Unit 1 INTRODUCTION TO RDBMS

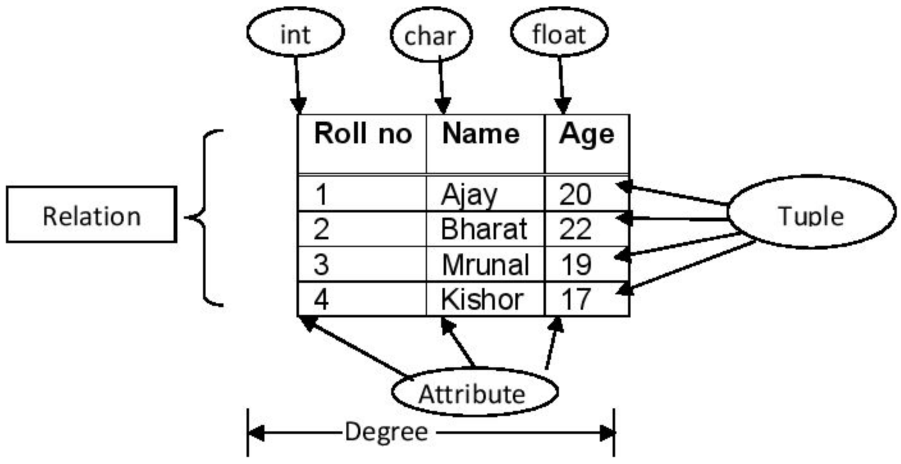
• Introduction to RDBMS, • Introduction to Open Source software PostgreSQL, • Installation of open source software PostgreSQL on Windows and Linux, • Data types of PostgreSQL

Introduction to RDBMS :

The relational model is an abstract theory of data which is based on mathematical theory, introduced by Dr. E. F. Codd. The following table gives a list of relational terms & their corresponding informal equivalent.

Sr. No.	Formal relational terms	Informal equivalent
1	Relation	Table
2	Tuple	Row or record
3	Cardinality	Number of rows or records
4	Attribute	Column or field
5	Degree	No. of columns
6	Primary key	Unique identifier
7	Domain	Set of legal values or possible data values.

Consider following relational data structure.



Introduction to Open Source software PostgreSQL

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development phase and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.

This tutorial will give you a quick start with PostgreSQL and make you comfortable with PostgreSQL programming.

What is PostgreSQL?

PostgreSQL (pronounced as **post-gress-Q-L**) is an open source relational database management system (DBMS) developed by a worldwide team of volunteers. PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge.

A Brief History of PostgreSQL

PostgreSQL, originally called Postgres, was created at UCB by a computer science professor named Michael Stonebraker. Stonebraker started Postgres in 1986 as a follow-up project to its predecessor, Ingres, now owned by Computer Associates.

- **1977-1985** − A project called INGRES was developed.
  - Proof-of-concept for relational databases
  - Established the company Ingres in 1980
  - Bought by Computer Associates in 1994
- **1986-1994** − POSTGRES
  - Development of the concepts in INGRES with a focus on object orientation and the query language - Quel
  - The code base of INGRES was not used as a basis for POSTGRES
  - Commercialized as Illustra (bought by Informix, bought by IBM)
- **1994-1995** − Postgres95
  - Support for SQL was added in 1994
  - Released as Postgres95 in 1995
  - Re-released as PostgreSQL 6.0 in 1996
  - Establishment of the PostgreSQL Global Development Team

Key Features of PostgreSQL

PostgreSQL runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It supports text, images, sounds, and video, and includes programming interfaces for C / C++, Java, Perl, Python, Ruby, Tcl and Open Database Connectivity (ODBC).

PostgreSQL supports a large part of the SQL standard and offers many modern features including the following −

- Complex SQL queries
- SQL Sub-selects
- Foreign keys
- Trigger
- Views
- Transactions
- Multiversion concurrency control (MVCC)
- Streaming Replication (as of 9.0)
- Hot Standby (as of 9.0)

You can check official documentation of PostgreSQL to understand the above-mentioned features. PostgreSQL can be extended by the user in many ways. For example by adding new −

- Data types
- Functions
- Operators
- Aggregate functions
- Index methods

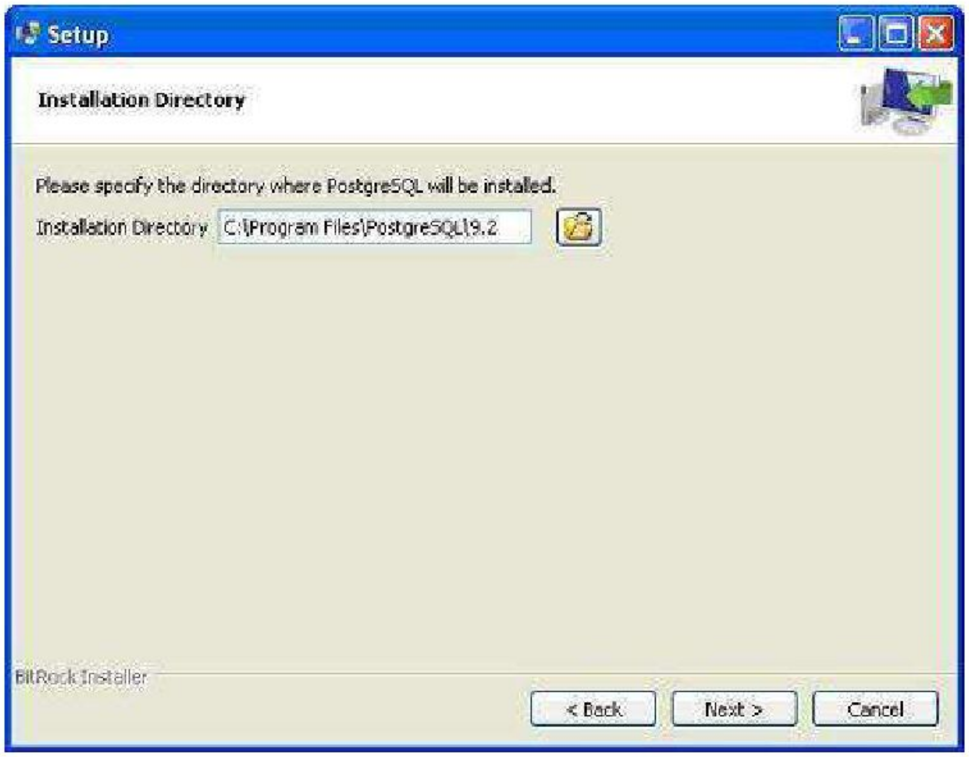
Procedural Languages Support

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

Installing PostgreSQL on Windows

Follow the given steps to install PostgreSQL on your Windows machine. Make sure you have turned Third Party Antivirus off while installing.

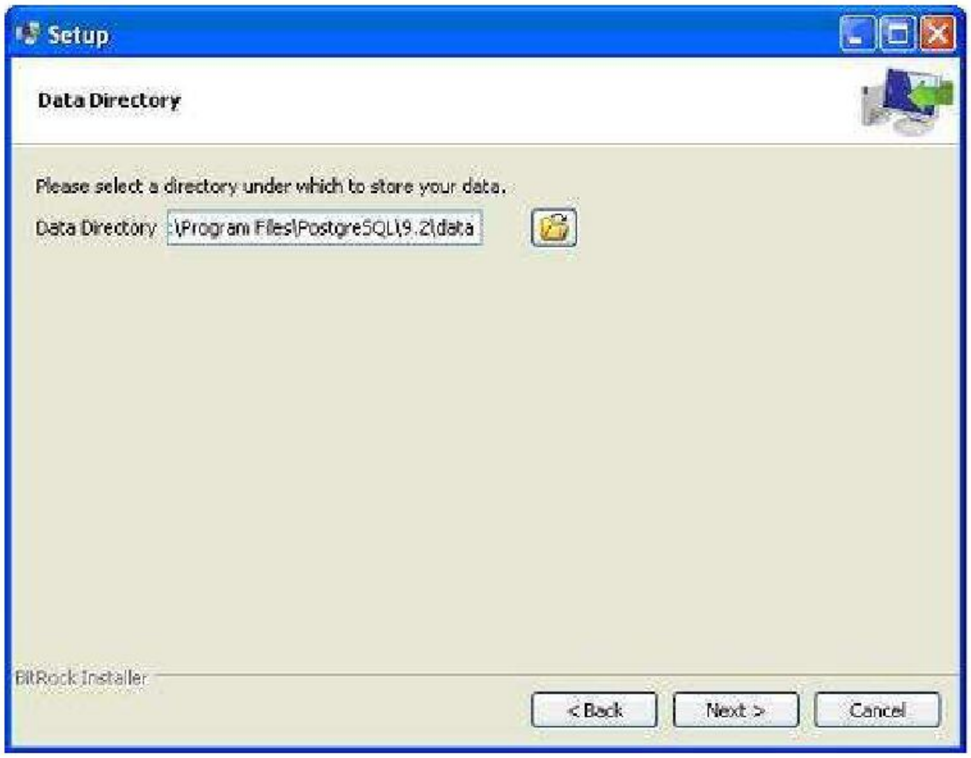
- Pick the version number of PostgreSQL you want and, as exactly as possible, the platform you want from EnterpriseDB
- I downloaded postgresql-9.2.4-1-windows.exe for my Windows PC running in 32bit mode, so let us run **postgresql-9.2.4-1-windows.exe** as administrator to install PostgreSQL. Select the location where you want to install it. By default, it is installed within Program Files folder.



Please specify the directory where PostgreSQL will be installed.

Installation Directory C:\Program Files\PostgreSQL\9.2

- The next step of the installation process would be to select the directory where your data would be stored. By default, it is stored under the "data" directory.

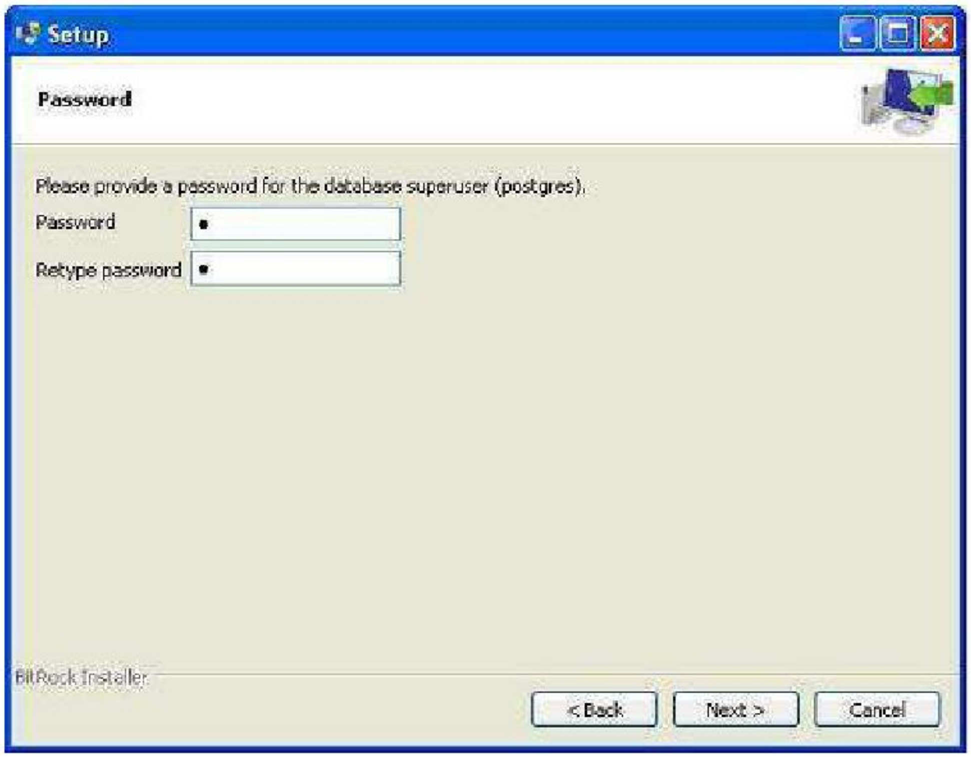

Setup

Data Directory

Please select a directory under which to store your data.

Data Directory: :\Program Files\PostgreSQL\9.2\data

- Next, the setup asks for password, so you can use your favorite password.


Setup

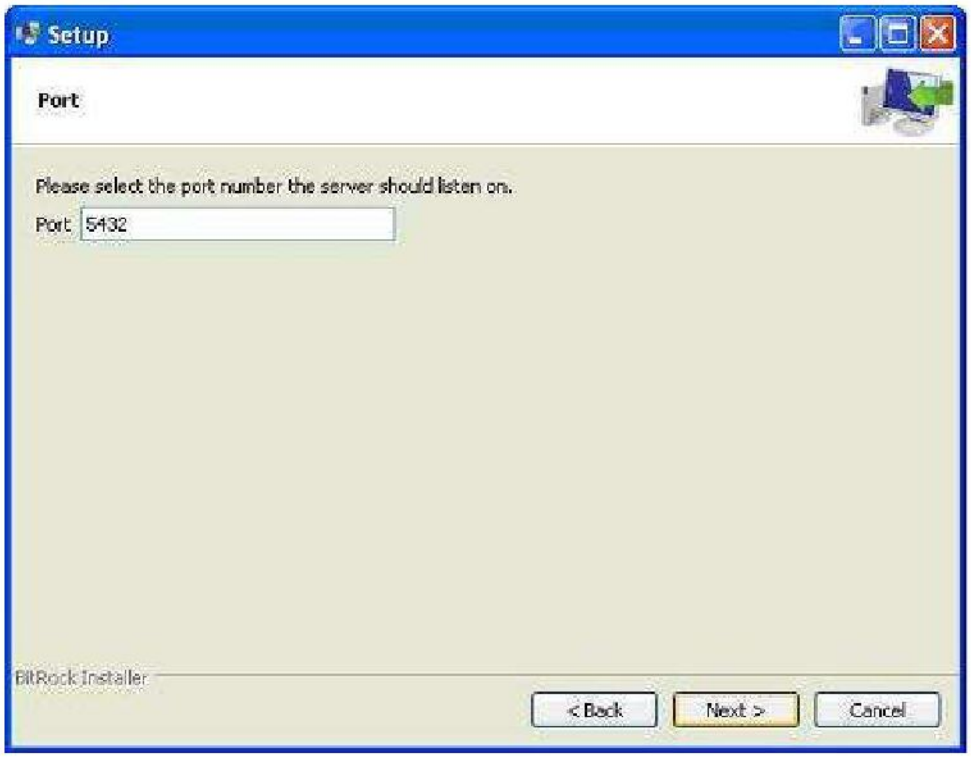
Password

Please provide a password for the database superuser (postgres).

Password: •

Retype password: •

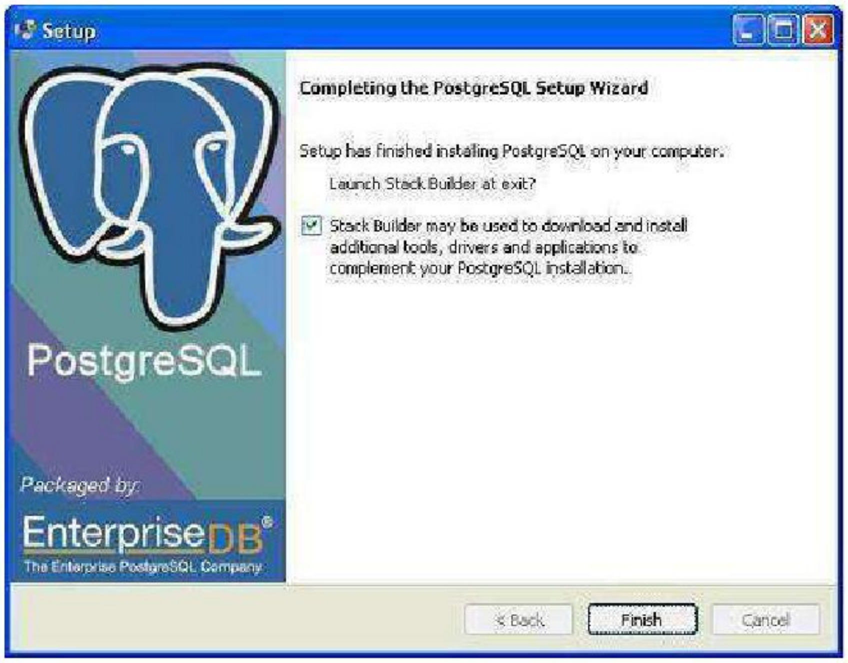
- The next step; keep the port as default.



Please select the port number the server should listen on.

Port: 5432

- In the next step, when asked for "Locale", I selected "English, United States".
- It takes a while to install PostgreSQL on your system. On completion of the installation process, you will get the following screen. Uncheck the checkbox and click the Finish button.



Completing the PostgreSQL Setup Wizard

Setup has finished installing PostgreSQL on your computer.

Launch Stack Builder at exit?

[✓] Stack Builder may be used to download and install additional tools, drivers and applications to complement your PostgreSQL installation.

After the installation process is completed, you can access pgAdmin III, StackBuilder and PostgreSQL shell from your Program Menu under PostgreSQL 9.2.

Data types of PostgreSQL

Numeric Types

Numeric types consist of two-byte, four-byte, and eight-byte integers, four-byte and eight-byte floating-point numbers, and selectable-precision decimals. The following table lists the available types.

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to 9223372036854775807
decimal	variable	user-specified precision,exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision,exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision,inexact	6 decimal digits precision
double precision	8 bytes	variable-precision,inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Monetary Types

The *money* type stores a currency amount with a fixed fractional precision. Values of the *numeric, int, and bigint* data types can be cast to *money*. Using Floating point numbers is not recommended to handle money due to the potential for rounding errors.

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

Character Types

The table given below lists the general-purpose character types available in PostgreSQL.

S. No.	Name & Description
1	**character varying(n), varchar(n)** variable-length with limit
2	**character(n), char(n)** fixed-length, blank padded
3	**text** variable unlimited length

Binary Data Types

The *bytea* data type allows storage of binary strings as in the table given below.

Name	Storage Size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

Date/Time Types

PostgreSQL supports a full set of SQL date and time types, as shown in table below. Dates are counted according to the Gregorian calendar. Here, all the types have resolution of **1 microsecond / 14 digits** except **date** type, whose resolution is **day**.

Name	Storage Size	Description	Low Value	High Value
timestamp [(p)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD
TIMESTAMPTZ	8 bytes	both date and time, with time	4713 BC	294276 AD

		zone		
date	4 bytes	date (no time of day)	4713 BC	5874897 AD
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00
time [(p)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459
interval [fields] [(p)]	12 bytes	time interval	-178000000 years	178000000 years

Boolean Type

PostgreSQL provides the standard SQL type Boolean. The Boolean data type can have the states *true*, *false*, and a third state, *unknown*, which is represented by the SQL null value.

Name	Storage Size	Description
boolean	1 byte	state of true or false

Enumerated Type

Enumerated (enum) types are data types that comprise a static, ordered set of values. They are equivalent to the enum types supported in a number of programming languages.

Unlike other types, Enumerated Types need to be created using CREATE TYPE command. This type is used to store a static, ordered set of values. For example compass directions, i.e., NORTH, SOUTH, EAST, and WEST or days of the week as shown below −

```
CREATE TYPE week AS ENUM ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun');
```

Enumerated, once created, can be used like any other types.

Geometric Type

Geometric data types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

Name	Storage Size	Representation	Description
point	16 bytes	Point on a plane	(x,y)
line	32 bytes	Infinite line (not fully implemented)	((x1,y1),(x2,y2))
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))

box	32 bytes	Rectangular box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n	Polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	Circle	<(x,y),r> (center point and radius)

Unit 2 DATABASE AND TABLE OPERATIONS [L : 05 M: 10] • Database Operations - 1.Creating a Database 2.Dropping the Database • Table Operations – 1. Create 2. Alter3. Drop

Database Operations-

1. **Creating a Database**

The CREATE DATABASE statement allows you to create a new PostgreSQL database.
The following shows the syntax of the CREATE DATABASE statement:

```
CREATE DATABASE database_name
WITH
  [OWNER = role_name]
  [TEMPLATE = template]
  [ENCODING = encoding]
  [LC_COLLATE = collate]
  [LC_CTYPE = ctype]
  [TABLESPACE = tablespace_name]
  [ALLOW_CONNECTIONS = true | false]
  [CONNECTION LIMIT = max_concurrent_connection]
  [IS_TEMPLATE = true | false ]
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

To execute the CREATE DATABASE statement you need to have a superuser role or a special CREATEDB privilege.

To create a new database:

- First, specify the name of the new database after the CREATE DATABASE keywords. The database name must be unique in the PostgreSQL database server. If you try to create a database whose name already exists, PostgreSQL will issue an error.
- Then, specify one or more parameters for the new database.

Parameters

OWNER

Assign a role that will be the owner of the database. If you omit the OWNER option, the owner of the database is the role that you use to execute the CREATE DATABASE statement.

TEMPLATE

Specify the template database from which the new database is created. By default, PostgreSQL uses the template1 database as the template database if you don't explicitly specify the template database.

ENCODING

Determine the character set encoding in the new database.

LC_COLLATE

Specify the collation order (LC_COLLATE) that the new database will use. This parameter affects the sort order of string in the queries that contain the ORDER BY clause. It defaults to the LC_COLLATE of the template database.

LC_CTYPE

Specify the character classification that the new database will use. It affects the classification of character e.g., lower, upper, and digit. It defaults to the LC_CTYPE of the template database

TABLESPACE

Specify the tablespace name for the new database. The default is the tablespace of the template database.

CONNECTION LIMIT

Specify the maximum concurrent connections to the new database. The default is -1 i.e., unlimited. This parameter is useful in the shared hosting environments where you can configure the maximum concurrent connections for a particular database.

ALLOW_CONNECTIONS

The allow_connections parameter is a boolean value. If it is false, you cannot connect to the database.

TABLESPACE

Specify the tablespace that the new database will use. It defaults to the tablespace of the template database.

IS_TEMPLATE

If the IS_TEMPLATE is true, any user with the CREATEDB privilege can clone it. If false, only superusers or the database owner can clone it.

PostgreSQL CREATE DATABASE examples

1) Create a database with default parameters

First, log in to the PostgreSQL using any client tool.

Second, execute the following statement to a new database with default parameters:

```
CREATE DATABASE sales;
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

PostgreSQL created a new database named sales that has default parameters from the default template database (template1).

Third, if you use the psql client tool, you can view all the databases in the current PostgreSQL database server using the \l command:

```
\l
```

2) Create a database with some parameters

The following example uses the CREATE DATABASE statement to create a database named college with some parameters:

```
CREATE DATABASE college
WITH
        ENCODING='UTF8'
        OWNER       = postgres
        CONNECTION LIMIT=100
```

In this example, we created the postgres database with the encoding UTF8, the owner is hr and the number of concurrent connections to the database is 100.

2. **Dropping the Database**

Once a database is no longer needed, you can drop it by using the DROP DATABASE statement.
The following illustrates the syntax of the DROP DATABASE statement:

```
DROP DATABASE [IF EXISTS] database_name;
```

Code language: SQL (Structured Query Language) (sql)
To delete a database:

- Specify the name of the database that you want to delete after the DROP DATABASE clause.
- Use IF EXISTS to prevent an error from removing a non-existent database. PostgreSQL will issue a notice instead.

The DROP DATABASE statement deletes catalog entries and data directory permanently. This action cannot be undone so you have to use it with caution.
Only superusers and the database owner can execute the DROP DATABASE statement. In addition, you cannot execute the DROP DATABASE statement if the database still has active connections. In this case, you need to disconnect from the database and connect to another database e.g., postgres to execute the DROP DATABASE statement.
PostgreSQL also provides a utility program named dropdbthat allows you to remove a database. The dropdb program executes the DROP DATABASE statement behind the scenes.

1) Drop a database that has no active connection example
To remove the hrdbdatabase, use the hrdb owner to connect to a database other than hrdbdatabase e.g., postgres and issue the following statement:

```
DROP DATABASE hrdb;
```

Code language: SQL (Structured Query Language) (sql)
PostgreSQL deleted the hrdbdatabase.

2) Drop a database that has active connections example
The following statement deletes the testdb1database:

```
DROP DATABASE testdb1;
```

Code language: SQL (Structured Query Language) (sql)

However, PostgreSQL issued an error as follows:

```
ERROR: database "testdb1" is being accessed by other users
SQL state: 55006
Detail: There is 1 other session using the database.
```

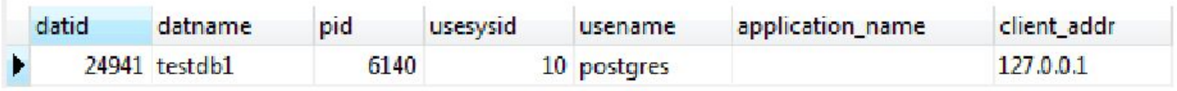
Code language: JavaScript (javascript)

To drop the testdb1 database, you need to terminate the active connection and drop the database.

First, query the pg_stat_activityview to find what activities are taking place against the testdb1database:

```
SELECT *
FROM pg_stat_activity
WHERE datname = 'testdb1';
```

Code language: SQL (Structured Query Language) (sql)



datid	datname	pid	usesysid	usename	application_name	client_addr
24941	testdb1	6140	10	postgres		127.0.0.1

The testdb1database has one connection from localhosttherefore it is safe to terminate this connection and remove the database.

Second, terminate the connection to the testdb1database by using the following statement:

```
SELECT
        pg_terminate_backend (pg_stat_activity.pid)
FROM
        pg_stat_activity
WHERE
        pg_stat_activity.datname = 'testdb1';
```

Code language: SQL (Structured Query Language) (sql)

Third, issue the DROP DATABASE command to remove the testdb1database:

```
DROP DATABASE testdb1;
```

Code language: SQL (Structured Query Language) (sql)

PostgreSQL drops the testdb1permanently.

In this tutorial, you have learned how to use the PostgreSQL DROP DATABASE statement to drop a database. In addition, you also learned how to delete a database that has active connections.

3. Table Operations

CREATE TABLE-

The PostgreSQL CREATE TABLE statement is used to create a new table in any of the given database.

Syntax

Basic syntax of CREATE TABLE statement is as follows –

```
CREATE TABLE table_name(
  column1 datatype,
  column2 datatype,
  column3 datatype,
  .....
  columnN datatype);
```

CREATE TABLE is a keyword, telling the database system to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Initially, the empty table in the current database is owned by the user issuing the command.

Then, in brackets, comes the list, defining each column in the table and what sort of data type it is. The syntax will become clear with an example given below.

Examples

The following is an example, which creates a EMPLOYEE table

```
CREATE TABLE employee(
  EMPID              INT,
  NAME        TEXT,
  AGE        INT,
  ADDRESS      CHAR(50),
  SALARY      REAL
);
```

Let us create one more table, which we will use in our exercises in subsequent chapters –

```
CREATE TABLE DEPARTMENT(
  DEPTID              INT,
  DEPT_NAME      VARCHAR(20),
   LOCATION         VARCHAR(20)
);
```

You can verify if your table has been created successfully using **\d** command, which will be used to list down all the tables in an attached database.

When the above two tables were created , we can list all the tables which are available in the databases by following command.

```
SELECT * FROM PG_TABLES;
```

Describe the Table

To get information on columns of a table, you query the information_schema.columns catalog. For example:

```
SELECT
  table_name,
  column_name,
  data_type
FROM
  information_schema.columns
WHERE
  table_name = 'employee';
```

ALTER TABLE

The PostgreSQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table.

You would also use ALTER TABLE command to add and drop various constraints on an existing table.

Syntax

ALTER TABLE (ADD)

The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows −

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

EXAMPLE:

If we consider the above table EMPLOYEE for alter the table. Add the new column contact_no , the example is as follows

```
ALTER TABLE EMPLOYEE
  ADD CONTACT_NO INT;
```

ALTER TABLE (modify)

The basic syntax of ALTER TABLE to change the **DATA TYPE** of a column in a table is as follows −
ALTER TABLE table_name ALTER COLUMN column_name TYPE datatype;

Example:
```
ALTER TABLE employee
  ALTER COLUMN NAME TYPE VARCHAR(20)
```

ALTER TABLE(DROP)

The basic syntax of ALTER TABLE to **DROP COLUMN** in an existing table is as follows −
ALTER TABLE table_name DROP COLUMN column_name;

Example

```
  ALTER TABLE EMPLOYEE
    DROP COLUMN CONTACT_NO;
```

DROP TABLE-

The PostgreSQL DROP TABLE statement is used to remove a table definition and all associated data, indexes, rules, triggers, and constraints for that table.

You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

Syntax

Basic syntax of DROP TABLE statement is as follows −

```
DROP TABLE table_name;
```

To drop a table from the database, you use the `DROP TABLE` statement as follows:

```
DROP TABLE [IF EXISTS] table_name
[CASCADE | RESTRICT];
```
Code language: SQL (Structured Query Language) (sql)

In this syntax:

- First, specify the name of the table that you want to drop after the `DROP TABLE` keywords.
- Second, use the `IF EXISTS` option to remove the table only if it exists.

If you remove a table that does not exist, PostgreSQL issues an error. To avoid this situation, you can use the `IF EXISTS` option.
In case the table that you want to remove is used in other objects such as views, triggers, functions, and stored procedures, the `DROP TABLE` cannot remove the table. In this case, you have two options:

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

To remove multiple tables at once, you can place a comma-separated list of tables after the `DROP TABLE` keywords:

```
DROP TABLE [IF EXISTS]
   table_name_1,
   table_name_2,
   ...
[CASCADE | RESTRICT];
Code language: CSS (css)
```

Note that you need to have the roles of the superuser, schema owner, or table owner in order to drop tables.

Example:

DROP TABLE student;

This command Drop the table student

DROP TABLE student, teacher;

Above command Drop multiple tables simultaneously

DROP TABLE library cascade;

The command Drop library table and its related object

Unit 3 SQL – STATEMENTS, OPERATORS, FUNCTIONS [L : 10 M: 20] • Statements - SELECT, INSERT, UPDATE, DELETE • Null value and Default value • Operators - Arithmetic, Logical, Comparison, Bitwise, Relational • Functions - Aggregate functions, Date and Time functions, String functions• Clauses:- where, order by, AND, OR, Between, Like, CASE, Distinct, Group by, Having

SELECT Statement

The SELECT statement is used to select data from a database.

The result is stored in a result table, called the result-set.

SQL SELECT Syntax

```
SELECT column_name,column_name
FROM table_name;
```

and

```
SELECT * FROM table_name;
```

SELECT Column Example

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:

Example

```
SELECT CustomerName,City FROM Customers;
```

The SQL SELECT DISTINCT Statement

In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.

The DISTINCT keyword can be used to return only distinct (different) values.

SQL SELECT DISTINCT Syntax

```
SELECT DISTINCT column_name,column_name
FROM table_name;
```

SELECT DISTINCT Example

The following SQL statement selects only the distinct values from the "City" columns from the "Customers" table:

Example

```
SELECT DISTINCT City FROM Customers;
```

INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

SQL INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two forms.

The first form does not specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name
VALUES (value1,value2,value3,...);
```

The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

INSERT INTO Example

Assume we wish to insert a new row in the "Customers" table.

We can use the following SQL statement:

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal','Tom B. Erichsen','Skagen 21','Stavanger','4006','Norway');
```

UPDATE Statement

SQL SERVER: UPDATE STATEMENT

This SQL Server tutorial explains how to use the **UPDATE statement** in SQL Server (Transact-SQL) with syntax and examples.

DESCRIPTION

The PostgreSQL **UPDATE statement** is used to update existing records in a table in Postgre database. There are 3 syntaxes for the UPDATE statement depending on whether you are performing a traditional update or updating one table with data from another table.

SYNTAX

The syntax for the **UPDATE statement** when updating one table in PostgreSQL) is:

```
UPDATE table
SET column1 = expression1,
   column2 = expression2,
   ...
WHERE conditions;
```

OR

The syntax for the **UPDATE statement** when updating one table with data from another table in syntax is

```
UPDATE table1
SET column1 = (SELECT expression1
            FROM table2
            WHERE conditions)
WHERE conditions;
```

DELETE STATEMENT

DESCRIPTION

DELETE statement is used to delete a single record or multiple records from a table in SQL Server.

SYNTAX

In the simplest form, the syntax for the **DELETE statement** is:

```
DELETE FROM table
WHERE conditions;
```

However, the full syntax for the **DELETE statement** in PostgreSQL Server is:

- **NULL Value and Default Value**

NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column. A NOT NULL constraint is always written as a column constraint.

A NULL is not the same as no data; rather, it represents unknown data.

Example

For example, the following PostgreSQL statement creates a new table called EMPLOYEE and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULL values –

```
CREATE TABLE EMPLOYEE(
  ID          INT    NOT NULL,
  NAME         TEXT   NOT NULL,
  AGE          INT    NOT NULL,
  ADDRESS       CHAR(50),
  SALARY        REAL
);
```

The **DEFAULT Constraint** is used to fill a column with a default and fixed value. The value will be added to all new records when no other value is provided.

Using DEFAULT on CREATE TABLE :

Syntax :
CREATE TABLE tablename (
Columnname **DEFAULT** 'defaultvalue');

Example –
To set a DEFAULT value for the “Location” column when the “Geeks” table is created –

```
CREATE TABLE student (
SID int NOT NULL,
sname varchar(20),
Age int,
Location varchar(20) DEFAULT 'Bhusawal');
```

```
CREATE TABLE jobs (
JOB_ID varchar(10) NOT NULL UNIQUE,
JOB_TITLE varchar(35) NOT NULL DEFAULT ' ',
MIN_SALARY decimal(6,0) DEFAULT 8000,
MAX_SALARY decimal(6,0) DEFAULT NULL
);
```

- **Operators - Arithmetic, Logical, Comparison, Bitwise, Relational**

An operator is a reserved word or a character used primarily in a PostgreSQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in a PostgreSQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators

PostgreSQL Arithmetic Operators

Assume variable **a** holds 2 and variable **b** holds 3, then −

Example

Operator	**Description**	**Example**
+	Addition - Adds values on either side of the operator	a + b will give 5
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -1
*	Multiplication - Multiplies values on either side of the operator	a * b will give 6
/	Division - Divides left hand operand by right hand operand	b / a will give 1
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 1
^	Exponentiation - This gives the exponent value of the right hand operand	a ^ b will give 8

|/	square root	|/ 25.0 will give 5
||/	Cube root	||/ 27.0 will give 3
!	Factorial	5 ! will give 120
!!	factorial (prefix operator)	!! 5 will give 120

PostgreSQL Comparison Operators

Assume variable a holds 10 and variable b holds 20, then −

Show Examples

Operator	**Description**	**Example**
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right	(a <= b) is true.

	operand, if yes then condition becomes true.	

PostgreSQL Logical Operators

Here is a list of all the logical operators available in PostgresSQL.

Show Examples

S. No.	**Operator & Description**
1	**AND** The AND operator allows the existence of multiple conditions in a PostgresSQL statement's WHERE clause.
2	**NOT** The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN etc. **This is negate operator**.
3	**OR** The OR operator is used to combine multiple conditions in a PostgresSQL statement's WHERE clause.

PostgreSQL Bit String Operators

Bitwise operator works on bits and performs bit-by-bit operation. The truth table for & and | is as follows −

P	**q**	**p & q**	**p | q**
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

~A = 1100 0011

Show Examples

The Bitwise operators supported by PostgreSQL are listed in the following table −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the	A >> 2 will give 15 which is 0000 1111

	number of bits specified by the right operand.	
#	bitwise XOR.	A # B will give 49 which is 0100 1001

AGGREGATE FUNCTIONS: -

function serves the purpose of manipulating data items and returning a result. Functions are also capable of accepting user supplied variable or constant and operating on them. Such variables or constant are called arguments. Any number of arguments can be passed to the function in the following format,
Function_name(argumet1,argumet2,……)

PostgreSQL function can be club together depending upon wheatear they operate on a single row or a group of single retrieving a table. According a function can be classified as follows,

Group function /Aggregate Function:-
Functions that act on a set of values are called group functions.

E.g. Sum( ) is a function which calculate the total set of numbers. A group function returns a single result row for a group of queried rows.

There are five group /aggregate functions:-

➢ **AVG( ): -**
Returns an average value of 'n' ignoring null values in acloumn.

Syntax:-

```
AVG(n)
```

E.g. i) List the average sale price from product master table.

```
Select  AVG( sale_price)
From product_master;
→ AVG
2804000
```

ii) List average salary of dept no 10

```
select AVG(salary)
```

```
from employee

where dept_no=10;

→ AVG (salary)

567000
```

- **COUNT( ): -**

  Returns the no. of rows where the expression is not null

  E.g. i) How many product exist I the prod_master table.

```
Select COUNT(prod_no)

From prod_master;

→ count(prod_no)

20
```

  ii) How many employee works in department no 10

```
select COUNT(employee_no)
from employee
where dept_no=10;
→ COUNT(employee)
28
```

- **MAX( ):-**

  Returns the maximum value of expression

  Syntax:-

  MAX(expression)

  Eg. i) List the maximum salary of employee table

```
Select MAX(salary)
From employee;
```

  ii) List the maximum salary in dept_no 10

```
Select MAX(salary)
From employee
Where dept_no=10;
```

- **MIN( ):-**

  It returns the minimum value from expression.

  Syntax:-

  MIN(expression)

E.g. i) Display the minimum salary from emp table.

```
Select MIN(salary)
From emp;
```

ii) Select the minimum salary from dept_no 10

```
Select MIN(salary)
From emp
Where dept_no=10;
```

- **SUM( ) :-**

  Returns the sum of values of'n'

  Syntax:-

  SUM(n)

E.g. i) Display the total salary expenses from employee table.

```
Select SUM(salary)
From employee;
```

ii) Display the total salary from dept_no 10.

```
Select SUM(salary)
From employee
Where dept_no=10;
```

String Functions

Function that act an only one value at a time are called scalar function. A single row function returns one result for every row of a queried table.

I) String Function:-

Works for string datatype.

1) LOWER( ):-

Returns character with all letters in lower case.

Syntax:- LOWER(char)

E.g. i) select LOWER('RAMESH')

ii) Select LOWER(emp_name)
from emp;

```
iii) select LOWER(emp_name)
     from emp
     where dept_no=10;
```

2) UPPER( ) :-

Returns character with all letters in uppercase.

Syntax:- UPPER(char)

E.g. i) Select UPPER('sunil')

3) INTICAP( ):-

Returns a string with a first letter of each word I uppercase.

Syntax:- INTICAP(char)

E.g. i) select INTICAP('RAM')

```
ii) select INTICAP(emp_name)
    from employee;
```

4) SUBSTR( ):-

Returns operation of character beginning at character 'm' and going up to character 'n' is up to the last char in the string. The first position of character is 1.

Syntax:- SUBSTR(char,m,)

```
E.g. select SUBSTR('ulka',2,4)
     From dual;
```

5) LENGTH( ):-

Returns the length of a word

Syntax:- LENGTH(word)

E.g. i) select LENGTH('my name')

```
ii) select  length('emp_name')
    from employee
    where emp_no=1;
```

6) LTRIM( ):-

Removes character from the left of character with initial character remove up to the first character not in the set.

Syntax:- LTRIM(char[, set])

E.g. select LTRIM('NISHA','N')

From dual;
→ ISHA

7) RTRIM( ):-

Returns character with final character removed after the last character not in the set. Set is optional.

Syntax:- RTRIM(char[, set])

E.g. i) select RTRIM('SUNILA','A')

→ SUNIL

ii) select empno,RTRIM('emp_name'),salary
from emp;

Following are also String functions in tabular form with example

Function	**Description**	**Example**	**Result**
ASCII	Return the ASCII code value of a character or Unicode code point of a UTF8 character	SELECT ASCII('A')	65
CHR	Convert an ASCII code to a character or a Unicode code point to a UTF8 character	SELECT CHR(65)	'A'
CONCAT	Concatenate two or more strings into one	SELECT CONCAT('A','B','C')	'ABC'
CONCAT_WS	Concatenate strings with a separator	CONCAT_WS(',','A','B','C')	'A,B,C'
FORMAT	Format arguments based on a format string	FORMAT('Hello %s','Students')	'Hello Students'
INITCAP	Convert words in a string to title case	SELECT INITCAP('hI tHERE')	Hi There
LEFT	Return the first n character in a string	SELECT LEFT('ABC',1)	'A'
LENGTH	Return the number of characters in a string	SELECT LENGTH('ABC')	3
LOWER	Convert a string to lowercase	SELECT LOWER('hI tHERE')	'hi there'

Function	**Description**	**Example**	**Result**
LPAD	Pad on the left a a string with a character to a certain length	SELECT LPAD('12', 5, '00')	‘00012’
LTRIM	Remove the longest string that contains specified characters from the left of the input string	SELECT LTRIM(‘00123’)	‘123’
MD5	Return MD5 hash of a string in hexadecimal	MD5(‘ABC’)	
POSITION	Return the location of a substring in a string	POSTION(‘B’ in ‘A B C’)	3
REGEXP_MATCHES	Match a POSIX regular expression against a string and returns the matching substrings	SELECT REGEXP_MATCHES(‘ABC’, ‘^(A)(..)$’, ‘g’);	{A,BC}
REGEXP_REPLACE	Replace substrings that match a POSIX regular expression by a new substring	REGEXP_REPLACE(‘John Doe’,'(.*) (.*)’,’\2, \1’);	‘Doe, John’
REPEAT	Repeat string the specified number of times	REPEAT(‘*’, 5)	‘*****’
REPLACE	Replace all occurrences in a string of substring from with substring to	REPLACE(‘ABC’,’B’,’A’)	‘AAC’
REVERSE	Return reversed string.	REVERSE(‘ABC’)	‘CBA’
RIGHT	Return last n characters in the string. When n is negative, return all but first |n| characters.	RIGHT(‘ABC’, 2)	‘BC’
RPAD	Pad on the right of a string with a character to a certain length	RPAD(‘ABC’, 6, ‘xo’)	‘ABCxox’
RTRIM	Remove the longest string that contains specified characters from the right of the input string	RTRIM(‘abcxxzx’, ‘xyz’)	‘abc’

Function	Description	Example	Result
SPLIT_PART	Split a string on a specified delimiter and return nth substring	SPLIT_PART('2017-12-31′,'-',2)	'12'
SUBSTRING	Extract a substring from a string	SUBSTRING('ABC',1,1)	A'
TRIM	Remove the longest string that contains specified characters from the left, right or both of the input string	TRIM(' ABC ')	'ABC'
UPPER	Convert a string to uppercase	UPPER('hI tHERE')	'HI THERE'

Date and Time functions

We typically have to calculate ages in business applications e.g., ages of people, years of services of employees, etc. In PostgreSQL, you can use the AGE() function to achieve these tasks.

The following illustrates the syntax of the AGE() function:

AGE(timestamp,timestamp);

Code language: SQL (Structured Query Language) (sql)

The AGE() function accepts two TIMESTAMP values. It subtracts the second argument from the first one and returns an interval as a result.

See the following example:

```
SELECT AGE('2021-04-05','2011-06-24');
        "9 years 9 mons 11 days"
```

CURRENT_DATE

The PostgreSQL CURRENT_DATE function returns the current date.

Syntax

The CURRENT_DATE function is so simple that requires no argument as follows:

CURRENT_DATE

Return value

The CURRENT_DATE function returns a DATE value that represents the current date.
Examples
The following example shows how to use the CURRENT_DATE function to get the current date:

```
SELECT CURRENT_DATE;
```

The output is a DATE value as follows:

```
"2021-04-04"
```

CURRENT_TIME

The following illustrates the syntax of the CURRENT_TIME function:

```
CURRENT_TIME(precision)
```

Arguments
The CURRENT_TIME function accepts one optional argument:

1) precision

The precision argument specifies the returned fractional seconds precision. If you omit the precision argument, the result will include the full available precision.

Return value
The CURRENT_TIME function returns a TIME WITH TIME ZONE value that represents the current time with time zone.

Examples
The following example shows how to get the current time:

```
SELECT CURRENT_TIME;
```

The output is a TIME WITH TIME ZONE value as follows:

```
 "10:17:18.590166+05:30"
```

CURRENT_TIMESTAMP()
The PostgreSQL CURRENT_TIMESTAMP() function returns the current date and time with time zone, which is the time when the transaction starts.

Syntax
The following illustrates the syntax of the PostgreSQL CURRENT_TIMESTAMP() function:

```
CURRENT_TIMESTAMP(precision)
```

Code language: SQL (Structured Query Language) (sql)

Arguments

The PostgreSQL CURRENT_TIMESTAMP() function accepts one optional argument.

1) precision

The precision specifies the number of digits in the fractional seconds precision in the second field of the result.

If you omit the precision argument, the CURRENT_TIMESTAMP() function will return a TIMESTAMP with a time zone that includes the full fractional seconds precision available.

Return value

The CURRENT_TIMESTAMP() function returns a TIMESTAMP WITH TIME ZONE that represents the date and time at which the transaction started.

Examples

The following example shows how to use the CURRENT_TIMESTAMP() function to get the current date and time:

```
SELECT CURRENT_TIMESTAMP;
```

The result is:

```
"2021-04-04 10:21:21.938079+05:30"
```

DATE_PART('field', source)

These functions get the subfields. The *field* parameter needs to be a string value, not a name.

The valid field names are: *century, day, decade, dow, doy, epoch, hour, isodow, isoyear, microseconds, millennium, milliseconds, minute, month, quarter, second, timezone, timezone_hour, timezone_minute, week, year.*

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');

 date_part
-----------
      16
(1 row)

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
 date_part
-----------
       4
```

- **Clauses:- where, order by, AND, OR, Between, Like, CASE, Distinct, Group by, Having**

SELECT Command:-

The select statement is used to select a data from a table. The tabular result is stored in the result table. For extracting all rows & all columns from a table use can use the syntax

```
 Select *
From <table ame>;
Eg. SELECT * from item;
```

It retrieves all items from item table. It is not compulsory to display or retrieve all the data from table. The ways of filtering data will be,

i) Selected columns & all rows

ii) Selected rows & all columns

iii) Selected columns selected rows

FROM Command –

This SQL command is used for using table names from where we have to extract or retrieve data. More than one tables also be used with from clause.

Syntax is

```
From <table name>
```

i) Selected columns and all rows:-

The syntax is

```
select col1,clo2
 from <table name >

    eg. select item_no,Item_name
        from Item;
```

It retrieves only two columns on the screen.

Where Clause

ii) Selected rows & all columns:-

The table content all type of information but you want to view some specific type into but the simple command like select & from can't satisfy condition because there was no condition set that informed about the need to view a specific set of rows from the tables. PostgreSQL provides the option of using a where clause in an SQL query to apply a filter on the rows retrieves when a where clause is added to a SQL queries the engine compares each record in the table with the condition specify in the where clause displays only those records that satisfy the specific condition. The Syntax is

```
select *
 from <table name>
 where <codition>;
```

```
eg. 1)  select *
        from Item
        where Item_no=20;

    2) select Item_no,rate
       from Item
        where Item_name like 'A%';

    3) select item_name,qty
       from Item
        where raTe>=500;
```

with the where clause the following operators can be used

Operator	Description
=	equal
<>	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
Between	between the range
Like	search for pattern

Following are the arithmetic operators can be used in SQL SELECT clause as follows

+	addition
-	subtraction
/	division
*	multiplication
**	exponential
( )	enclosed operation

Consider the table Item with (Item_no,name,rate,qty,date_of_purchase)

If you apply the above arithmetic operator with SQL object clause the queries may executes for the following statements,

- **List all item_no either their totals(rate*qty)**

```
select Item_no, rate*qty
from Item;
```

- **List all the item no and their total with totals increased by 0.5**

```
select Item_on, (rate*qty)+0.05
from Item;
```

- We can also provide the where clause in the above computation as select all those item which are purchased on 10-Aug-06 with their increased rate by 50 Rs.

```
select Item_no, rate+50
From Item
Where date_of_purchase >'10-Aug-06';
```

LIKE Clause

The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator.

- The percent sign (%)
- The underscore (_)

The percent sign represents zero, one or multiple characters. The underscore represents a single number or character. These symbols can be used in combinations.

Syntax

The basic syntax of % and _ is as follows −

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'
```

*Examples :

Finds any values that start with 200

```
SELECT *
FROM EMPLOYEE
WHERE SALARY::text LIKE '200%'
Finds any values that start with 200
```

Finds all rows which contains salary 45 anywhere

```
SELECT *
FROM EMPLOYEE
WHERE SALARY::TEXT LIKE '_45%'
```

List all name starts with 'B'

```
SELECT *
FROM EMPLOYEE
WHERE ENAME LIKE 'B%
```

AND ,OR Operators

The PostgreSQL AND and OR operators are used to combine multiple conditions to narrow down selected data in a PostgreSQL statement. These two operators are called conjunctive operators.

These operators provide a means to make multiple comparisons with different operators in the same PostgreSQL statement.

The AND Operator

The AND operator allows the existence of multiple conditions in a PostgreSQL statement's WHERE clause. While using AND operator, complete condition will be assumed true when all the conditions are true. For example [condition1] AND [condition2] will be true only when both condition1 and condition2 are true.

Syntax

The basic syntax of AND operator with WHERE clause is as follows –

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using AND operator. For an action to be taken by the PostgreSQL statement, whether it be a transaction or query, all conditions separated by the AND must be TRUE.

List all employees whose age is greater than 25 and salary greater than 65000

```
SELECT *
FROM EMPLOYEE
WHERE AGE >= 25 AND SALARY >= 65000;
```

The OR Operator

The OR operator is also used to combine multiple conditions in a PostgreSQL statement's WHERE clause. While using OR operator, complete condition will be assumed true when at least any of the conditions is true. For example [condition1] OR [condition2] will be true if either condition1 or condition2 is true.

Syntax

The basic syntax of OR operator with WHERE clause is as follows −

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using OR operator. For an action to be taken by the PostgreSQL statement, whether it be a transaction or query, only any ONE of the conditions separated by the OR must be TRUE.

List all employees whose age is greater than 25 or salary greater than 65000

```
SELECT *
FROM COMPANY
```

WHERE AGE >= 25 OR SALARY >= 65000; The PostgreSQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns.

ORDER BY Clause

The PostgreSQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns.

Syntax

The basic syntax of ORDER BY clause is as follows −

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be available in column-list.

The following is an example, which would sort the result in ascending order by SALARY −

```
SELECT *
FROM EMPLOYEE
ORDER BY SALARY ASC
```

GROUP BY clause

The PostgreSQL GROUP BY clause is used in collaboration with the SELECT statement to group together those rows in a table that have identical data. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups.

The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax

The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

You can use more than one column in the GROUP BY clause. Make sure whatever column you are using to group, that column should be available in column-list.

i) List all the dept_no with it’s average salary.

```
SELECT dept_no, avg(salary)
From emp
GROUP BY dept_no;
```

ii) How many employees works I each department.

```
SELECT dept_no,COUNT(*)
From emp
GROUP BY dept_no;
```

iii) Who earns more salary from each department .

```
SELECT dept_no,MAX(salary)
From emp
GROUP BY dept_no;
```

HAVING clause

The HAVING clause allows us to pick out particular rows where the function's result meets some condition.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax

The following is the position of the HAVING clause in a SELECT query −

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause −

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

i) List all the dept_no with their average salary>5000.

```
Select dept_no, avg(salary)
 From emp
 Group by dept_no
 Having avg(salary)>5000;
```

ii) Which dept earns max salary from dept 10,20,30.

```
Select  dept_no, max(salary)
From emp
Group by dept_no
Having dept_on IN (10,20,30);
```

DISTINCT Clause

The PostgreSQL DISTINCT keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows –

SELECT DISTINCT name FROM COMPANY;

This would produce the following result where we do not have any duplicate entry –
SELECT DISTINCT column1, column2,.....columnN
FROM table_name

SELECT DISTINCT name FROM EMPLOYEE;

This would produce the following result where we do not have any duplicate entry −]

BETWEEN Clause

You can use **BETWEEN** clause to replace a combination of "greater than equal AND less than equal" conditions.

value BETWEEN low AND high;

Code language: SQL (Structured Query Language) (sql)

If the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, otherwise, it returns false.

You can rewrite the BETWEEN operator by using the greater than or equal (>=) or less than or equal (<=) operators like this:

value >= low and value <= high

Code language: SQL (Structured Query Language) (sql)

If you want to check if a value is out of a range, you combine the NOT operator with the BETW*EEN operator as follows:

value NOT BETWEEN low AND high;
Consider the Example of **Student** TABLE

Student
Rollno
Stu_name
Class
Fesspaid_date
Fees_paid

1. List all the students who paid > 5000 and < 9000 fees

   SELECT *
   FROM STUDENT
   WHERE FEES_PAID **BETWEEN** 5000 AND 9000

2. List all the student who paid fees between 5th June 2020 and 15 Sept 2020

   SELECT *
   FROM STUDENT
   WHERE FEESPAID_DATE **BETWEEN** '2020-06-05' '2020-09-15'

3. List all the student who does not paid fees between 10 Aug 2020 and 30 Aug 2020

   SELECT *
   FROM STUDENT
   WHERE FEESPAID_DATE **NOT BETWEEN** '2020-08-10' '2020-08-30'

CASE Statement

The PostgreSQL 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.

Since CASE is an expression, you can use it in any places where an expression can be used e.g.,SELECT, WHERE, GROUP BY, and HAVING clause.

The CASE expression has two forms: general and simple form.

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

In this syntax, each condition (condition_1, condition_2…) is a boolean expression that returns either true or false.

When a condition evaluates to false, the CASE expression evaluates the next condition from the top to bottom until it finds a condition that evaluates to true.

If a condition evaluates to true, the CASE expression returns the corresponding result that follows the condition. For example, if the condition_2 evaluates to true, the CASE expression returns the result_2. Also, it immediately stops evaluating the next expression.

In case all conditions evaluate to false, the CASE expression returns the result (else_result) that follows the ELSE keyword. If you omit the ELSE clause, the CASE expression returns NULL.

Example:

Create the table

```
CREATE TABLE stu_result
(SNAME VARCHAR(20),
 SUBJECT VARCHAR(20),
 marksinper numeric)
```

Insert the Records

```
 INSERT INTO stu_result
 VALUES('SADANAND MANE','ENGLISH',58),
           ('KISHOR YOGI','COMPUTER SCIENCE',78),
           ('RAJNI SAGNE','TELECOM',87),
           ('SHRIKANT PATIL','MARATHI',45),
           ('NAYNA BHOLE','HOME SCIENCE',32),
           ('ATUL PATIL','IT',90)
```

Write CASE statement

```
SELECT sname, subject,
  CASE
    WHEN marksinper >= 80 THEN 'DISTINCTION'
    WHEN marksinper >= 40 and marksinper < 80 THEN 'PASS'
      ELSE 'FAIL' END status from stu_result
```

Output of CASE statement

sname	subject	status
SADANAND MANE	ENGLISH	PASS
KISHOR YOGI	COMPUTER SC	PASS
RAJNI SAGNE	TELECOM	DISTINCTION
SHRIKANT PATIL	MARATHI	PASS
NAYNA BHOLE	HOME SCIENC	FAIL
ATUL PATIL	IT	DISTINCTION

UNIT 4

VIEW, JOIN and DATA CONSTRANTS in SQL

Syllabus

- *Constraints - Data Integrity, Entity Integrity*
- *Keys - PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK, Not Null*
- *Views - Create, Alter, Drop*
- *Join - Joins, Cross Join, Inner Join, Outer Join, Self-Join*
- *Sub queries - Sub queries as Constants, Sub queries as Correlated Values, Sub queries as Lists of Values, NOT IN and Sub queries with NULL Values, Sub queries Returning Multiple Columns*
- *Statement - MERGE Statement*
- *Set operations - UNION, EXCEPT, and INTERSECT*
- *Clauses - ANY, ALL, and EXISTS Clauses*

(10 L, 20 M)

Constraints :

Constraints are the rules enforced on data before the data get inserted into the table. These are used to prevent invalid data from being entered into the database.

Constraints are used to ensure the accuracy and reliability of the data in the database.

Data integrity refers to the accuracy and consistency (validity) of data over its lifecycle whereas Entity integrity ensures that each row in a table represents a single instance of the entity.

Ex. A column containing a salary of the employee should probably only accept positive values. But there is no standard data type that accepts only positive numbers. At that time to specify the positive value acceptance rule constraints can be defined at column level or table level.

Column level constraints are applied only to one column whereas table level constraints are applied to the whole table.

The constraints available in PostgreSQL are as follows :

- NOT NULL Constraint − Ensures that a column cannot have NULL value.
- UNIQUE Constraint − Ensures that all values in a column are different.
- PRIMARY Key – Uniquely identifies each row/record in a database table.
- FOREIGN Key − Constraints data based on columns in other tables.
- CHECK Constraint − The CHECK constraint ensures that all values in a column satisfy certain conditions. **NOT NULL Constraint**

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for column. A NOT NULL constraint is always used as a column level constraint.

To represent unknown data NULL is used but it is not same as no data.

For example

```
CREATE TABLE STUDENT (
 ROLLNO INT PRIMARY KEY,
 NAME  TEXT NOT NULL,
 MARKS INT NOT NULL,
 ADDRESS CHAR(50));
```

UNIQUE Constraint

PostgreSQL provides the user with a UNIQUE constraint to prevent two records from having identical values in a particular column. UNIQUE constrain is used to make sure that values stored in a column or a group of columns are unique across rows in a table.

Example

For example in the CUSTOMER table, if you want to prevent two or more people from having same email-id.

Here, email-id column is set to UNIQUE, so that you cannot have two records with same email-id.

```
CREATE TABLE CUSTOMER
(
 NAME  TEXT NOT NULL,
 ADDRESS CHAR (50),
 Email-id TEXT UNIQUE
);
```

PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values. PRIMARY KEY constraint is a combination of UNIQUE constraint and NOT NULL constraint.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**.

Example

Every student in STUDENT table is uniquely identified by rollno. Hence rollno is a primary key in a table STUDENT.

```
CREATE TABLE STUDENT(
 ROLLNO INT PRIMARY KEY,
 NAME TEXT NOT NULL,
 CLASS TEXT NOT NULL,
 PERCENTAGE REAL
);
```

FOREIGN KEY Constraint

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in the same row of another table. We say this maintains the referential integrity between two related tables. They are called foreign keys because of the constraints are foreign; that is, outside the table. Foreign keys are sometimes called a referencing key. The table that comprises the foreign key is called the referencing table or child table, and the table to that the foreign key references are known as the referenced table or parent table. It is possible for a table to have more than one foreign key.

Example

DEPT table has one primary key DEPT_NO which is used as foreign key in table EMP.

```
CREATE TABLE DEPT
(
 DEPT_NO INT PRIMARY KEY,
 DEPT_NAME TEXT NOT NULL,
 LOC TEXT
);

CREATE TABLE EMP
(
 EMP_ID INT PRIMARY KEY NOT NULL,
 EMP_NAME TEXT NOT NULL,
 SALARY REAL NOT NULL,
 DEPT_NO INT references DEPT(DEPT_NO)
);
```

We can also specify foreign key constraint at table level as follows,

```
CREATE TABLE EMP
(
 EMP_ID INT PRIMARY KEY NOT NULL,
 EMP_NAME TEXT NOT NULL,
 DEPT_NO INT,
 SALARY REAL NOT NULL,
 FOREIGN KEY (DEPT_NO) REFERENCES DEPT(DEPT_NO)
);
```

While using foreign key constraint ON **UPDATE** and **ON DELETE** options can have the following actions :

1. **NO ACTION :** UPDATEs and DELETEs to the primary key are prohibited if referenced by a foreign key row.
2. **CASCADE :** UPDATEs to the primary key update all foreign key columns that reference it. DELETEs on the primary key cause the deletion of all foreign key rows that reference it.
3. **SET NULL** : UPDATEs and DELETEs to the primary key row cause the foreign key to be set to NULL.

CHECK Constraint

The **CHECK** Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and is not entered into the table. The CHECK constraint enforces column value restrictions. Such constraints can restrict a column, for example, to a set of values, only in between given domain, or age in between specific range.

Example :

```
CREATE TABLE CUST
(
 ID INT PRIMARY KEY,
NAME TEXT NOT NULL,
AGE INT CHECK (AGE BETWEEN 22 AND 60),
CITY TEXT CHECK (CITY IN ('BOMBAY','NASHIK','PUNE'))
);
```

Views :

Views are pseudo-tables. That is, they are not real tables; nevertheless appear as ordinary tables to SELECT. A view can even represent joined tables. Because views are assigned separate permissions, you can use them to restrict table access so that the users see only specific rows or columns of a table.

A view can contain all rows of a table or selected rows from one or more tables. A view can be created from one or many tables, which depends on the written PostgreSQL query to create a view.

Since views are not ordinary tables, you may not be able to execute a DELETE, INSERT, or UPDATE statement on a view.

Creating Views :

The PostgreSQL views are created using the CREATE VIEW statement. The PostgreSQL views can be created from a single table, multiple tables, or another view.

The syntax for CREATE VIEW is as follows −

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

Multiple tables can be used in SELECT statement for creating view based on it. View can be created in the temporary space by using the optional TEMP or TEMPORARY keyword. Such types of temporary views are automatically dropped at the end of the current session.

Example :

In this example a view is created on table EMP which has five columns as EMP_ID, EMP_NAME, DEPT_NO, and SALARY. A view is created to work with only two columns EMP_ID and EMP_NAME of EMP table.

```
CREATE VIEW EMP_VIEW AS
SELECT EMP_ID, EMP_NAME FROM EMP;
```

View can be created from multiple tables as follows,

```
CREATE VIEW EMP_VIEW AS
SELECT E.EMP_ID, E.EMP_NAME,D.DEPT_NAME FROM EMP E,DEPT D
WHERE E.DEPT_NO=D.DEPT_NO;
```

You can now query EMP_VIEW as though it were a table EMP.

```
SELECT * FROM EMP_VIEW;
```

Alter Views

By using alter view statement you can change the definition of existing view or can rename it.

Example

```
ALTER VIEW EMP_VIEW RENAME TO EMPV;
```

CREATE TABLE LOG (id int, ts timestamptz);

CREATE VIEW log_view AS SELECT * FROM LOG;

ALTER VIEW log_view ALTER COLUMN ts SET DEFAULT now();

Dropping Views

To drop a view the statement DROP VIEW with the view_name can be used. The syntax for DROP VIEW is as follows,

```
DROP VIEW view_name;
```

Example

```
DROP VIEW EMP_VIEW;
```

Joins

The PostgreSQL Joins clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Join Types in PostgreSQL are −

- The CROSS JOIN
- The INNER JOIN
- The OUTER JOIN
- The SELF JOIN

The CROSS JOIN

A CROSS JOIN matches every row of the first table with every row of the second table. If the input tables have x and y columns, respectively, the resulting table will have x+y columns. The syntax for CROSS JOIN is as follows,

SELECT ... FROM table1 CROSS JOIN table2

Example

Consider the two tables CUSTOMER AND ORDER.

CUSTOMER			**ORDER**		
CUST_NO	**NAME**		**ORDER_NO**	**PRODUCT**	**CUST_NO**
1	STEPHEN		220	KEYBOARD	1
2	JOHN		365	PRINTER	3
3	ELON		512	SCANNER	3
4	STEVE				

```
SELECT * FROM CUSTOMER CROSS JOIN ORDER;
```

The above given query will produce the following result as,

CUST_NO	**NAME**	**ORDER_NO**	**PRODUCT**	**CUST_NO**
1	STEPHEN	220	KEYBOARD	1

1	STEPHEN	365	PRINTER	3
1	STEPHEN	512	SCANNER	3
2	JOHN	220	KEYBOARD	1
2	JOHN	365	PRINTER	3
2	JOHN	512	SCANNER	3
3	ELON	220	KEYBOARD	1
3	ELON	365	PRINTER	3
3	ELON	512	SCANNER	3
4	STEVE	220	KEYBOARD	1
4	STEVE	365	PRINTER	3
4	STEVE	512	SCANNER	3

The INNER JOIN :

INNER JOIN returns all rows from both tables where there is a match. If there are rows in CUSTOMER that does not have matches in ORDER, those rows will not be a part of output select statement using the inner join. This is the default type of join. Hence, it is optional to use INNER keyword.

The syntax of INNER JOIN is as following,

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example :

```
SELECT CUSTOMER.NAME, ORDER.PRODUCT FROM CUSTOMER INNER JOIN ORDER
ON CUSTOMER.CUST_NO=ORDER.CUST_NO;
```

The above given query will produce the following result,

NAME	**PRODUCT**
STEPHEN	KEYBOARD
ELON	PRINTER
ELON	SCANNER

The LEFT OUTER JOIN :

The LEFT OUTER JOIN returns all the rows from the first table (CUSTOMER), even if there are no matches in the second table (ORDER). If there are rows in CUSTOMER that do not have matches in ORDER, those rows also will be displayed with NULL values.

Example :

```
SELECT CUSTOMER.NAME, ORDER.PRODUCT
FROM CUSTOMER LEFT OUTER JOIN ORDER
ON CUSTOMER.CUST_NO=ORDER.CUST_NO;
```

The above given query will produce the following result,

NAME	**PRODUCT**

STEPHEN	KEYBOARD
JOHN	[null]
ELON	PRINTER
ELON	SCANNER
STEVE	[null]

The RIGHT OUTER JOIN :

The RIGHT OUTER JOIN returns all the rows from the second table (ORDER), even if there are no matches in the first table(CUSTOMER). If there are rows in ORDER that do not have matches in CUSTOMER, those rows also will be displayed.

Example :

```
SELECT CUSTOMER.NAME, ORDER.PRODUCT
FROM CUSTOMER RIGHT OUTER JOIN ORDER
ON CUSTOMER.CUST_NO=ORDER.CUST_NO;
```

The above given query will produce the following result,

NAME	**PRODUCT**
STEPHEN	KEYBOARD
ELON	PRINTER
ELON	SCANNER

The FULL OUTER JOIN :

The FULL OUTER JOIN keyword returns all records when there is a match in the first table (CUSTOMER) or second table (ORDER) records.

```
SELECT CUSTOMER.NAME, ORDER.ORDER_NO, ORDER.PRODUCT
FROM CUSTOMER FULL OUTER JOIN ORDER
ON CUSTOMER.CUST_NO=ORDER.CUST_NO;
```

The above given query will produce the following result as,

NAME	**ORDER_NO**	**PRODUCT**
STEPHEN	220	KEYBOARD
JOHN	[null]	[null]
ELON	365	PRINTER
ELON	512	SCANNER
STEVE	[null]	[null]

Subqueries :

A query within another query is known as subquery. Subquery is specified in the where clause of another query which is known as outer query or main query. The subquery executes first, and its output is used to complete the query condition for the main or outer query. Subquery must be enclosed in parentheses and always be on the right side of the comparison operator.

The Subquery can be place in a number of SQL clauses such as WHERE clause, HAVING clause, FROM clause. It can be used with SELECT, UPDATE, INSERT, DELETE statements along equality operator or comparison operator including =, >, =, <= and Like operator.

Self-Join:-

A self-join is used to join a table to itself. A self-join simplifies the task of nested queries in which the inner and outer queries refer to the same table. You should use a self-join only when you need to retrieve data that joins rows in the table with the rows in the reference of same table twice within th'
e same query, you must provide an alias to at least one of the instances of the table.

Syntax

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and *T2* are different table aliases for the same table.

Consider the following table

empid	ename	emp_mgr_id
1	RAJAN	2
2	SUSHIL	NULL
3	SHOBHA	5
4	RADHA	2
5	MILAN	NULL

By using self join we have to display empname and their manager name, by using above syntax of self join, the SQL is as follows

```
SELECT E1.ENAME AS "Manager Name" ,E2.ENAME AS "Emp Name"
FROM EMPSELF E1 JOIN EMPSELF E2
ON E1.EMPID=E2.EMP_MGR_ID
```

The output is

Manager Name	Emp Name
SUSHIL	RAJAN
SUSHIL	RADHA
MILAN	SHOBHA

Subqueries as Constants :

A subquery, also called a subselect, can replace a constant in a query. While constant never changes, a subquery's value are computed every time the query is executed.

Example : Consider the following tables.

CUSTOMER :

CUST_NO	NAME	CITY
1	STEPHEN	BOSTON
2	JOHN	HISTON
3	ELON	TEXAS
4	STEVE	TEXAS

If you want to find the customer who are not in the same city as 'ELON' You can place his city in the query using the constant string 'TEXAS' like,

```
    SELECT name, city FROM CUSTOMER
    WHERE city != 'TEXAS';
```

But if 'ELON' moves to another city then you have to change the query. Hence using the column city is more reliable.

```
    SELECT name,city FROM CUSTOMER
    WHERE city != (SELECT city FROM CUSTOMER WHERE name='ELON');
```

Whenever computed value is needed you can use sub queries. Each subquery has its own FROM and WHERE clauses also. It can also have its own aggregate, GROUP BY, and HAVING clauses. A subquery always returns the value to its outer. Using this approach comparisons can be done effectively, that would be difficult if the subquery's clauses had to be combined with those of the outer query.

Subqueries as Correlated Values :

In a correlated subquery, a table in the inner SELECT will be joined to a table in the outer SELECT, thereby defining a relationship between these two queries. This is a powerful group of subqueries. A correlated subquery is evaluated once for each row processed by the parent statement.

Syntax of correlated query,

```
    SELECT columnA from table1 T1
    WHERE T1.columnB =(SELECT T2.columnB FROM table2 T2
    WHERE T2.columnC = T1.columnC);
```

Example : Consider following two tables

CUSTOMER

CUST_NO	NAME	CITY
1	STEPHEN	BOSTON
2	JOHN	HISTON

3	ELON	TEXAS
4	STEVE	TEXAS

ORDER

ORDER_NO	PRODUCT	CUST_NO	QTY
220	KEYBOARD	1	5
365	PRINTER	3	4
512	SCANNER	3	2
220	SCANNER	1	5

The following correlated query display the values of CUST_NO, PRODUCT and QTY of the customer who lives in Boston.

```
SELECT PRODUCT,QTY FROM ORDER
WHERE ORDER.CUST_NO=(SELECT CUST_NO FROM CUSTOMER WHERE
CITY='BOSTON' );
```

Result :

PRODUCT	QTY
KEYBOARD	5
SCANNER	5

Subqueries as Lists of Values :

The subqueries in above example returned one row of data to the outer query. If any of the previous subqueries returned more than one row, an error would be generated: ERROR: More than one tuple returned by a subselect used as an expression. However, it is possible to have subquery which return multiple rows.

The comparison operators like =, <, > expect a single value on the left and on the right.

For example, equality expects one value on the left of the equals sign (=) and one on the right.

For example col1=5. There are two special comparisons, IN and NOT IN which allow multiple values to appear on the right side. For example, the condition quantity IN (2, 4, 5) compares col against three values.

If quantity equals any of the three values, the comparison will return true and output the row. The condition quantity NOT IN (2, 4, 5) will return true if quantity does not equal any of the three values. You can specify any number of values on the right side of an IN or NOT IN comparison. More importantly, a subquery (instead of a constant) can be placed on the right side. It can then return multiple rows. The subquery is evaluated, and its output used likes a list of constant values.

NOT IN and Subqueries with NULL Values :

When the value return by NOT IN subquery is NULL, the NOT IN comparison always returns false. NOT IN requires the outer column to be not equal to every value returned by the subquery. Because all comparisons with NULL return false even inequality comparisons NOT IN returns false.

We can prevent NULL values from reaching the outer query by adding IS NOT NULL to the subquery.

For example if any NULL cust_no value exists the query would return no rows. We can prevent this situation by adding WHERE cust_no IS NOT NULL to the subquery. The subquery using IN does not have the problem with NULLs.

Examples :

```
SELECT DISTINCT customer1.name
FROM customer1, order1
WHERE customer1.cust_no=order1.cust_no and customer1.city='BOSTON';

SELECT name
FROM customer1
WHERE cust_no NOT IN ( SELECT cust_no FROM order1 WHERE product='SCANNER');

SELECT name
FROM customer1
WHERE cust_no IN ( SELECT cust_no FROM order1 WHERE product='SCANNER');
```

Subqueries Returning Multiple Columns :

Generally most subqueries return a single column to the outer query, but you can use subqueries for returning more than one column.

For example, SELECT CUST_NO, NAME FROM CUSTOMER WHERE (1, 5) IN (SELECT CUST_NO, QTY FROM ORDER)

This query returns true if the subquery returns a row with 1 in the first column and 5 in the second column. The condition WHERE (outercol1, outercol2) IN (SELECT col1, col2 FROM subtable) performs equality comparisons between the outer query's two columns and the subquery's two columns. Multiple columns in the outer query can then be compared with multiple columns in the subquery. While using such type of subqueries , the number of values given on the left of IN or NOT IN must be the same as the number of columns returned by the subquery.

MERGE statement :

The MERGE statement in PostgreSQL is used to merge data from a source table to a target table based on a specified condition. The MERGE command in SQL is a combination of three SQL statements: INSERT, UPDATE and DELETE. Basically merge statement is used if you want to make changes in the required target table with the help of provided source table which consists of latest details.

Consider the example, if you have two tables PRODUCT table which contains the current details of the products like PID, NAME, and PRICE and NEW_PRODUCT table that contains the new details of the products ID, NAME, and PRICE and you have to update the details of the products in the PRODUCT table as per the NEW_PRODUCT.

PRODUCT :

ID	NAME	PRICE
101	KEYBOARD	400
102	MOUSE	150

103	PENDRIVE	600

NEW_PRODUCT :

ID	NAME	PRICE
101	KEYBOARD	400
102	MOUSE	200
104	LED PEN	100

Syntax :

```
MERGE INTO table [ [ AS ] alias ]
USING source-query
ON join_condition
[when_clause [...]]
/* Where when_clause is */
{ WHEN MATCHED [ AND condition ] THEN { merge_update | DELETE }
 WHEN NOT MATCHED [ AND condition ] THEN { merge_insert | DO NOTHING } }
/* Where merge_update is */
UPDATE SET { column = { expression | DEFAULT } |
  ( column [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
/* Where merge_insert is */
INSERT [( column [, ...] )] { VALUES ( { expression | DEFAULT } [, ...] ) | DEFAULT VALUES }
```

Example :

```
MERGE INTO PRODUCT USING NEW_PRODUCT
ON PRODUCT.ID=NEW_PRODUCT.ID
WHEN MATCHED THEN UPDATE SET PRICE=NEW_PRODUCT.PRICE
WHEN NOT MATCHED THEN INSERT VALUES
(NEW_PRODUCT.ID,NEW_PRODUCT.NAME,NEW_PRODUCT.PRICE);
```

Set Operations :

1. UNION :

The PostgreSQL UNION operator combines the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order but they do not have to be the same length.

Syntax :

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION
```

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Example

Consider two tables DEPT and EMP

DEPT

DEPT_NO	DEPT_NAME
1	IT
2	FINANCE
3	HR
4	SALES

EMP

EMP_ID	NAME	DEPT_NO	SALARY
101	RAM PATIL	1	65000
102	SHITAL VARMA	1	70000
103	HEMANT RAO	2	45000
104	SUNDAR REDDY	3	80000
105	NISHANT GOYAL	2	50000
106	VIREN GHOSH	3	57000
107	MANJIT SINGH	3	40000

```
SELECT DEPT_NO FROM DEPT
   UNION
   SELECT DEPT_NO FROM EMP ;
```

This will return output as,

DEPT_NO
2
4
3
1

2. EXCEPT :

The EXCEPT operator returns distinct rows from the first (left) query that are not in the output of the second (right) query.

To use EXCEPT, operator the number of columns and their orders must be the same in the two queries also the data types of the respective columns must be compatible.

Syntax :

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
```

```
[WHERE condition]
EXCEPT
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Example

```
SELECT DEPT_NO FROM DEPT
    EXCEPT
    SELECT DEPT_NO FROM EMP ;
```

This will return output as,

DEPT_NO
4

3. INTERSECT :

The INTERSECT operator returns the common data among both the tables in a single result set. As like UNION and EXCEPT to use INTERSECT operator the number of columns and their orders must be the same in the two queries also the data types of the respective columns must be compatible.

Syntax :

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

INTERSECT
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Example

```
SELECT DEPT_NO FROM DEPT
    INTERSECT
    SELECT DEPT_NO FROM EMP ;
```

his will return output as,

DEPT_NO
3
2
1

Clauses -ANY, ALL, and EXISTS Clauses

4. ANY :

The ANY operator returns true if any of the subquery values meet the condition To select any tuples of SELECT STATEMENT the ANY operator is used. Also it is used to compare a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row.

Always the comparison operators like <,>,<=,>=,= is followed by ANY operator .You can use any with SELECT, WHERE and HAVING statement.

Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
(SELECT column_name FROM table_name WHERE condition);
```

Example :

Consider the above EMP table. The below query return the records of employee whose salary is greater than any of the employee's salary from department 2.

```
SELECT EMP_NAME, SALARY
FROM EMP
WHERE SALARY > Any (SELECT SALARY
FROM EMP
WHERE DEPT_NO = 2);
```

5. ALL :

The ALL operator returns true if all of the subquery values meet the condition. To select all tuples of SELECT STATEMENT the ALL operator is used. Also it is used to compare a value to every value in another value list or result from a subquery.

The ALL operator returns TRUE when all of the subqueries values meet the condition. The comparison operators like <,>,<=,>=,= are used with ALL operator .You can use ALL with SELECT, WHERE and HAVING statement.

Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
(SELECT column_name FROM table_name WHERE condition);
```

Example :

Consider the above EMP table. The below query return the records of employee whose salary is greater than all of the employee's salary from department 2.

```
SELECT EMP_NAME, SALARY
FROM EMP
WHERE SALARY > ALL (SELECT SALARY
FROM EMP
WHERE DEPT_NO = 2);
```

6. EXISTS :

The EXISTS condition in PostgreSQL is used to check whether the result of a correlated nested query contains any tuples or not. The result of EXISTS is always a Boolean type of vale(True or False). The EXISTS operator can be used in a SELECT, UPDATE, INSERT or DELETE statement to specify existence condition.

Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name(s)
FROM table_name
WHERE condition);
```

Example :

Consider the above EMP and DEPT table. The following query returns the names of department in which at least one employee exist.

```
Select dept_name
FROM dept
WHERE EXISTS (SELECT * FROM emp
WHERE emp.dept_no=dept.dept_no);
```

Unit - 5 TRANSACTION COMMANDS , INDEX AND SEQUENCE [L : 5 M: 10]

• Transaction commands-Commit, Rollback • Indexing -Creating an Index, Unique Indexes • Sequences- Creating Sequence, using nextval(), currval() and setval()

- **Transaction Commands (Commit, Rollback)**

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, Creating, Updating and Deleting Record from table is the transaction on table. , then you are performing transaction on the table. It is important to control transactions to ensure data integrity and to handle database errors.

Transaction Control

The following commands are used to control transactions −

BEGIN TRANSACTION − To start a transaction.

COMMIT − To save the changes, alternatively you can use END TRANSACTION command.

ROLLBACK − To rollback the changes.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The BEGIN TRANSACTION Command

Transactions can be started using BEGIN TRANSACTION or simply BEGIN command. Such transactions usually persist until the next COMMIT or ROLLBACK command is encountered. But a transaction will also ROLLBACK if the database is closed or if an error occurs.

The following is the simple syntax to start a transaction −

BEGIN;

or

BEGIN TRANSACTION;

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for COMMIT command is as follows −

```
COMMIT;
```

or

```
END TRANSACTION;
```

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for ROLLBACK command is as follows −

```
ROLLBACK;
```

Example

Consider the EMPLOYEE table is having the following records −

empid	ename	designation	salary
1	RAMA PATIL	ANALYST	34500
2	NITIL DHANDE	PROGRAMMER	54000
3	SHIRISH ZALTE	DEVELOPER	46000

Now, let us start a transaction and delete records from the table having empid = 3 and finally we use ROLLBACK command to undo all the changes.

```
BEGIN;
DELETE FROM EMPLOYEE WHERE empid = 3;
ROLLBACK;
```

If you will check EMPLOYEE table is still having the following records −

empid	ename	designation	salary
1	RAMA PATIL	ANALYST	34500
2	NITIL DHANDE	PROGRAMMER	54000
3	SHIRISH ZALTE	DEVELOPER	46000

Now, let us start another transaction and delete records from the table having empid = 3 and finally we use COMMIT command to commit all the changes.

BEGIN;

DELETE FROM COMPANY WHERE emipd = 3;

COMMIT;

If you will check the EMPLOYEE table, it still has the following records −

empid	ename	designation	salary
1	RAMA PATIL	ANALYSY	34500
2	NITIL DHANDE	PROGRAMMER	54000

Practically, combining many PostgreSQL queries into a group and you will execute all of them together as a part of a transaction.

- INDEXING

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you have to first refer to the index, which lists all topics alphabetically and then refer to one or more specific page numbers.

An index helps to speed up SELECT queries and WHERE clauses; however, it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Creating an INDEX

Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

The CREATE INDEX Command

The basic syntax of **CREATE INDEX** is as follows −

CREATE INDEX index_name ON table_name;

Index Types

PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST and GIN. Each Index type uses a different algorithm that is best suited to different types of queries. By default, the CREATE INDEX command creates B-tree indexes, which fit the most common situations.

Single-Column Indexes

A single-column index is one that is created based on only one table column. The basic syntax is as follows −

CREATE INDEX index_name
ON table_name (column_name);

Example:

Create simple table and insert records , more records can generates better result so at least insert 10 to 15 records,

```
CREATE TABLE EMPLOYEE
(EMPID INTEGER PRIMARY KEY,
ENAME VARCHAR(20),
DESIGNATION VARCHAR(20),
SALARY MONEY,
PHONENO INTEGER)
```

```
INSERT INTO EMPLOYEE
VALUES(156,'RAMDAS PATIL','PROGRAMMER',56000,561234),
        (90,'PRAJKTA BHMRE','ANALYST',67000,876541),
        (542,'SUJIT TAMBE','SOFTWARE ENGINEER',123000,967890),
        (123,'VINAY CHAUDHARI','PROGRAMMER',54390,784996)
```

Now , search the record from employee table , as if we search phoneno=876541 as

```
SELECT *
FROM EMPLOYEE
WHERE PHONENO=876541
```

```
Successfully run. Total query runtime: 429 msec. 1 rows affected.
```

Create Index on the Employee table on Phoneno column as

```
CREATE INDEX empphone_index
ON EMPLOYEE(PHONENO)
```

Search the same record , it requires less time as compare non-index table

```
SELECT *
FROM EMPLOYEE
WHERE PHONENO=876541
```

```
Successfully run. Total query runtime: 76 msec. 1 rows affected.
```

Multicolumn Indexes

A multicolumn index is defined on more than one column of a table. The basic syntax is as follows

```
CREATE INDEX index_name
ON table_name (column1_name, column2_name);
```

```
CREATE INDEX empphid_index
ON EMPLOYEE(PHONENO,EMPID)
```

Whether to create a single-column index or a multicolumn index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the multicolumn index would be the best choice.

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows −

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

```
CREATE TABLE ITEM
 (ITEMNO INT,
 INAME VARCHAR(20),
 RATE MONEY,
 DTOFPURCHASE DATE,
 QUANTITY INT)
```

```
INSERT INTO ITEM
VALUES(122,'CHAIR',2500,'12-SEP-2020',2)
```

```
INSERT INTO ITEM
VALUES(122,'TABLE',8500,'22-SEP-2020',2)
```

```
ERROR: duplicate key value violates unique constraint "uind" DETAIL: Key (itemno)=(122)
already exists. SQL state: 23505
```

- **Sequences- Creating Sequence, using nextval(), currval() and setval()**

By definition, a sequence is an ordered list of integers. The orders of numbers in the sequence are important. For example, {1,2,3,4,5} and {5,4,3,2,1} are entirely different sequences.
A sequence in PostgreSQL is a user-defined schema-bound object that generates a sequence of integers based on a specified specification.

To create a sequence in PostgreSQL, you use the CREATE SEQUENCE statement.
Introduction to PostgreSQL CREATE SEQUENCE statement
The following illustrates the syntax of the CREATE SEQUENCE statement:

```
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 } ]
```

sequence_name

sequence_name

Specify the name of the sequence after the CREATE SEQUENCE clause. The IF NOT EXISTS conditionally creates a new sequence only if it does not exist.
The sequence name must be distinct from any other sequences, tables, indexes, views, or foreign tables in the same schema.

[AS { SMALLINT | INT | BIGINT }]

Specify the data type of the sequence. The valid data type is SMALLINT, INT, and BIGINT.
The default data type is BIGINT if you skip it.
The data type of the sequence which determines the sequence's minimum and maximum values.

[INCREMENT [BY] increment]

The increment specifies which value to be added to the current sequence value to create new value.
A positive number will make an ascending sequence while a negative number will form a descending sequence.

The default increment value is 1.

[MINVALUE minvalue | NO MINVALUE]

[MAXVALUE maxvalue | NO MAXVALUE]

Define the minimum value and maximum value of the sequence. If you use NO MINVALUEand NO MAXVALUE, the sequence will use the default value.
For an ascending sequence, the default maximum value is the maximum value of the data type of the sequence and the default minimum value is 1.

In case of a descending sequence, the default maximum value is -1 and the default minimum value is the minimum value of the data type of the sequence.

[START [WITH] start]

The START clause specifies the starting value of the sequence.
The default starting value is minvalue for ascending sequences and maxvalue for descending ones.

cache

The CACHE determines how many sequence numbers are preallocated and stored in memory for faster access. One value can be generated at a time.
By default, the sequence generates one value at a time i.e., no cache.

CYCLE | NO CYCLE

The CYCLE allows you to restart the value if the limit is reached. The next number will be the minimum value for the ascending sequence and maximum value for the descending sequence.
If you use NO CYCLE, when the limit is reached, attempting to get the next value will result in an error.
The NO CYCLE is the default if you don't explicitly specify CYCLE or NO CYCLE.

OWNED BY table_name.column_name

The OWNED BY clause allows you to associate the table column with the sequence so that when you drop the column or table, PostgreSQL will automatically drop the associated sequence.
Note that when you use the SERIAL pseudo-type for a column of a table, behind the scenes, PostgreSQL automatically creates a sequence associated with the column.

PostgreSQL CREATE SEQUENCE examples

Let's take some examples of creating sequences to get a better understanding.

1) Creating an ascending sequence example

This statement uses the CREATE SEQUENCE statement to create a new ascending sequence starting from 100 with an increment of 5:

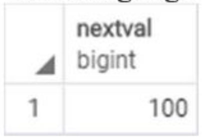
```
CREATE SEQUENCE mysequence
INCREMENT 5
START 100;
```

Code language: SQL (Structured Query Language) (sql)

To get the next value from the sequence to you use the nextval() function:

```
SELECT nextval('mysequence');
```

Code language: SQL (Structured Query Language) (sql)



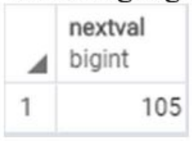
	nextval bigint
1	100

NEXTVAL()

If you execute the statement again, you will get the next value from the sequence:

```
SELECT nextval('mysequence');
```

Code language: SQL (Structured Query Language) (sql)



	nextval bigint
1	105

2) Creating a descending sequence example

The following statement creates a descending sequence from 3 to 1 with the cycle option:

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

When you execute the following statement multiple times, you will see the number starting from 3, 2, 1 and back to 3, 2, 1 and so on:

```
SELECT NEXTVAL('three')
```

3) Creating a sequence associated with a table column

First, create a new table named order_details:

```
 CREATE TABLE order_details(
   order_id SERIAL,
   item_id INT NOT NULL,
   item_name VARCHAR NOT NULL,
   price DEC(10,2) NOT NULL,
   PRIMARY KEY(order_id, item_id)
);
```

Second, create a new sequence associated with the item_id column of the order_details table:

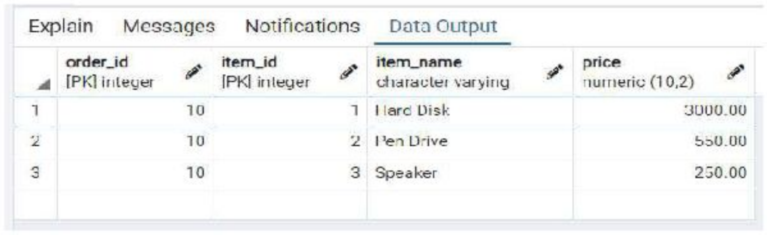
```
CREATE SEQUENCE ord_itemid
START 1
INCREMENT 1
MINVALUE 1
OWNED BY order_details.item_id;
```

Third, insert three order line items into the order_details table:

```
 INSERT INTO
   order_details(order_id, item_id, item_name, price)
VALUES
   (10, nextval('ord_itemid'),'Hard Disk',3000),
   (10, nextval('ord_itemid'),'Pen Drive ',550),
   (10, nextval('ord_itemid'),'Speaker',250);
```

In this statement, we used the nextval() function to fetch item id value from the order_item_id sequence.
Fourth, query data from the order_details table:

```
SELECT
   order_id,
   item_id,
   item_name,
   price
FROM
   order_details;
```



	order_id [PK] integer	item_id [PK] integer	item_name character varying	price numeric (10,2)
1	10	1	Hard Disk	3000.00
2	10	2	Pen Drive	550.00
3	10	3	Speaker	250.00

Listing all sequences in a database

To list all sequences in the current database, you use the following query:

```
SELECT
   relname sequence_name
FROM
   pg_class
WHERE
   relkind = 'S';
```
Code language: SQL (Structured Query Language) (sql)

CURRVAL()

Syntax:

Currval(sequence_name) – Display Current value of sequence

```
CREATE SEQUENCE incre
INCREMENT 1
MINVALUE 1
MAXVALUE 5
START 1
CYCLE;
```

The above sequence generate 1,2,3,4,5 and again same cycle multiple times by following command.

```
SELECT NEXTVAL('INCRE')
```

At the time of execution of Nextval() , in between when we stop and want to display current value , If sequence generates 1,2,3 only and stops then execute

```
SELECT CURRVAL('incre')
```

Displays 3 , is the current value of sequence

Deleting sequences

If a sequence is associated with a table column, it will be automatically dropped once the table column is removed or the table is dropped.

You can also remove a sequence manually using the DROP SEQUENCE statement:

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

Unit – 6 - PL/PGSQL - SQL PROCEDURAL LANGUAGE [L : 15 M: 20]

• Introduction to PL/PGSQL-Advantages of PL/PGSQL, structure of PL/PGSQL, basic Statements and control structures • Function -Creating functions, Removing functions • Cursors- Creation of Cursors, Using Cursors, Looping • Triggers-Introduction, Triggers Vs constraints, DML Triggers, DDL Triggers 7 Under • Error handling -Introduction Error Handling, RAISE Statement

- **Introduction of PostgreSQL PL/pgSQL**

PL/pgSQL is a procedural programming language for the PostgreSQL database system.

PL/pgSQL allows you to extend the functionality of the PostgreSQL database server by creating server objects with complex logic.

PL/pgSQL was designed to :

- Create user-defined functions, stored procedures, and triggers.
- Extend standard SQL by adding control structures such as if, case, and loop statements.
- Inherit all user-defined functions, operators, and types.

SQL is a query language that allows you to query data from the database easily. However, PostgreSQL only can execute SQL statements individually.

It means that you have multiple statements, you need to execute them one by one like this:

- First, send a query to the PostgreSQL database server.
- Next, wait for it to process.
- Then, process the result set.
- After that, do some calculations.
- Finally, send another query to the PostgreSQL database server and repeat this process.

This process incurs the interprocess commmunication and network overheads.

To resolve this issue, PostgreSQL uses PL/pgSQL.

PL/pgSQL wraps multiple statements in an object and store it on the PostgreSQL database server.

So instead of sending multiple statements to the server one by one, you can send one statement to execute the object stored in the server. This allows you to:

- Reduce the number of round trips between the application and the PostgreSQL database server.
- Avoid transferring the immediate results between the application and the server.

- **Advantages of using PL/pgSQL**

1. Open Source DBMS

Only PostgreSQL provides enterprise-class performance and functions among current Open Source DBMS with no end of development possibilities. Also, PostgreSQL users can directly participate in the community and post and share inconveniences and bugs.

2. Diverse Community

One of the characteristics of PostgreSQL is that there are a wide variety of communities. Regarding PostgreSQL as Open Source DBMS, users themselves can develop modules and propose the module to the community. The development possibility is superiorly high with collecting opinions from its own global community organized with all different kinds of people. Collective Intelligence, as some might call it, facilitates transmission of indigenous knowledge greatly within the communities.

3. Function

SQL functions called 'Store Procedure' can be used for server environment. Also, we support languages similar to PL/SQL in Oralcle such as PL/pgSQL, PL/Python, PL/Perl, C/C++, and PL/R.

4. ACID and Transaction

PostgreSQL support ACID(Atomicity, Consistency, Isolation, Durability).

5. Diverse indexing techniques

PostgreSQL not only provides B+ tree index techniques, but various kinds of techniques such as GIN(Generalized Inverted Index), and GiST(Generalized Search Tree), etc as well.

6. Flexible Full-text search

Full-text search is available when searching for strings with execution of vector operation and string search.

7. Diverse kinds of replication

PostgreSQL supports a variety of replication methods such as Streaming Replication , Slony-I, and cascading.

8. Diversified extension functions

PostgreSQL supports different kinds of techniques for geographic data storage such as PostGIS, Key-Value Store, and DBLink.

- **Structure of PL/pgSQL**

PL/pgSQL is a block-structured language, therefore, a PL/pgSQL function or stored procedure is organized into blocks.

The following illustrates the syntax of a complete block in PL/pgSQL:

```
[ declare
   declarations ]
begin
   statements;
          ...
end [ label ];
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

Let's examine the block structure in more detail:

- Each block has two sections: declaration and body. The declaration section is optional while the body section is required. A block is ended with a semicolon (;) after the END keyword.
- A block may have an optional label located at the beginning and at the end. You use the block label when you want to specify it in the EXIT statement of the block body or when you want to qualify the names of variables declared in the block.
- The declaration section is where you declare all variables used within the body section. Each statement in the declaration section is terminated with a semicolon (;).
- The body section is where you place the code. Each statement in the body section is also terminated with a semicolon (;).

- **Basic Statement and Control Statement:**

Variables and Constants

All variables, rows and records used in a block or its sub-blocks must be declared in the declarations section of a block. The exception being the loop variable of a FOR loop iterating over a range of integer values.

PL/pgSQL variables can have any SQL datatype, such as INTEGER, VARCHAR and CHAR. All variables have as default value the SQL NULL value.

Here are some examples of variable declarations:

```
user_id INTEGER;

quantity NUMBER(5);

url VARCHAR;
```

Constants and Variables With Default Values

The declarations have the following syntax:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } value ];
```

The value of variables declared as CONSTANT cannot be changed. If NOT NULL is specified, an assignment of a NULL value results in a runtime error. Since the default value of all variables is the SQL NULL value, all variables declared as NOT NULL must also have a default value specified.

The default value is evaluated every time the function is called. So assigning 'now' to a variable of type timestamp causes the variable to have the time of the actual function call, not when the function was precompiled into its bytecode.

Examples:

```
quantity INTEGER := 35;

url varchar := ''http://basponc.org'';

u_id CONSTANT INTEGER := 38;
```

Parameters Passed to Functions

Parameters passed to functions are named with the identifiers $1, $2, etc. Optionally, aliases can be declared for the $n parameter names for increased readability. Some examples:

Attributes

Using the %TYPE and %ROWTYPE attributes, you can declare variables with the same datatype or structure of another database item (e.g: a table field).

variable%TYPE

%TYPE provides the datatype of a variable or database column. You can use this to declare variables that will hold database values. For example, let's say you have a column named emp_id in your employee table. To declare a variable with the same datatype as employee.emp_id you write:

```
emp_id   employee.emp_id%TYPE;
```

By using %TYPE you don't need to know the datatype of the structure you are referencing, and most important, if the datatype of the referenced item changes in the future (e.g: you change your table definition of emp_id to become a REAL), you won't need to change your function definition.

table%ROWTYPE

%ROWTYPE provides the composite datatype corresponding to a whole row of the specified table. table must be an existing table or view name of the database. The fields of the row are accessed in the dot notation. Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a rowtype, and fields can be selected from it, for example $1.user_id.

Only the user-defined attributes of a table row are accessible in a rowtype variable, not OID or other system attributes (because the row could be from a view). The fields of the rowtype inherit the table's field sizes or precision for char() etc. data types.

```
DECLARE

   emp_rec empoyee%ROWTYPE;

   emp_id employee.emp_id%TYPE;
```

The RAISE NOTICE is the equivalent to these in Postgres PL/pgSQL

, as shown in the following anonymous block:

```
do $$
 BEGIN
    raise notice 'Hello World!';
 END;
$$;
```

It prints:

NOTICE: Hello World!

Control Statement:

The if statement determines which statements to execute based on the result of a boolean expression.

PL/pgSQL provides you with three forms of the if statements.

if then , if then else, if then elsif

PL/pgSQL if-then statement

The following illustrates the formats of the if statement:

if-then	**If-then-else**	**If-then-elsif**

if condition then
  statements;
end if;
	if condition then
	  statements;
	else
	  alternative-statements;
	END if;
		if condition_1 then
		  statement_1;
		elsif condition_2 then
		  statement_2
		...
		elsif condition_n then
		  statement_n;
		else
		  else-statement;
		end if;

if statement executes statements if a condition is true. If the condition evaluates to false, the control is passed to the next statement after the END if part.

The condition is a boolean expression that evaluates to true or false.

The statements can be one or more statements that will be executed if the condition is true. It can be any valid statement, even another if statement.

When an if statement is placed inside another if statement, it is called a nested-if statement.

Example:

Program

```
do $$
declare
  selected_item item%rowtype;
  input_item_id item.item_id%type := 2;
begin

  select * from item
  into selected_item
  where item_id = input_item_id;

  if not found then
     raise notice 'The item % could not be
found',
                input_item_id;
  else
     raise notice 'The item_name is %',
selected_item.item_name;
  end if;
end $$
```

Description /Requirement

Item table contains
Item(Item_id,Ttem_name,rate,pdata)

Select * from Item

item_id	item_name	rate	purdate
1	pen	230	5/10/2019
2	box	300	5/10/2019
3	stapler	230	5/10/2019
4	paper-rim	430	5/10/2019

After exection the pl/pgSQL following is the output

```
NOTICE: The item_name is box DO
 Query returned successfully in 93 msec.
```

For Loop

The following illustrates the syntax of the for loop statement that loops through a range of integers:

[<<label>>]
for loop_counter in [reverse] from.. to [by step] loop
   statements
end loop [label];

In this syntax:

First, the for loop creates an integer variable loop_counter which is accessible inside the loop only. By default, the for loop adds the step to the loop_counter after each iteration. However, when you use the reverse option, the for loop subtracts the step from loop_counter.

Second, the from and to are expressions that specify the lower and upper bound of the range. The for loop evaluates these expressions before entering the loop.

Third, the step that follows the by keyword specifies the iteration step. It defaults to 1. The for loop evaluates this step expression once only.

```
do $$
 begin
 FOR i IN 1..5 LOOP
  -- some expressions here
    RAISE NOTICE 'i is %',i;
END LOOP;
end; $$

 Output:
NOTICE:  i is 1
NOTICE:  i is 2
NOTICE:  i is 3
NOTICE:  i is 4
NOTICE:  i is 5
```

```
do $$
BEGIN
FOR i IN REVERSE 5..1 LOOP
   -- some expressions here
         RAISE NOTICE 'i is %',i;
END LOOP;
END; $$

Output:
NOTICE:  i is 5
NOTICE:  i is 4
NOTICE:  i is 3
NOTICE:  i is 2
NOTICE:  i is 1
```

The following statement uses the for loop to display the titles of the top 10 longest films.

```
do
$$
declare
   f record;
begin
   for f in select title, length
               from film
               order by length desc, title
               limit 10
   loop
         raise notice '%(% mins)', f.title, f.length;
   end loop;
end;
$$
```

The WHILE loop

The WHILE loop is used to do the job repeatedly within the block of statements until the condition mentioned becomes false. In this type of loop the condition mentioned will be executed first before the statement block is executed.

Here is the syntax of the WHILE loop:

Syntax:

```
[ <<label>> ]
WHILE condition LOOP
statement;
[...]
END LOOP;
```

You can write a hello_world function as a named PL/pgSQL block:

```
CREATE FUNCTION hello_world()
RETURNS text AS
$$
DECLARE
  output  VARCHAR(20);
BEGIN
  /* Query the string into a local variable. */
```

```
    SELECT 'Hello World!' INTO output;

    /* Return the output text variable. */
    RETURN output;
   END
   $$ LANGUAGE plpgsql;
```

You can call it with the following:

```
SELECT hello_world();
```

It prints:

```
 hello_world
--------------
 Hello World!
(1 row)
```

- **Function (Stored Procedures) -Creating functions, Removing functions**

PostgreSQL **functions**, also known as **Stored Procedures**, allow you to carry out operations that would normally take several queries and round trips in a single function within the database. Functions allow database reuse as other applications can interact directly with your stored procedures instead of a middle-tier or duplicating code.

Functions can be created in a language of your choice like SQL, PL/pgSQL, C, Python, etc.

Introduction to PL/pgSQL parameter modes

The parameter modes determine the behaviors of parameters. PL/pgSQL supports three parameter modes: in, out, and inout. A parameter takes the in mode by default if you do not explicitly specify it.

The following table illustrates the three parameter modes:

IN	**OUT**	**INOUT**
The default	Explicitly specified	Explicitly specified
Pass a value to function	Return a value from a function	Pass a value to a function and return an updated value.
in parameters act like constants	out parameters act like uninitialized variables	inout parameters act like an initialized variables
Cannot be assigned a value	Must assign a value	Should be assigned a value

Syntax

The basic syntax to create a function is as follows −

```
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;
```

Where,

- **function-name** specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The function must contain a **return** statement.
- **RETURN** clause specifies that data type you are going to return from the function. The **return_datatype** can be a base, composite, or domain type, or can reference the type of a table column.
- **function-body** contains the executable part.
- The AS keyword is used for creating a standalone function.
- **plpgsql** is the name of the language that the function is implemented in. Here, we use this option for PostgreSQL, it Can be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name can be enclosed by single quotes.

Example

```
CREATE TABLE EMPLOYEE
(EMPID INTEGER,
 ENAME TEXT,
 DESIGNATION VARCHAR(20),
 SALARY DECIMAL(12,2))

INSERT INTO EMPLOYEE
VALUES(1,'RAMA PATIL','ANALYSY',34500),
          (2,'NITIL DHANDE','PROGRAMMER',54000),
          (3,'SHIRISH ZALTE','DEVELOPER',46000)
```

Example-1

```
CREATE OR REPLACE FUNCTION totalRecords()
RETURNS integer AS $total$
declare
        total integer;
BEGIN
  SELECT count(*) INTO total
  FROM EMPLOYEE;
  RETURN total;
END;
$total$ LANGUAGE plpgsql;
---
SELECT totalRecords()
```

Example-2

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
     BEGIN
          RETURN i + 1;
     END;
$$ LANGUAGE plpgsql;

SELECT increment(4) --- Output: 5
```

Example-3

```
create or replace function swap(
        inout x int,
        inout y int
)
language plpgsql
as $$
begin
  select x,y into y,x;
end; $$;

select swap(20,30) --  output (30,20)
```

- **Trigger**

What are Postgresql Triggers?

A PostgreSQL trigger is a function that is triggered automatically when a database event occurs on a database object. For example, a table.

Examples of database events that can activate a trigger include INSERT, UPDATE, DELETE, etc. Moreover, when you create a trigger for a table, the trigger will be dropped automatically when that To create a new trigger in PostgreSQL, you follow these steps:

- First, create a trigger function using CREATE FUNCTION statement.
- Second, bind the trigger function to a table by using CREATE TRIGGER statement. table is deleted.

Create trigger function syntax

A trigger function is similar to a regular user-defined function. However, a trigger function does not take any arguments and has a return value with the type trigger.

The following illustrates the syntax of creating trigger function:

```
CREATE FUNCTION trigger_function()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
AS $$
BEGIN
  -- trigger logic
END;
$$
```

Notice that you can create a trigger function using any languages supported by PostgreSQL. In this tutorial, we will use PL/pgSQL.

A trigger function receives data about its calling environment through a special structure called TriggerData which contains a set of local variables.

For example, OLD and NEW represent the states of the row in the table before or after the triggering event.

Once you define a trigger function, you can bind it to one or more trigger events such as INSERT, UPDATE, and DELETE. The CREATE TRIGGER statement creates a new trigger. The following illustrates the basic syntax of the CREATE TRIGGER statement:

CREATE TRIGGER

```
CREATE TRIGGER trigger_name
  {BEFORE | AFTER} { event }
  ON table_name
  [FOR [EACH] { ROW | STATEMENT }]
     EXECUTE PROCEDURE trigger_function
```

In this syntax we must go through following steps:

1. Specify the name of the trigger after the TRIGGER keywords.
2. Specify the timing that cause the trigger to fire. It can be BEFORE or AFTER an event occurs.
3. Specify the event that invokes the trigger. The event can be INSERT , DELETE, UPDATE or TRUNCATE.
4. Specify the name of the table associated with the trigger after the ON keyword.
5. Specify the type of triggers which can be:

Row-level trigger that is specified by the FOR EACH ROW clause.

Statement-level trigger that is specified by the FOR EACH STATEMENT clause.

A row-level trigger is fired for each row while a statement-level trigger is fired for each transaction.

Example:

In the following table we insert information of the employees.

```
CREATE TABLE employees(
  id INT GENERATED ALWAYS AS IDENTITY,
  first_name VARCHAR(40) NOT NULL,
  last_name VARCHAR(40) NOT NULL,
  PRIMARY KEY(id)
);
```

Following table is created for storing updated value of last_name of employee .

```
CREATE TABLE employee_change (
  id INT GENERATED ALWAYS AS IDENTITY,
  employee_id INT NOT NULL,
  last_name VARCHAR(40) NOT NULL,
  changed_on TIMESTAMP(6) NOT NULL
);


CREATE OR REPLACE FUNCTION last_name_changes()
 RETURNS TRIGGER
 LANGUAGE PLPGSQL
 AS
$$
BEGIN
        IF NEW.last_name <> OLD.last_name THEN
                 INSERT INTO employee_change(employee_id,last_name,changed_on)
```

```
                VALUES(OLD.id,OLD.last_name,now());
        END IF;
        RETURN NEW;
END;
$$
```

The OLD represents the row before update while the NEW represents the new row that will be updated. The OLD.last_name returns the last name before the update and the NEW.last_name returns the new last name.

Creating Trigger

```
 CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE last_name_changes();
```

Insert Values in the Employees table

```
 INSERT INTO employees(first_name,last_name)
  values('Manish','Nimbhore'),
          ('Rahul','Borse'),
         ('Sanjay','Tripathi');
```

select * from employees

id	first_name	last_name
1	Manish	Nimbhore
3	Sanjay	Tripathi
2	Rahul	Borse

Now, update the last_name of any employee

```
UPDATE employees
SET last_name='Chandole'
WHERE id=2;
```

After update , the change will occur in both of the tables, ‘Borse’ replaces by ‘Chandole’

select * from employees

id	first_name	last_name
1	Manish	Nimbhore

3	Sanjay	Tripathi
2	Rahul	Chandole

Updated employee value stored in employee_change table

select * from employee_change

id	employee_	last_name	changed_on	
1	2	Borse	42:49.8	

- **Cursor**

A **cursor** is a temporary work area created in the system memory when a SQL statement is executed. A **cursor** contains information on a select statement and the rows of data accessed by it. ... A **cursor** can hold more than one row, but can process only one row at a time.

There are four steps in the lifecycle of a cursor:

Declare

The Declare step of a cursor is where you specify the name of the cursor and the SQL statement that is used to populate it.

Put the CURSOR keyword followed by a list of comma-separated arguments (name datatype) that defines parameters for the query. These arguments will be substituted by values when the cursor is opened.

After that, you specify a query following the FOR keyword. You can use any valid SELECT statement here.

The following example illustrates how to declare cursors:

```
declare
    cursorname  CURSOR  FOR
          SELECT ststement
```

Open

The next step is Open, which processes and runs the SQL statement that is mentioned in the Declare section.

```
OPEN cursorname;
```

Fetch

The third step is Fetch, which reads a single row from the set of rows stored in the cursor and stores this single row into another variable.

When you fetch the row, you can perform actions and logic on the data in the row. You can modify other variables, run SQL commands, perform IF statements, and more.

The Fetch step is usually run on each row in the overall result.

```
FETCH cursorname INTO variable1,variable2;
```

Close

Finally, once all of the results have been processed and the Fetch stage is finished, the Close step will release the cursor from memory and allow you to continue with the application.

```
CREATE TABLE product (
  id INT,
  product_name VARCHAR(100),
  product_category_id INT,
  price INT
);

INSERT INTO product (id, product_name, product_category_id, price)
VALUES (1, 'Toaster', 1, 2000);
INSERT INTO product (id, product_name, product_category_id, price)
VALUES (2, 'TV', 2, 9000);
INSERT INTO product (id, product_name, product_category_id, price)
VALUES (3, 'Couch', 2, 4500);
INSERT INTO product (id, product_name, product_category_id, price)
VALUES (4, 'Fridge', 1, 8000);
INSERT INTO product (id, product_name, product_category_id, price)
VALUES (5, 'Kettle', 1, 2500);
INSERT INTO product (id, product_name, product_category_id, price)
VALUES (6, 'Microwave', 1, 4000);
```

```
CREATE OR REPLACE FUNCTION t_cursor() RETURNS text
language plpgsql AS $$
DECLARE
 test_cursor CURSOR FOR
 SELECT id, product_name, price
 FROM product;
 currentID INT;
 currentProductName VARCHAR(100);
 currentPrice INT;
BEGIN
 OPEN test_cursor;
 LOOP
  FETCH test_cursor INTO currentID, currentProductName, currentPrice;
  EXIT WHEN NOT FOUND;
RAISE NOTICE '% % (ID: %)', currentProductName, currentPrice, currentID;
 END LOOP;
 CLOSE test_cursor;
 RETURN 'Done';
END $$;

SELECT * FROM t_cursor()
```

```
NOTICE: Toaster 2000 (ID: 1)
NOTICE: TV 9000 (ID: 2)
NOTICE: Couch 4500 (ID: 3)
NOTICE: Fridge 8000 (ID: 4)
NOTICE: Kettle 2500 (ID: 5)
NOTICE: Microwave 4000 (ID: 6)
```

The code for a cursor in PostgreSQL works a little differently to other databases. Here's what the code is doing:

Create a function. We start by creating a function to contain the code for our cursor. This is an easy way to get the cursor created and able to call it.

Declare the cursor. We then declare the cursor, calling it test_cursor, and defining the SELECT query that's used.

Declare the variables to store the row data. We declare three variables here (currentID, currentProductName, and currentPrice) to store the row data. This could be stored in a single variable with a type that matches the table row, as an alternative.

Begin the execution code. The BEGIN statement starts the code execution.

Open the cursor. The next step is to open the cursor, which runs the SELECT statement.

Begin the loop. This will start a loop and execute all code between here and the END LOOP statement, until it exits.

Fetch a row of data into the variables. The FETCH command will get the first row of data from test_cursor and place the values into the three variables mentioned. For each loop, it will get the next row.

Exit the loop when no row is found. If no row is found, then the loop exits. This is to ensure the loop does not run indefinitely.

Output the data using RAISE NOTICE. In PostgreSQL, you can use the RAISE NOTICE command to output a message to the output panel. The % symbols indicate placeholders for variables, and there are three which match the number of variables mentioned.

End the loop. The END LOOP command will end the loop code and rerun the loop.

Close the cursor. The cursor will be closed and memory released to the application.

Return a text value. Because a text value needs to be returned from the function we return one here.

End the function. This is done by using the END statement.

Call the function to run it. Finally, we call the function by running it inside a SELECT command to see the output.

Example 2:

```
CREATE OR REPLACE FUNCTION e_cursor() RETURNS text
language plpgsql AS $$
 DECLARE
   cur_emp CURSOR FOR
```

```
     SELECT empid,salary FROM employee
          WHERE salary > 20000;
        eid int;
        esal int;

  BEGIN
    OPEN cur_emp;
  LOOP
    FETCH cur_emp INTO EID,ESAL;
          EXIT WHEN NOT FOUND;
          RAISE NOTICE '% %',eid,esal;
  END LOOP;
  CLOSE cur_emp;
   RETURN 'OK';
  END $$;

  select * from e_cursor()
```

Output:

```
NOTICE: 1 34500
NOTICE: 2 54000
NOTICE: 3 46000
```

- **Error handling -Introduction Error Handling, RAISE Statement**

When an error occurs in a block, PostgreSQL will abort the execution of the block and also the surrounding transaction.

To recover from the error, you can use the exception clause in the begin...end block.

The following illustrates the syntax of the exception clause:

```
<<label>>
declare
begin
   statements;
exception
   when condition [or condition...] then
     handle_exception;
  [when condition [or condition...] then
     handle_exception;]
  [when others then
     handle_other_exceptions;
  ]
end;
```

How it works.

1. When an error occurs between the begin and exception, PL/pgSQL stops the execution and passes the control to the exception list.
2. PL/pgSQL searches for the first condition that matches the occurring error.
3. If there is a match, the corresponding handle_exception statements will execute. PL/pgSQL passes the control to the statement after the end keyword.
4. If no match found, the error propagates out and can be caught by the exception clause of the enclosing block. In case there is no enclosing block with the exception clause, PL/pgSQL will abort the processing.

The condition names can be **no_data_found** in case of a select statement return no rows or **too_many_rows** if the select statement returns more than one row. For a complete list of condition names on the PostgreSQL website.

1) Handling no_data_found exception example
The following example issues an error because the Item with id 2 does not exist.

Error occurs without Exception	**Error occurs with exception**

Do
$$
declare
        rec record;
        e_item_id int = 8;
begin
        -- select a item
        select item_id, item_name
        into strict rec
        from item
        where item_id = e_item_id;
end;
$$
language plpgsql;
	Do
	$$
	declare
	        rec record;
	        e_item_id int = 8;
	begin
	        -- select a item
	        select item_id, item_name
	        into strict rec
	        from item
	        where item_id = e_item_id;
	
	        exception
	          when no_data_found then
	             raise exception 'item % not found', e_item_id;
	end;
	$$
	language plpgsql;

`ERROR: query returned no rows CONTEXT: PL/pgSQL function inline_code_block line 7 at SQL statement SQL state: P0002`	`ERROR: item 8 not found CONTEXT: PL/pgSQL function inline_code_block line 14 at RAISE SQL state: P0001`

2) Handling too_many_rows exception example

The following example illustrates how to handle the too_many_rows exception:

```
do
$$
declare
        rec record;
begin
        -- select film
        select item_id, item_name
        into strict rec
        from item
        where item_name LIKE 'M%';

        exception
          when too_many_rows then
            raise exception 'Search query returns too many rows';
end;
$$
language plpgsql;
```

Output:

```
ERROR: Search query returns too many rows CONTEXT: PL/pgSQL
function inline_code_block line 13 at RAISE SQL state: P0001
```

In this example, the too_many_rows exception occurs because the select into statement returns more than one row while it is supposed to return one row.

3) SQLSTATE codes

As per above discussed in various example of Exceptions in PostgreSQL , we used No_data_found and too_many_rows named exceptions. We can use SQLSTATE codes fro named exceptions. The list is very long but for example some codes are as follows.

Class P0 — PL/pgSQL Error

P0000	PLPGSQL ERROR	plpgsql_error
P0001	RAISE EXCEPTION	raise_exception
P0002	NO DATA FOUND	no_data_found
P0003	TOO MANY ROWS	too_many_rows

https://docstore.mik.ua/manuals/sql/postgresql-8.2.6/errcodes-appendix.html