydb1$# BEGIN
mydb1$#   SELECT COUNT(*) INTO Car_count
mydb1$#   FROM Car1
mydb1$#   WHERE Car_price BETWEEN Price_from AND Price_to;
mydb1$#
mydb1$#   RETURN Car_count;
mydb1$# END;
mydb1$# $$;
CREATE FUNCTION
mydb1=#
```

Notes:

- Use CREATE OR REPLACE to avoid "already exists" errors.
- Use $$ ... $$ for the body.
- Ensure you RETURN Car_count; (not a typo like Price_count).

8) List / inspect the function

\df get_car_price1

-- or for more details:

\df+ get_car_price1

Shows function name, argument types, return type, language, owner, source.

```
mydb1=# \df get_car_price1
                                   List of functions
 Schema |      Name      | Result data type |         Argument data types          | Type
--------+----------------+------------------+--------------------------------------+------
 public | get_car_price1 | integer          | price_from integer, price_to integer | func
(1 row)


mydb1=#
```

9) Call / test the function

SELECT get_car_price1(26000, 60000);

```
mydb1=# SELECT get_car_price1(26000, 60000);
 get_car_price1
----------------
              2
(1 row)


mydb1=#
```

(30000 and 50000 fall inside the range → 2)

Named:

SELECT get_car_price1(Price_from => 26000, Price_to => 60000);

Mixed (positional first):

SELECT get_car_price1(26000, Price_to => 60000);

10) Common fixes & helpful commands

If function exists and you want to replace it:

CREATE OR REPLACE FUNCTION ...

To remove:

DROP FUNCTION IF EXISTS get_car_price1(INT, INT);

If relation "Car" does not exist → check \dt and ensure you are in the correct database/schema.

If you get syntax errors → check AS $$ ... $$; and semicolons.

If permission denied → run as a superuser or grant appropriate rights.

11) Exit psql

\q

PostgreSQL Date & Time Functions

PostgreSQL provides a rich set of functions to work with **dates, times, intervals, and timestamps**.

1. AGE() Function

The AGE() function calculates the difference between two dates/timestamps.

Syntax

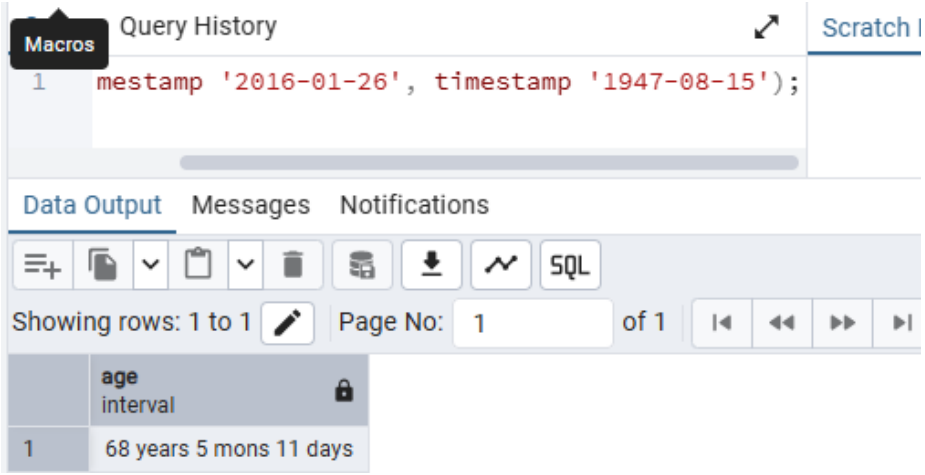
AGE(timestamp, timestamp)
AGE(timestamp)

Variants

- **AGE(timestamp1, timestamp2)** → Returns an interval showing the difference between the two timestamps in **years, months, days**.
- **AGE(timestamp)** → Returns the interval between the given timestamp and the **current date**.

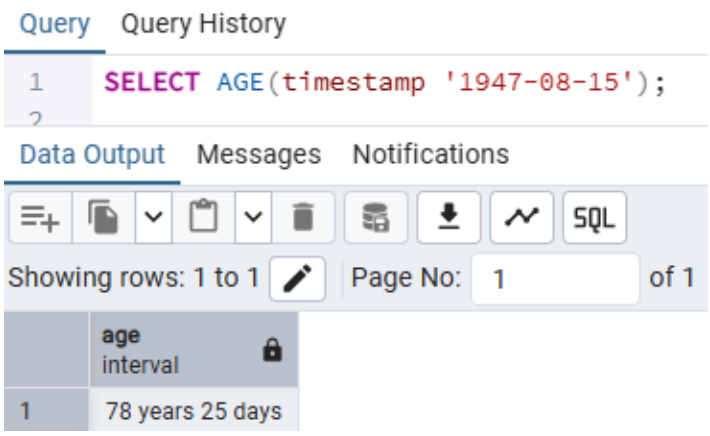
Examples

```
-- Difference between two timestamps
SELECT AGE(timestamp '2016-01-26', timestamp '1947-08-15');
```




```
-- Age from current date
```

SELECT AGE(timestamp '1947-08-15');



2. CURRENT Date/Time Functions

These functions return the **current system date and time**.

Function	Description
CURRENT_DATE	Returns current date.
CURRENT_TIME	Returns current time with time zone.
CURRENT_TIMESTAMP	Returns current date & time with time zone.
CURRENT_TIME(precision)	Current time rounded to given fractional seconds.
CURRENT_TIMESTAMP(precision)	Current timestamp rounded to given fractional seconds.
LOCALTIME	Returns current time without time zone.
LOCALTIMESTAMP	Returns current timestamp without time zone.
LOCALTIME(precision)	Local time rounded to given fractional seconds.
LOCALTIMESTAMP(precision)	Local timestamp rounded to given fractional seconds.

Examples

Current date

SELECT CURRENT_DATE;

Query Query History

```
SELECT CURRENT_DATE;
```

Data Output Messages Notifications

Showing rows: 1 to 1 Page No: 1 of 1

	current_date date
1	2025-09-04

Current time (with time zone)

SELECT CURRENT_TIME;

Query Query History

```
SELECT CURRENT_TIME;
```

Data Output Messages Notifications

Showing rows: 1 to 1 Page No: 1 of 1

	current_time time with time zone
1	13:02:24.985931+05:30

Current timestamp (date + time)

SELECT CURRENT_TIMESTAMP;

Query Query History

```
SELECT CURRENT_TIMESTAMP;
```

Data Output Messages Notifications

Showing rows: 1 to 1 Page No: 1 o

	current_timestamp timestamp with time zone
1	2025-09-04 13:02:58.842205+05:30

Current timestamp with precision

Default precision = 6 (microseconds).

SELECT CURRENT_TIMESTAMP(2);

Query Query History

```
1  SELECT CURRENT_TIMESTAMP(2);
2
```

Data Output Messages Notifications

Showing rows: 1 to 1 Page No: 1

	current_timestamp timestamp with time zone (2)
1	2025-09-04 13:03:28.68+05:30

Local timestamp (no time zone)

SELECT LOCALTIMESTAMP;

Query Query History

```
1  SELECT LOCALTIMESTAMP;
2
```

Data Output Messages Notifications

Showing rows: 1 to 1 Page No: 1 of 1

	localtimestamp timestamp without time zone
1	2025-09-04 13:05:29.963148

With these functions, you can easily:

- Calculate ages or durations (AGE())

- Get current system date & time (CURRENT_DATE, CURRENT_TIMESTAMP)
- Work with precision in time values
- Choose results **with** or **without** timezone

PostgreSQL Date & Time Functions – Practice Q&A

1. Find the difference between 2000-01-01 and 1990-01-01.
SELECT AGE(timestamp '2000-01-01', timestamp '1990-01-01');

2. Find the age from your birthday (e.g., 1995-05-20) till today.
SELECT AGE(timestamp '1995-05-20');

3. Display the current date.
SELECT CURRENT_DATE;

4. Display the current time with time zone.
SELECT CURRENT_TIME;

5. Display the current timestamp with 2-digit fractional precision.
SELECT CURRENT_TIMESTAMP(2);

6. Display the local timestamp (without time zone).
SELECT LOCALTIMESTAMP;

7. Extract the year from the current date.
SELECT DATE_PART('year', CURRENT_DATE);

8. Extract the month from the current timestamp.
SELECT EXTRACT(MONTH FROM CURRENT_TIMESTAMP);

9. Extract the day of week from the current timestamp.
SELECT EXTRACT(DOW FROM CURRENT_TIMESTAMP);

10. Test whether 'infinity'::date is finite.
SELECT ISFINITE('infinity'::date);

PostgreSQL – Common psql Commands

psql is the interactive terminal for PostgreSQL. It lets you run queries and special commands efficiently.

1. Connect to Database

psql -d database -U user -W

Connects to a database with a given user (asks for password).

Example:

psql -d akshay-U postgres -W

To connect to a remote host:

psql -h host -d database -U user -W

With SSL mode:

psql -U user -h host "dbname=db sslmode=require"

2. Switch to Another Database

\c database_name

Example:

\c akshay

```
postgres-# \c akshay
You are now connected to database "akshay" as user "postgres".
akshay-#
```

3. List Databases

\l

```
akshay-# \l
                                                    List of databases
    Name     |  Owner   | Encoding | Locale Provider |          Collate           |           Ctype            | Locale | ICU Rules |   Access privileges
-------------+----------+----------+-----------------+----------------------------+----------------------------+--------+-----------+-----------------------
 Origination | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 akshay      | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 asfdf       | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 bh          | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 deepa       | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 demo2       | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 left        | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 mydb        | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 mydb1       | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 postgres    | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 sdfsfsdff   | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           |
 template0   | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |        |           | =c/postgres          +
-- More  -- |
```

4. List Tables

\dt

```
akshay-# \dt
              List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | customer          | table | postgres
 public | employee          | table | postgres
 public | employee1         | table | postgres
 public | employee2         | table | postgres
 public | employee_backup   | table | postgres
 public | num_test1         | table | postgres
 public | product           | table | postgres
 public | sales             | table | postgres
 public | students          | table | postgres
(9 rows)


akshay-# |
```

5. Describe a Table

\d table_name

Example:

\d customer

```
akshay-# \d customer
                          Table "public.customer"
    Column     |          Type          | Collation | Nullable | Default
---------------+------------------------+-----------+----------+---------
 customer_id   | character varying(255) |           | not null |
 customer_name | character varying(255) |           |          |
 segment       | character varying(255) |           |          |
 age           | integer                |           |          |
 country       | character varying(255) |           |          |
 city          | character varying(255) |           |          |
 state         | character varying(255) |           |          |
 postal_code   | bigint                 |           |          |
 region        | character varying(255) |           |          |
 test          | character varying(255) |           |          |
Indexes:
    "Customer_pkey" PRIMARY KEY, btree (customer_id)


akshay-#
```

6. List Functions

\df

```
akshay-# \df
                          List of functions
 Schema | Name | Result data type | Argument data types | Type
--------+------+------------------+---------------------+------
(0 rows)


akshay-#
```

7. List Schemas

\dn

```
akshay-# \dn
      List of schemas
  Name  |       Owner
--------+-------------------
 public | pg_database_owner
(1 row)
```

8. List Users & Roles

\du

```
akshay-# \du
                             List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS


akshay-#
```

9. List Views

\dv

10. Show PostgreSQL Version

SELECT version();

Repeat the last command:

\g

11. Run Commands from a File

\i filename

12. Help on Commands

\?

```
akshay=# \?
General
  \bind [PARAM]...       set query parameters
  \copyright             show PostgreSQL usage and distribution terms
  \crosstabview [COLUMNS] execute query and display result in crosstab
  \errverbose            show most recent error message at maximum verbosity
  \g [(OPTIONS)] [FILE]  execute query (and send result to file or |pipe);
                         \g with no arguments is equivalent to a semicolon
  \gdesc                 describe result of query, without executing it
  \gexec                 execute query, then execute each value in its result
  \gset [PREFIX]         execute query and store result in psql variables
  \gx [(OPTIONS)] [FILE] as \g, but forces expanded output mode
  \q                     quit psql
  \watch [[i=]SEC] [c=N] [m=MIN]
                         execute query every SEC seconds, up to N times,
                         stop if less than MIN rows are returned

Help
  \? [commands]          show help on backslash commands
  \? options             show help on psql command-line options
  \? variables           show help on special variables
  \h [NAME]              help on syntax of SQL commands, * for all commands

Query Buffer
  \e [FILE] [LINE]       edit the query buffer (or file) with external editor
  \ef [FUNCNAME [LINE]]  edit function definition with external editor
  \ev [VIEWNAME [LINE]]  edit view definition with external editor
  \p                     show the contents of the query buffer
```

Shows all psql meta-commands

\h

```
akshay=# \h
Available help:
  ABORT                            CREATE USER MAPPING
  ALTER AGGREGATE                  CREATE VIEW
  ALTER COLLATION                  DEALLOCATE
  ALTER CONVERSION                 DECLARE
  ALTER DATABASE                   DELETE
  ALTER DEFAULT PRIVILEGES         DISCARD
  ALTER DOMAIN                     DO
  ALTER EVENT TRIGGER              DROP ACCESS METHOD
  ALTER EXTENSION                  DROP AGGREGATE
  ALTER FOREIGN DATA WRAPPER       DROP CAST
  ALTER FOREIGN TABLE              DROP COLLATION
```

Shows help on SQL commands

Example:

\h ALTER TABLE

```
akshay=# \h ALTER TABLE
Command:     ALTER TABLE
Description: change the definition of a table
Syntax:
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

13. Show Query Execution Time

\timing

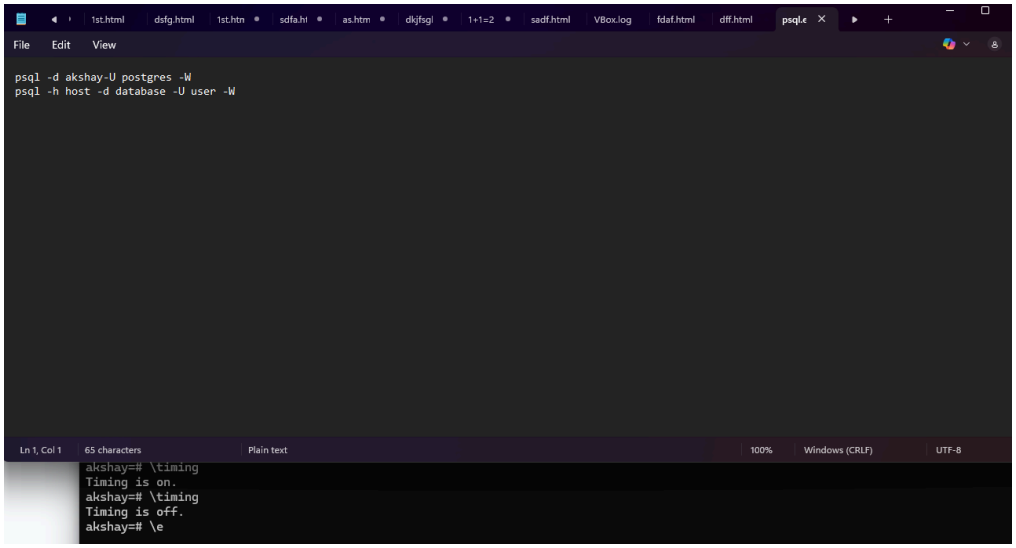
```
akshay=# \timing
Timing is on.
akshay=#
```

Run again to turn it off.

```
akshay=# \timing
Timing is off.
akshay=#
```


14. Edit Query in Editor

\e



Opens system editor, write query, save & close → query executes automatically.

15. Quit psql

\q

Summary Table

Command	Purpose
\c dbname	Connect to another database
\l	List all databases
\dt	List all tables
\d table	Describe a table

\df	List functions
\dn	List schemas
\du	List users & roles
\dv	List views
SELECT version();	Show PostgreSQL version
\g	Run previous command
\i filename	Run commands from a file
\?	Help on psql commands
\h	Help on SQL commands
\timing	Show execution time
\e	Edit query in editor
\q	Quit psql

PostgreSQL UNION Operator

1. What is the UNION operator?

The **UNION** operator in PostgreSQL is used to **combine results** from two or more SELECT queries into a single result set.

- It removes **duplicate rows** by default.
- All SELECT queries must have:
  - The **same number of columns**
  - Columns with **compatible data types**

Syntax

```
SELECT select_list_1
FROM table_expression_1
UNION
SELECT select_list_2
FROM table_expression_2;
```


Conditions:

- Number & order of columns must match.
- Data types must be compatible.

Example Tables

```
-- Top Rated Cars

CREATE TABLE top_rated_cars(
    Car_name VARCHAR NOT NULL,
    launch_year SMALLINT
);
```

```
INSERT INTO top_rated_cars VALUES
('Chevrolet Silveradon',2020),
('Nissan Rogue',2020),
('Mercedes-Benz GLB',2019);

-- Most Reliable Cars

CREATE TABLE most_reliable_cars(
    Car_name VARCHAR NOT NULL,
    launch_year SMALLINT
);

INSERT INTO most_reliable_cars VALUES
('Toyota Prius Prime',2020),
('Nissan Rogue',2020),
('Kia Forte',2019);
```


Simple UNION Example

```
SELECT * FROM top_rated_cars
UNION
SELECT * FROM most_reliable_cars;
```

Query | Query History

```
SELECT * FROM top_rated_cars
UNION
SELECT * FROM most_reliable_cars;
```

Data Output | Messages | Notifications

	car_name character varying	launch_year smallint
1	Chevrolet Silveradon	2020
2	Mercedes-Benz GLB	2019
3	Nissan Rogue	2020
4	Toyota Prius Prime	2020
5	Kia Forte	2019

Removes duplicate rows → only **unique cars** are shown.

UNION ALL

UNION ALL keeps duplicates.

```
SELECT * FROM top_rated_cars
UNION ALL
SELECT * FROM most_reliable_cars;
```

Query Query History

```
SELECT * FROM top_rated_cars
UNION ALL
SELECT * FROM most_reliable_cars;
```

Data Output Messages Notifications

	car_name character varying	launch_year smallint
1	Chevrolet Silveradon	2020
2	Nissan Rogue	2020
3	Mercedes-Benz GLB	2019
4	Toyota Prius Prime	2020
5	Nissan Rogue	2020
6	Kia Forte	2019

Both Nissan Rogue entries appear.

UNION / UNION ALL with ORDER BY

ORDER BY is applied **at the end of the last query**.

```
SELECT * FROM top_rated_cars
UNION ALL
SELECT * FROM most_reliable_cars
ORDER BY Car_name;
```

Query Query History

```
SELECT * FROM top_rated_cars
UNION ALL
SELECT * FROM most_reliable_cars
ORDER BY Car_name;
```

Data Output Messages Notifications

	car_name character varying	launch_year smallint
1	Chevrolet Silveradon	2020
2	Kia Forte	2019
3	Mercedes-Benz GLB	2019
4	Nissan Rogue	2020
5	Nissan Rogue	2020
6	Toyota Prius Prime	2020

If you use ORDER BY inside each query, final result may not be sorted.

Key Points

- **UNION** → removes duplicates.
- **UNION ALL** → keeps duplicates.
- ORDER BY must be placed **after the final query**.
- Useful to merge similar datasets from multiple tables.

PostgreSQL UNION & UNION ALL Practice Questions

1.You have two tables:

- students_2024 (name, course)
- students_2025 (name, course)

Write a query to combine all students into one list **without duplicates**.

2. Using the same tables (students_2024, students_2025), write a query to combine all students into one list **including duplicates**.

3. From the tables top_rated_cars and most_reliable_cars, write a query to display all cars from both tables, sorted by launch_year.

4. Which operator will you use if you want to:

    - a) Eliminate duplicate rows

    - b) Keep duplicate rows

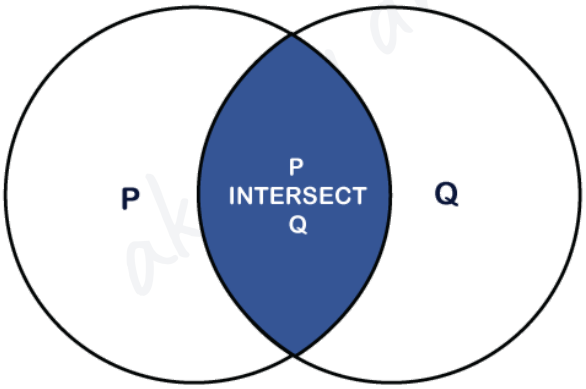
5. Suppose tableA and tableB both have columns (id, city).
   Write a query to merge both tables into a single list but ensure the result is sorted alphabetically by city.

PostgreSQL INTERSECT Operator

What is INTERSECT?

- The **INTERSECT** operator in PostgreSQL retrieves the **common rows** from two or more result sets.
- Works similar to UNION and EXCEPT, but only keeps rows that **exist in both queries**.
- Each SELECT inside an INTERSECT must return the **same number of columns** with **compatible data types**.

Visualization:
If dataset A = circle P and dataset B = circle Q, then INTERSECT = **overlapping area**.



Syntax

```
SELECT expression1, expression2, ... expression_n
FROM table1
[WHERE conditions]
INTERSECT
SELECT expression1, expression2, ... expression_n
FROM table2
[WHERE conditions];
```

Parameters

Parameter	Description
expression1, expression2, ...	Columns or calculations to retrieve
tables	Source tables
WHERE conditions	Optional filters

Rules:

1. Both queries must have the **same number of columns**.
2. Corresponding columns must have **same or compatible data types**.

Create Tables

```
CREATE TABLE top_rated_cars (
    car_id SERIAL PRIMARY KEY,
    car_name VARCHAR(50),
    launch_year INT,
    rating INT
);


CREATE TABLE most_reliable_cars (
    car_id SERIAL PRIMARY KEY,
    car_name VARCHAR(50),
    launch_year INT,
    reliability_score INT
);
```

-- Data for top_rated_cars

```
INSERT INTO top_rated_cars (car_name, launch_year, rating) VALUES
('BMW X5', 2020, 9),
```

```
('Audi Q7', 2019, 8),
('Mercedes GLC', 2021, 9),
('Toyota Corolla', 2018, 7),
('Honda Civic', 2020, 8),
('Ford Mustang', 2022, 9);
```

-- Data for most_reliable_cars

```
INSERT INTO most_reliable_cars (car_name, launch_year, reliability_score) VALUES
('Toyota Corolla', 2018, 9),
('Honda Civic', 2020, 8),
('Hyundai Elantra', 2019, 7),
('Mercedes GLC', 2021, 9),
('Kia Seltos', 2022, 8),
('BMW X5', 2020, 8);
```

-- Drop table if it already exists

```
DROP TABLE IF EXISTS employee;
```

-- Create employee table

```
CREATE TABLE employee (
    employee_id SERIAL PRIMARY KEY,
    employee_name VARCHAR(50) NOT NULL,
    address VARCHAR(100) NOT NULL,
    phone VARCHAR(20) NOT NULL
);
```

-- Insert sample employees

```
INSERT INTO employee (employee_name, address, phone) VALUES
('John',   'New York', '111-111'),
('Ross',   'London',   '222-222'),
```

```
('Monica', 'Paris',    '333-333'),

('Rachel', 'Tokyo',    '444-444');

-- Drop table if it already exists

DROP TABLE IF EXISTS department;

-- Create department table

CREATE TABLE department (

    dept_id SERIAL PRIMARY KEY,

    dept_name VARCHAR(50) NOT NULL,

    address VARCHAR(100) NOT NULL,

    phone VARCHAR(20) NOT NULL

);

-- Insert sample departments

INSERT INTO department (dept_name, address, phone) VALUES

('HR',        'Paris',     '333-333'),

('Finance',   'New York',  '111-111'),

('IT',        'Berlin',    '555-555'),

('Marketing', 'London',    '666-666');
```


Examples

1. INTERSECT with Single Column

```
SELECT Car_name
FROM top_rated_cars
INTERSECT
SELECT Car_name
FROM most_reliable_cars;
```

Query Query History

```
SELECT Car_name
FROM top_rated_cars
INTERSECT
SELECT Car_name
FROM most_reliable_cars;
```

Data Output Messages Notifications

	car_name character varying (50)
1	BMW X5
2	Honda Civic
3	Toyota Corolla
4	Mercedes GLC

Returns cars that exist in **both tables**.

2. INTERSECT with WHERE Clause

SELECT Car_name
FROM top_rated_cars
WHERE rating > 4.5
INTERSECT
SELECT Car_name
FROM most_reliable_cars
WHERE reliability_score >= 9;

Query Query History

```
SELECT Car_name
FROM top_rated_cars
WHERE rating > 4.5
INTERSECT
SELECT Car_name
FROM most_reliable_cars
WHERE reliability_score >= 9;
```

Data Output Messages Notifications

Showing rows: 1 to 2 Page No: 1 of 1

	car_name character varying (50)
1	Toyota Corolla
2	Mercedes GLC

Returns cars present in both sets **after filtering conditions**.

3. INTERSECT with Multiple Columns

```
SELECT address, phone
FROM employee
WHERE employee_name <> 'ross'
INTERSECT
SELECT address, phone
FROM department
WHERE address <> 'London';
```

Query Query History

```
1  SELECT address, phone
2  FROM employee
3  WHERE employee_name <> 'ross'
4  INTERSECT
5  SELECT address, phone
6  FROM department
7  WHERE address <> 'London';
```

Data Output Messages Notifications

	address character varying (100)	phone character varying (20)
1	Paris	333-333
2	New York	111-111

Returns rows where **address & phone match** in both tables.

4. INTERSECT with ORDER BY

```
SELECT address, phone
FROM employee
WHERE employee_name <> 'ross'
INTERSECT
SELECT address, phone
FROM department
WHERE address <> 'London'
ORDER BY 1;
```

Query Query History

```sql
SELECT address, phone
FROM employee
WHERE employee_name <> 'ross'
INTERSECT
SELECT address, phone
FROM department
WHERE address <> 'London'
ORDER BY 1;
```

Data Output Messages Notifications

	address character varying (100)	phone character varying (20)
1	New York	111-111
2	Paris	333-333

Sorts final output by the **first column (address)**.

Overview / Key Points

- **INTERSECT** = common rows between multiple queries.
- Can be used with WHERE filters.
- Can include **multiple columns**.
- Supports **ORDER BY** at the final stage.

Practice Questions on PostgreSQL INTERSECT

1. Create two tables students_2024 and students_2025 with columns (student_name, course).
    - Insert some students into both tables.
    - Write a query to find the **students enrolled in both years**.
2. Using the tables top_rated_cars and most_reliable_cars:
    - Write a query to find the **car names** that appear in **both tables**.

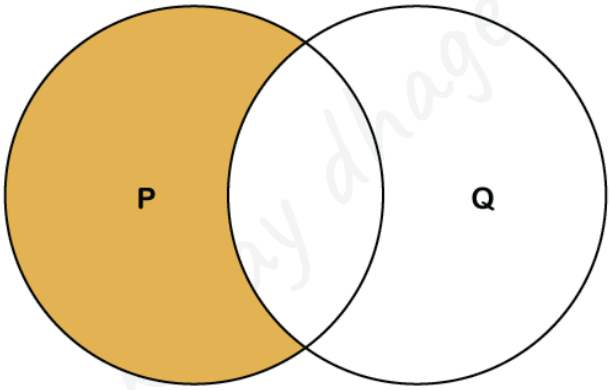
3. From the employee and department tables:
    - Write a query using INTERSECT to display **addresses and phone numbers** that are common in both tables.
4. Modify Question 2 by adding a condition:
    - Find cars that are **in both tables**, but only where launch_year = 2020.
5. Use INTERSECT with ORDER BY:
    - Get the list of car names present in both top_rated_cars and most_reliable_cars, and **order the result alphabetically**.

PostgreSQL EXCEPT Operator

What is PostgreSQL EXCEPT?

- The **EXCEPT** operator returns rows from the **first SELECT query** that **do not exist** in the result of the second SELECT query.
- Works like **UNION** and **INTERSECT**, but instead of merging or finding common records, it shows **differences**.

Think of it as:
Result = Query1 – Query2



Key Rules

1. Both SELECT queries must have the **same number of columns**.
2. The columns must have **compatible data types**.
3. Removes **duplicates automatically** (just like UNION).

Syntax

```
SELECT column1, column2, ...
FROM table1
[WHERE conditions]
EXCEPT
```

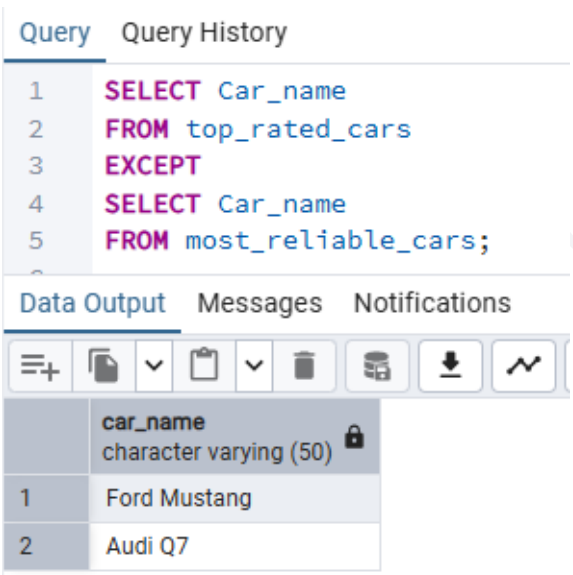
```
SELECT column1, column2, ...
FROM table2
[WHERE conditions];
```

Examples

1. EXCEPT with Single Column

Find cars that are in top_rated_cars but not in most_reliable_cars:

```
SELECT Car_name
FROM top_rated_cars
EXCEPT
SELECT Car_name
FROM most_reliable_cars;
```



Output → Only car names present in top_rated_cars but missing in most_reliable_cars.

2. EXCEPT with Multiple Columns

Compare two tables (employee and department) using multiple fields:

```
SELECT address, phone
FROM employee
WHERE employee_name <> 'ross'
```

```
EXCEPT
SELECT address, phone
FROM department
WHERE address <> 'London';
```

Query Query History

```
1  SELECT address, phone
2  FROM employee
3  WHERE employee_name <> 'ross'
4  EXCEPT
5  SELECT address, phone
6  FROM department
7  WHERE address <> 'London';
```

Data Output Messages Notifications

	address character varying (100)	phone character varying (20)
1	London	222-222
2	Tokyo	444-444

Output → Address & phone combinations that exist in employee but not in department.

3. EXCEPT with ORDER BY

Sort the result of EXCEPT:

```
SELECT *
FROM top_rated_cars
EXCEPT
SELECT *
FROM most_reliable_cars
ORDER BY Car_name;
```

Query Query History

```
1  SELECT *
2  FROM top_rated_cars
3  EXCEPT
4  SELECT *
5  FROM most_reliable_cars
6  ORDER BY Car_name;
```

Data Output Messages Notifications

	car_id integer	car_name character varying (50)	launch_year integer	rating integer
1	2	Audi Q7	2019	8
2	1	BMW X5	2020	9
3	6	Ford Mustang	2022	9
4	5	Honda Civic	2020	8
5	3	Mercedes GLC	2021	9
6	4	Toyota Corolla	2018	7

Output → Cars only in top_rated_cars, sorted by name.

Visual Understanding

- Circle **P** = First SELECT query.
- Circle **Q** = Second SELECT query.
- **P − Q** = Records only in P (not in Q).

Overview

Returns rows from the **first query** that are not in the **second query**.
Can be used with **single or multiple columns**.
Supports **ORDER BY** to sort results.
Useful to find **differences between datasets**.

Practice Questions on PostgreSQL EXCEPT

Q1. Find students who are not alumni.
Q2. Get cars that are in top_rated_cars but not in most_reliable_cars.
Q3. Find orders from 2024 but not in 2025.
Q4. Get employees who are not in ex_employee.
Q5. List products in India but not in the USA (sorted).

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:

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