# History of PostgreSQL

PostgreSQL is currently the most advanced open-source database available in the world. It was started in 1986 to overcome the problems of the contemporary database system. It was created by a computer science professor named **Michael Stonebraker** at UCB and was originally called Postgres, as it was started as a follow-up project and a post-Ingres project.

# Features of PostgreSQL

* + Runs on all major operating systems including Linux, UNIX and Windows.
  + Supports text, images, sounds, and video.
  + Includes programming interfaces for C / C++, Java, Perl, Python, Ruby, Tcl and Open Database Connectivity (ODBC).
  + Supports many features of SQL including some advanced features.
  + A table in PostgreSQL can be set to inherit their characteristics from a “parent” table.
  + To add additional functionality to PostgreSQL several extensions can be installed.

# Tools for PostgreSQL

* **Psql**

Command line tool.

Primary tool to manage PostgreSQL.

* **pgAdmin**

Free and open source graphical user interface administration tool for PostgreSQL.

# Install postgreSQL

* First, execute the following command to create the file repository configuration:

$ sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb\_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'

* Second, import the repository signing key:

$ wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add –

* Third, update the package list:

$ sudo apt-get update

* Finally, install the latest version of PostgreSQL:

$ sudo apt-get install postgresql

# Connect to the PostgreSQL

* To connect to PostgreSQL using the postgres role, you switch over to the postgres account on your server by typing:

$ sudo -i -u postgres

* Then, you can access the PostgreSQL using the psql by typing the following command:

$ psql

* To quit the PostgreSQL prompt, you run the following command:

postgres=# \q

* To return to your regular system user, you execute the exit command like this:

postgres@ubuntu-dev:~$ exit

# Create database in PostgreSQL

* **Syntax :**

CREATE DATABASE name;

**name:** It is used to specify the name of the database to be created.

* **Example :**

CREATE DATABASE students;

A new database will be created with the name ‘students’.

# Drop database in PostgreSQL

* **Syntax :**

DROP DATABASE name;

**name**: It is used to specify the name of the database to be eliminated.

* **Example :**

DROP DATABASE country;

The ‘country’ database will be removed.

# Create table in PostgreSQL

* + **Syntax :**

CREATE TABLE table\_name (

column\_1 data\_type column\_constraint,

column\_2 data\_type column\_constraint,

... );

* **Example :**

CREATE TABLE employment

( id INT NOT **NULL** AUTO\_INCREMENT,

state VARCHAR(100) NOT **NULL**,

rate REAL );

* **Explanation**

**Column 1:**  
id: Name of Column 1.  
INT: Datatype of the Column 1 which is Integer.  
NOT NULL: It cannot contain null values.

**Column 2:**  
state: Name of Column 2.  
VARCHAR: Datatype of Column 2 which also specifies a maximum limit of 100 characters in length for the “state”.  
NOT NULL: It cannot contain null values.

**Column 3:**  
rate: Name of Column 3.  
REAL: Datatype of Column 3.  
NULL: It is NULL by default and thus can contain null values.

# Drop table in PostgreSQL

* **Syntax :**

DROP TABLE name;

**name:** It is used to specify the name of the TABLE to be dropped.

* **Example :**

DROP TABLE employment;

The ‘employment’ TABLE will be removed.

# Insert query in PostgreSQL

* **Syntax :**

INSERT into table\_name(column\_1, column\_2, ... column\_n )

VALUES (value\_1, value\_2, .. value\_n);

* **Example :**

INSERT INTO “EMPLOYMENT”

(“ID”, “STATE”, “RATE”)

VALUES

(3, 'C’, 65);

* **Explanation:**

The EMPLOYMENT is an existing table in which we are adding a new row with corresponding values under the respective columns.

# Select query in PostgreSQL

* **Syntax 1: To select all fields from a table.**

SELECT \* FROM “table\_name”;

* **Example 1: Selecting all fields from a table.**

SELECT \* from “EMPLOYMENT”;

* **Syntax 2: To select specific fields from a table.**

SELECT “expressions”

FROM “table\_name”

* **Example 2: Selecting specific fields from a table.**

SELECT “STATE”, “RATE”

FROM “EMPLOYMENT”;

# Update query in PostgreSQL

* **Syntax :**

UPDATE table\_name

SET column\_1 = expr\_1,

column\_2 = expr\_2,

...

column\_n = expr\_n

WHERE conditions;

* **Example :**

UPDATE “EMPLOYMENT”

SET “RATE” = 85

WHERE “ID” = 3;

* **Explanation:**

The RATE will be updated to 85 where ID is 3.

# Delete query in PostgreSQL

To remove or delete existing records from a table, the PostgreSQL DELETE statement is used.

* **Syntax: To delete all rows from a table.**

DELETE FROM table\_name;

* **Syntax: To delete specific rows from a table.**

DELETE FROM table\_name

WHERE conditions;

* **Example 1: Deleting all rows from a table.**

DELETE FROM “EMPLOYMENT”;

* **Explanation:**

The EMPLOYMENT is an already existing table, from which we are deleting all the existing records but not the table structure.

* **Example 2: Deleting single row from a table.**

DELETE FROM “EMPLOYMENT”

WHERE “ID” = 3;

* **Explanation:**

The EMPLOYMENT is an already existing table, from which we are deleting the row where the value of ID is 3.

# Where clause in PostgreSQL

The PostgreSQL WHERE clause is used with SELECT, INSERT, UPDATE and DELETE statements to return the result only when the condition is satisfied.

* **Syntax:**

WHERE conditions;

* **Example 1: Selecting specific fields from a table.**

SELECT \*

FROM “EMPLOYMENT”

WHERE “RATE” > 60;

* **Example 2: Using WHERE clause with AND condition.**

SELECT \*

FROM “EMPLOYMENT”

WHERE “RATE” > 60

AND “ID” < 4;

* **Example 3: Using WHERE clause with OR condition.**

SELECT \*

FROM “EMPLOYMENT”

WHERE “RATE” > 60

OR “ID” < 4;

* **Example 4: Using WHERE clause with both AND and OR condition.**

SELECT \*

FROM “EMPLOYMENT”

WHERE (“RATE” > 60 AND “STATE” = ‘C’)

OR “ID” > 4;

# Order by in PostgreSQL

PostgreSQL ORDER BY clause is used to sort or re-arrange the records in the result set. It is used with the PostgreSQL SELECT statement. It however does not have a mandatory existence with the PostgreSQL SELECT statement.

* **Syntax:**

SELECT expressions

FROM table\_name

WHERE conditions;

ORDER BY expression [ ASC | DESC ];

* **Example :**

SELECT \*

FROM “EMPLOYMENT”

WHERE “ID” > 2

ORDER BY “RATE”;

# Group by in PostgreSQL

PostgreSQL GROUP BY clause is used to collect data from multiple records and then to group the identical or similar results. It is used with the PostgreSQL SELECT statement. It, however, does not have a mandatory existence with the PostgreSQL SELECT statement.

* **Syntax:**

SELECT expressions

FROM table\_name

GROUP BY columns;

* **Example 1 :**

SELECT “RATE”

FROM “EMPLOYMENT”

GROUP BY “RATE”

ORDER BY “RATE”;

* **Explanation:**

The EMPLOYMENT is an already existing table from which we are retrieving unique employment rates using the GROUP BY clause.

* **Example 2:**

SELECT “RATE”, COUNT (“RATE”) AS “TOTAL COUNT”

FROM “EMPLOYMENT”

GROUP BY “RATE”;

* **Explanation:**

The EMPLOYMENT is an already existing table from which we are retrieving the count of the occurrence of each unique employment rates using the GROUP BY clause.

# Having in PostgreSQL

PostgreSQL HAVING clause is used to return the groups of rows only when the condition is TRUE. It is used with the PostgreSQL GROUP BY Clause. It, however, does not have a mandatory existence with the PostgreSQL GROUP BY Clause.

* **Syntax:**

SELECT expressions

FROM table\_name

GROUP BY columns

HAVING having\_conditions;

* **Example:**

SELECT “RATE”

FROM “EMPLOYMENT”

GROUP BY “RATE”

HAVING COUNT (“RATE”) &gt; 1

ORDER BY “RATE”;

* **Explanation:**

The EMPLOYMENT is an already existing table from which we are retrieving unique employment rates with count greater than 1 using the HAVING condition with the GROUP BY clause.

# Conditions in PostgreSQL

Conditions in PostgreSQL are generally used with CRUD operations to retrieve more specific results. They are usually used with the WHERE clause. Below is the list of conditions facilitated by the PostgreSQL database.

|  |  |
| --- | --- |
| **CONDITION** | **USES** |
| AND | When 2 or more conditions to be met. |
| OR | When any one of the conditions are met. |
| AND & OR | When AND & OR both conditions are met. |
| LIKE | Simple pattern matching (wildcards) in a where clause. |
| IN | As an alternative to multiple OR conditions. |
| NOT IN | To negate the IN conditions. |
| NOT | To negate a condition. |
| BETWEEN | To retrieve within a range (inclusive). |

# And condition in PostgreSQL

To filter the results, the PostgreSQL AND condition is used with SELECT, INSERT, UPDATE and DELETE statements to test more than one conditions.

* **Syntax:**

WHERE condition\_1

AND condition\_2

AND condition\_3

….

AND condition\_n;

* **Example:**

SELECT \*

FROM “STUDENTS”

WHERE “AGE” > 10

AND “AGE” < 15

ORDER BY “NAME”;

# NOT condition in PostgreSQL

* **Syntax:**

NOT condition;

* **Example: Using NOT with LIKE condition.**

SELECT “AGE”

FROM “STUDENTS”

WHERE “NAME” NOT LIKE 'Dav%';

# Like condition in PostgreSQL

To filter the results, the PostgreSQL LIKE condition is used with a combination of WHERE Clause in SELECT, INSERT, UPDATE and DELETE statements to perform pattern matching.

* **Example:**

SELECT “AGE”

FROM “STUDENTS”

WHERE “NAME” LIKE 'Dav%';

# IN condition in PostgreSQL

To filter the results, the PostgreSQL IN condition is used with SELECT, INSERT, UPDATE and DELETE statements to replace the use of multiple OR conditions.

* **Syntax:**

WHERE expressions IN (value1, value2, .... value\_n);

* **Example:**

SELECT \*

FROM “STUDENTS”

WHERE “NAME” IN (‘Joy’, 'Happy', 'Smiley')

ORDER BY “NAME”;

# NOT IN condition in PostgreSQL

To filter the results, the PostgreSQL NOT IN condition is used with SELECT, INSERT, UPDATE and DELETE statements to negate the IN conditions.

* **Syntax:**

WHERE expressions NOT IN (value1, value2, .... value\_n);

* **Example:**

SELECT \*

FROM “STUDENTS”

WHERE “NAME” NOT IN (‘Joy’, 'Happy', 'Smiley');

# BETWEEN condition in PostgreSQL

To filter the results, the PostgreSQL BETWEEN condition is used with SELECT, INSERT, UPDATE and DELETE statements to get values within a specific range from an expression.

* **Syntax:**

WHERE expression BETWEEN value1 AND value2;

* **Example:**

SELECT “NAME”, “AGE”

FROM “STUDENTS”

WHERE “ID” BETWEEN 2 AND 6;

# JOINS in PostgreSQL

To create a new table with the records as a combination of records from multiple tables, the PostgreSQL Join Query is used.

## **Types of Joins:**

* + Inner or Simple Join
  + Left Outer Join
  + Right Outer Join
  + Full Outer Join
  + Cross Join

# INNER JOIN

The INNER Join being the most popular and common join is often called a SIMPLE Join as it returns all the rows from multiple tables where the join condition is met.

* **Syntax:**

SELECT expr\_1, expr\_2, ... expr\_n

FROM table\_1

INNER JOIN table\_2

ON join\_predicate;

**join\_predicate:** It is used to specify the joining conditions to be strictly followed for joining.

* **Example:**

SELECT “EMPLOYMENT”.“ID”, “EMPLOYMENT”.“STATE”, “DEPARTMENT”.“NAME”

FROM “EMPLOYMENT”

INNER JOIN “DEPARTMENT”

ON “EMPLOYMENT”.“ID” = “DEPARTMENT”.“ID”;

# RIGHT OUTER JOIN In PostgreSQL

The Right Outer Join query after joining returns all the records from the Right table for the specified fields along with the records from the Left table where the join condition is met.

* **Syntax:**

SELECT expr\_1, expr\_2, ... expr\_n

FROM table\_1

RIGHT OUTER JOIN table\_2

ON join\_predicate;

* **Example:**

SELECT “EMPLOYMENT”.“ID”, “EMPLOYMENT”.“STATE”, “DEPARTMENT”.“NAME”

FROM “EMPLOYMENT”

RIGHT OUTER JOIN “DEPARTMENT”

ON “EMPLOYMENT”.“ID” = “DEPARTMENT”.“ID”;

# LEFT OUTER JOIN in PostgreSQL

The Left Outer Join query after joining returns all the records from the Left table for the specified fields along with the records from the Right table where the join condition is met.

* **Syntax:**

SELECT expr\_1, expr\_2, ... expr\_n

FROM table\_1

LEFT OUTER JOIN table\_2

ON join\_predicate;

* **Example:**

SELECT “EMPLOYMENT”.“ID”, “EMPLOYMENT”.“STATE”, “DEPARTMENT”.“NAME”

FROM “EMPLOYMENT”

LEFT OUTER JOIN “DEPARTMENT”

ON “EMPLOYMENT”.“ID” = “DEPARTMENT”.“ID”;

# FULL OUTER JOIN in PostgreSQL

The Full Outer Join query after joining returns all the records from the selected fields of both the tables irrespective of the fact that the join condition is met or not.

* **Syntax:**

SELECT expr\_1, expr\_2, ... expr\_n

FROM table\_1

FULL OUTER JOIN table\_2

ON join\_predicate;

* **Example:**

SELECT “EMPLOYMENT”.“ID”, “EMPLOYMENT”.“STATE”, “DEPARTMENT”.“NAME”

FROM “EMPLOYMENT”

FULL OUTER JOIN “DEPARTMENT”

ON “EMPLOYMENT”.“ID” = “DEPARTMENT”.“ID”;

# CROSE JOIN in PostgreSQL

The PostgreSQL CROSS Join query combines each row of the first table with each row of the second table in the result set. Thus, if we select all the fields of both the table than the resultant table contains x\*y rows, where the FIRST table has x number of rows and the Second Table has y number of rows.

* **Syntax:**

SELECT columns

FROM table\_1

CROSS JOIN table\_2;

* **Example:**

SELECT \*

FROM “EMPLOYMENT”

CROSS JOIN “DEPARTMENT”;

# FUNCTION in PostgreSQL

PostgreSQL functions are stored procedures and can be easily understood as a set of SQL and procedural statements. They are stored on the database server. A function can be invoked using the SQL interface. It facilitates to ease the operations within the database. A PostgreSQL function can be created in several languages including SQL, PL/pgSQL, C and Python.

* **Syntax:**

CREATE [OR REPLACE] FUNCTION function\_name (arguments)

RETURNS return\_datatype AS $variable\_name$

DECLARE

declaration;

[...]

BEGIN

< function\_body >

[...]

RETURN { variable\_name | value }

END;

LANGUAGE plpgsql;

* **Explanation :**

**function\_name:** It is used to specify the name of the function.  
**[OR REPLACE]:**It is an optional method which facilitates to modify an existing function.  
**RETURN:** It is used to specify the data type to be returned from the function. Its value can reference the type of a table column or can be a base, composite, or domain type.  
**function\_body:** It is used to specify the executable parts.  
**plpgsql:** It is used to specify the name of the language in which the function is implemented.

* **Example:**

CREATE OR REPLACE FUNCTION total\_students()

RETURNS integer AS $total$

declare

total integer;

BEGIN

SELECT count(\*) into total FROM STUDENTS;

**RETURN** total;

END;

$total$ LANGUAGE plpgsql;