PostgreSQL

RDBMS (What is it?)

An `RDBMS` (Relational Database Management System) is a type of database management system that store data in a structured format using `table` (also known as `relation`).

These `Tables` consist of `rows` and `columns` .

Where,

- `row:` represent a `record`
- `column:` represent a `data` attribute.

RDBMSs are based on the principles of the `relational model` of data, introduced by `edger F. Codd` in `1970`.

`Key Characteristics of RDBMS:`

- `Data organization in tables:`

  Data is organized in `tables(relations)` with `rows(records)` and `columns(attributes)`. each table has `primary key` that uniquely identifies each `row`.

- `Relationships:`

  RDBMS allows defining relationships between different tables using `foreign keys`. A `foreign key` in one table points to the `primary key` of other table.

- `SQL (Structured Query Language):`

  `SQL` is the standard language used to `query, insert, update,` and `delete` data from the `database`. SQL also allows the creation and modification of `database schemas` and structures.

- `Data integirty and Constraints:`

  RDBMSs support various data integrity rules, such as `primary keys`, `foreign keys`, `unique constraints`, and `check constraints`, which ensure the accuracy and consistency of data.

- `ACID:`

  RDBMSs support the `ACID` properties `(Atomicity, Consistency, Isolation, Durability)`, which ensure that transactions are processed reliably.

- `Normalization:`

  To reduce `redundancy` and `dependency`, data in RDBMS can be normalized (organized into different tables) according to certain rules `(1NF, 2NF, 3NF, etc.)`.

- `Scalability and Security:`

RDBMSs provide mechanisms for security `(user roles, access control)` and scalability, allowing them to handle large amounts of data and users.

PostgreSQL

Year of First Release

Internal architecture of PostgreSQL

Indexes

An index is a data structure that increases the data retrieval speed by providing a rapid way to locate rows within a table.

```
-- syntax create index (B-Tree)
CREATE INDEX index_name
ON table_name(column_name);

-- display all index in the database
SELECT * FROM pg_indexes WHERE schemaname  !='pg_catalog' AND schemaname  !=
'information_schema';

-- display all index in a table
SELECT * FROM pg_indexes where tablename='users';
```

`Types of indexes:`

B-Tree index

It is a default index type in PostgreSQL, stand for `balanced tree`. B-tree indexes maintain the `sorted values` them efficient for `exact matches` and `range queries`.

```
-- syntax
CREATE INDEX index_name ON table_name(column_name);

-- example
CREATE INDEX idx_users_name ON users(name);
```

`Key Characteristics:`

- `Self-balancing:` Ensures that the tree remains balanced, which optimizes search times.
- `Sorted Data:` Stores keys in a sorted order, making it efficient for queries that require ordering.
- `Logarithmic Search Time:` B-trees guarantee O(log N) search time, which is much faster than linear search in unsorted data.

`How does it work:` B-Tree index works by maintaining a `sorted` structure.

Where:

- `leaf nodes:` contain the actual indexed values along with pointers to the corresponding table rows.
- `Non-leaf nodes:` contain pointers to lower-level nodes (either leaf or other non-leaf nodes)

```
-- how to query B-Tree index
SELECT * FROM employees WHERE salary BETWEEN 50000 AND 70000;
```

`Internals:`

- `Page Structure:` PostgreSQL's B-Tree index is based on a structure called a `page`, where each page is a `block` of data (usually `8KB` in size). Each page can store a certain number of keys and pointers to child page or data rows.

  Node Spilliting and Merging:

  - When an index page becomes full, it splits into `two` pages.
  - When the number of entries decreases too much, nodes can be `merged`.

  Efficient updates:

  When you insert, update and delete rows that affect indexed columns, the B-Tree index adjusts ifself to maintain its balance and sorted order.

`Multi-column B-Tree index:`

A B-Tree index can also be created on multiple columns. This is useful for the queries that filter by mutiple columns.

```
-- syntax
CREATE INDEX index_column_name ON users(multiple_column_name);

-- example
CREATE INDEX idx_name_salary ON users(name,salary);
```

`Partial B-Tree Index:`

You can create a partial index to index only a subset of rows that meet a specific condition. This is useful when you have many rows, but only a small fraction of them are frequently queried.

```
--example
CREATE INDEX idx_high_salary ON users (salary) WHERE salary > 100000;
-- This index will only index users with a salary greater than 100,000.
```

`B-Tree Index and Sorting:`

Since B-Tree indexes store data in sorted order, they can also improve the performance of queries that require sorting.

```sql
SELECT * FROM employees ORDER BY salary DESC;
```

`Real-world Scenarios:`

- `OLTP (Online Transaction Processing):` B-Tree indexes are commonly used in OLTP systems for efficient retrieval and updates. Since OLTP systems involve a high rate of inserts, updates, and selects, the B-Tree index strikes a good balance between read and write performance.
- `Analytics:` While B-Tree indexes can be useful in analytical queries, more complex queries often benefit from different index types (e.g., hash, GiST, or GIN indexes), depending on the query pattern.

Hash index

A Hash Index in `PostgreSQL` is a type of `index` that uses a `hash table` to store `pointers` to the rows in the `table` based on a `hash function` applied to the indexed column(s).

Hash indexes directly map the values to hash buckets.

Hash indexes maintain 32-bit hash code created from values of the indexed columns, handled only simple equality comparisions (=).

```sql
-- syntax
CREATE INDEX index_name ON table_name USING HASH(column_name);

-- example
CREATE INDEX idx_users_name ON users  USING HASH (name);

-- how this will optimize queries
SELECT * FROM users WHERE name='pradeep kumar';
```

`Key Characteristics:`

- `Equality Queries:` Hash indexes are particularly optimized for `equality queries` such as `=`. `WHERE column_name = value`
- `No Ordering:` Hash index do not maintain any order of the indexed values.
- `Faster Loopkup:` For large datasets with equality-based queries, hash indexes can provide faster lookups then B-Tree indexes.
- `Collision handling:` Hash indexes handle collisions (where multiple values hash to the same bucket) using techniques like `chaining` and `open addressing`.

`How hash indexes work:`

- `Hash Function:` When a value from the indexed column is `inserted` or `queried`, `PostgreSQL` applies a h`ash function` to the `value`, which produces a hash value (a fixed-size binary value).
- `Hash Buckets:` The hash value is used to determine the `"bucket"` where the data is stored. Each bucket stores a list of rows that have the same hash value.
- `Direct Lookup:` When a query uses the indexed column with an `equality` condition, PostgreSQL can directly look up the corresponding bucket and retrieve the rows matching that hash value.

GIN index

GIN indexes are inverted indexes taht are suitable for composite values such as `arrays, JSONB data`, and `full-text search`.

GiST index

SP-GiST index

BRIN (Block Range Index) Index

Database

Database is a collection of related data serves as a container for

- `tables`
- `indexes`
- `views`
- `etc.`

and other database `objects`.

Create new databse

`Syntax:`

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

- `OWNER:` Assign a role(user) that will be the owner of the database. the owner have the highest level of control over the database.

The owner has `full privileges` over the `database`, including the ability to `drop` it, `alter` it, and `grant/revoke` privileges to other `users`.

```
OWNER = myuser

-- DEFAULT USER IS postgres
```

- `TEMPLATE:` Specify the `template` database for the new database. postgres use the `[template1]` database as default template database.
  - PostgreSQL comes with two default templates:
    - `template0:` A minimal database with no user-created objects.
    - `template1:` A more commonly used template that may include objects and customizations.

```
TEMPLATE=template1
```

`NOTE:`

The database created from a template inherits all objects `(tables, functions, etc.)` in the template. However, certain templates, like `template0`, are clean and have no `user-defined` objects.

- `ENCODING:`
  - Defines the character encoding for the database.
  - This setting determines how `text` is stored in the `database`.
  - Common encodings include `UTF8` (recommended for most applications).
  - Common values: `UTF-8`,`LATIN1`,`SQL_ASCII`, etc.

    ```
    ENCODING='UTF-8'
    ```

- `LC_COLLATE:`
  - Specifies the `collation` order to use for string `sorting` and `comparison.`
  - it determines the rules for character ordering in the database.
  - `Collations` are based on `locale` settings and can differ for different `languages` and `regions`.
  - Common values: `en_US.UTF-8`, `fr_FR.UTF-8`, etc

```
    LC_COLLATE = 'en_US.UTF-8'
    ```

    `NOTE:`

    You cannot change these properties after the database is created, so they should be set carefully.

- `LC_TYPE:` Defines the `character` classification and `case conversion` behavior, such as upper and lower case conversions. like `LC_COLLATE`, this is local-based.

  Common values: `en_US.UTF-8`, `fr_FR.UTF-8`, etc

  ```
  LC_TYPE = 'en_US.UTF-8'
  ```

  `NOTE:`

  You cannot change these properties after the database is created, so they should be set carefully.

- `TABLESPACE:` Specifies the tablespace where the `database` should reside. A `tablespace` is a `storage` location on the `disk` where the database `files` are stored.

  - If not specified, `PostgreSQL` uses the default tablespace.
  - The default tablespace is `PostgreSQL` is `pg_default`, which maps to `/data` directory in PostgreSQL.
  - `PostgreSQL` also has a second default tablespace called `pg_global`, which stores global data.

  ```
  TABLESPACE=my_tablespace

  -- if you want to print available tablespace use below commands.
  SELECT spcname from pg_tablespace;

  -- show the physical location of tablespace
  show data_directory;

  -- To list all tables that are stored in the pg_default tablespace
  SELECT tabelname from pg_tables;
  ```

- `ALLOW_CONNECTIONS:` Determines whether the database should allow connections. Setting this to false will prevent anyone from connecting to the database, but the database will still exist for administrative purposes.

  Possible values: `TRUE|FASLE`

```
ALLOW_CONNECTIONS = true
```

- `CONNECTION LIMIT:` Defines the maximum number of concurrent connections allowed to the database. A `-1` value (or omitting this option) means unlimited connections.

  ```
  CONNECTION LIMIT = 100
  ```

- `IS_TEMPLATE:` Specifies whether the database should be treated as a template for creating new databases.

  Possible values: `TRUE|FALSE`

  ```
  IS_TEMPLATE=true
  ```

`Example:`

```
CREATE DATABASE blogs
  OWNER = pkuser
  TEMPLATE = template1
  ENCODING = 'UTF8'
  LC_COLLATE = 'en_US.UTF-8'
  LC_CTYPE = 'en_US.UTF-8'
  TABLESPACE = my_tablespace
  ALLOW_CONNECTIONS = true
  CONNECTION LIMIT = 100
  IS_TEMPLATE = false;


-- OR
-- rest using default values of each parameter
CREATE DATABASE blogs;

-- retrieve the database names from the `pg_database`
SELECT datname from pg_database;

-- list all database in [psql]
\l

-- connect to created or any database
\c db_name
```

Alter database

`ALTER DATABASE` statement allow you to carry the following action on the database.

- Change the attributes of the database.
- Rename the database.
- Change the owner of the database.
- Change the default tablespace of a database.
- Change the session default for a non-runtime configuration variable for a database.

`Changing attributes of a database`

```
-- syntax
ALTER DATABASE name WITH option;

-- option can be;
-- IS_TEMPLATE
-- CONNECTION LIMIT
-- ALLOW_CONNECTIONS

-- Only superusers or database owner can change these settings;
```

`Rename database:`

```
-- syntax
ALTER DATABASE db_name
RENAME TO new_db_name;

-- It is not possible to rename the current database.
-- Only superusers and database owners with [CREATEDB] privilege can rename the
database.
```

`Change the owner of the database:`

```
-- syntax
ALTER DATABASE db_name
OWNER TO new_owner | current_user | session_user;

-- to check current user or session user run below command
SELECT current_user; -- Returns the current role executing the query
SELECT session_user;  -- Returns the role that authenticated the session
SELECT user;  -- Equivalent to SELECT CURRENT_USER;

-- session information: pg_stat_activity
SELECT usename, application_name, client_addr, backend_start, state
FROM ps_stat_activity
WHERE pid=pg_backend_pid();

-- In PostgreSQL,[ pg_backend_pid()] is a function that returns the[ process ID
(PID)] of the current backend process.
```

```sql
-- how to terminate the current session
SELECT pg_terminate_backend(pg_backend_pid());
```

Change the default tablespace of a database:

```sql
-- syntax
ALTER DATABASE db_name
SET TABLESPACE new_tablespace;

-- To set the new tablespace, the tablespace needs to be empty and there is a connection to the database.
-- Superusers and database owners can change the default tablespace of the database
```

Change session defaults for run-time configuration variables:

Whenever you connect to a `database`, `PostgreSQL` loads the `configuration variables` from the `postgresql.conf` file and uses these variables by default

```sql
-- syntax
ALTER DATABASE database_name
SET configuration_parameter = value;

-- check the current setting from [pg_settings]
SELECT name, setting FROM pg_settings;

-- change configuration variables for current session
SET <configuration_parameter> TO <value>;


-- examples
SET work_mem TO '64MB'; -- (Memory used for sorting operations)
SET search_path TO my_schema, public; -- (Schema search order)
SET timezone TO 'UTC'; -- (Time zone setting)
SET statement_timeout TO '5min'; -- (Maximum execution time for a statement)
SET log_statement TO 'ddl'; -- (Logging level for SQL statements)
SET default_transaction_isolation TO 'READ COMMITTED'; -- (Transaction isolation level)

-- Reset configuration variable to default values
RESET <<variable_name>>
-- OR
RESET ALL; -- reset all

-- exmple reset work_mem only
RESET work_mem;
```

`Examples:`

```
-- create database
CREATE DATABASE testdb2;

-- rename to testhrdb
ALTER DATABASE testdb2
RENAME TO testhrdb;

-- change owner postgres to hr
ALTER DATABASE testhrdb
OWNER TO hr;

-- change the default tablespace of the testhrdbfrom pg_default to hr_default
ALTER DATABASE testhrdb
SET TABLESPACE hr_default;

-- set escape_string_warning configuration variable to off by using the following
statement:
ALTER DATABASE testhrdb
SET escape_string_warning = off;
```

Drop database

The `DROP DATABASE` statement deletes a database from a `PostgreSQL` server.

```
-- syntax
DROP DATABASE [IF EXISTS] database_name
[WITH (FORCE)]

-- The FORCE option will attempt to terminate all existing connections to the
target database.
```

`NOTE:`

- The `DROP DATABASE` statement deletes the database from both `catalog entry` and `data directory`.
- Since `PostgreSQL` does not allow you to `roll back` this operation, you should use it with caution.
- To execute the `DROP DATABASE` statement, you need to be the database `owner`.

`Examples:`

```
-- Create some database
CREATE DATABASE hr;
CREATE DATABASE test;

-- Drop the hr database
```

```
DROP DATABASE hr;

-- Removing a non-existing database example (IF EXISTS WILL CHECK THE DATABASE
THEN DELETE IF EXISTS OTHERWISE DO NOTHING)
DROP DATABASE IF EXISTS non_existing_database;

-- Drop a database that has active connections example
DROP DATABASE test WITH (FORCE)
```

Rename database

```
-- Create database bots
CREATE DATABASE bots;

-- RENAME TO robots
ALTER DATABASE bots
RENAME TO robots;
```

Copy database within the same server

You want to copy a PostgreSQL database wihin a database server for testing purpose.

```
-- syntax
CREATE DATABASE targetDb
WITH TEMPLATE sourceDb;

-- example
CREATE DATABASE dvdrental_test
WITH TEMPLATE dvdrental;
```

`Copy database from one server to another:`

```
-- Step-1: dump the source database into a file.
pg_dump -U postgres -d sourcedb -f sourcedb.sql

-- Step-2: create new database
CREATE DATABSE demo;

-- Step-3: restore the dump file
psql -U postgres -d demo -f sourcedb.sql
```

`How to Get Sizes of Database Objects in PostgreSQ`

- Use the `pg_size_pretty()` function to format the size.
- Use the `pg_relation_size()` function to get the size of a table.

- Use the `pg_total_relation_size()` function to get the total size of a table.
- Use the `pg_database_size()` function to get the size of a database.
- Use the `pg_indexes_size()` function to get the size of an index.
- Use the `pg_total_index_size()` function to get the size of all indexes on a table.
- Use the `pg_tablespace_size()` function to get the size of a tablespace.
- Use the `pg_column_size()` function to obtain the size of a column of a specific type.

`Examples:`

```sql
-- Getting table sizes:
select pg_relation_size('actor');

-- he pg_size_pretty() function formats a number using bytes, kB, MB, GB, or TB
appropriately. For example:
SELECT
    pg_size_pretty (pg_relation_size('actor')) size;

-- To get the total size of a table
SELECT
    pg_size_pretty (
        pg_total_relation_size ('actor')
    ) size;


-- the following query returns the top 5 biggest tables in the dvdrental database
SELECT
    relname AS "relation",
    pg_size_pretty (
        pg_total_relation_size (C .oid)
    ) AS "total_size"
FROM
    pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C .relnamespace)
WHERE
    nspname NOT IN (
        'pg_catalog',
        'information_schema'
    )
AND C .relkind <> 'i'
AND nspname !~ '^pg_toast'
ORDER BY
    pg_total_relation_size (C .oid) DESC
LIMIT 5;


-- Getting database size
SELECT
    pg_size_pretty (
        pg_database_size ('dvdrental')
    ) size;

-- Getting index sizes
```

```
SELECT
    pg_size_pretty (pg_indexes_size('actor')) size;

-- Getting tablespace sizes
SELECT
    pg_size_pretty (
        pg_tablespace_size ('pg_default')
    ) size;

-- Getting PostgreSQL value sizes
SELECT
  pg_column_size(5 :: smallint) smallint_size,
  pg_column_size(5 :: int) int_size,
  pg_column_size(5 :: bigint) bigint_size;
```

PostgreSQL Schema

In PostgreSQL, a schema is a named collection of database object, including

- Tables
- Indexes
- Views
- Data Types
- Functions
- Procedures
- Operators
- Sequences
- Triggers
- Materialized Views
- Domains
- Aggregates
- Collations
- Foreign Tables

and many more.

A schema allows you to organize and namespace objects within a database.

A schema can be thought of as a `container` for `database` objects.

- A database can contain one or more schemas.
- A schema belongs to only one database.
- Two schemas can have different objects that share the same name.

`Access an object in schema:`

```
-- syntax
schema_name.object_name
```

```
-- example
SELECT * FROM public.users; -- public is schema
```

`For example:`

You may have `auth` schema that has `user` table and `public` schema which also has the `user` table.

```
public.user
-- Or
auth.user
```

`Advantage of using schema:`

- Schemas allow you to organize database objects, tables into logical group to make them more managabale.
- Schema enable multiple users to use one database without interfering with each other.

`The public schema:`

PostgreSQL autometically creates a schema called `public` for every new database. Whatever object you create without specifying the schema name, PostgreSQl will place it into this `public` schema.

```
CREATE TABLE table_name(
  ...
)

-- and
CREATE TABLE public.table_name(
  ....
)
```

`Schema search path:`

When you refer to a table name without its schema name. `user` table instead of a fully qualified name such as `public.user` table.

PostgreSQL searches for the table by using the `schema search path`, which is a list of schemas to look in.

PostgreSQL will access the first matching table in the `schema` search path. If there is no match, it will return an error, even if the name exists in another schema in the database.

The first schema in the search path is called the current schema.

```
-- access current schema
SELECT current_schema(); -- it will return the current schema, default is public
```

```
-- To view the current search path
SHOW search_path; -- RESULT "$user", public
```

The `"$user"` specifies that the first schema that PostgreSQL will use to search for the object, which has the same name as the current user.

`For cxample:`

If you use the `postgres` user to log in and access `user` table. PostgreSQL will search for the `user` table in the `postgres` schema. if it cannot find any object like that, it continues to look for the object in the `public` schema.

The second element referes to the `public` schema as we have seen in the result.

`List schema of current database:`

```
-- run this command in psql
\dn
```

`Add new created schema to the search path:`

```
-- suppose you have created new [auth] schema
SET search_path TO auth, public;
```

Now, if you create new table named `user` without specifying the schema name, PostgreSQL will put this `user` table into the `auth` schema:

```
CREATE TABLE user(
  user_id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(255) NOT NULL,
  age int NOT NULL CHECK(age > 0 and age < 100),
  mobile VARCHAR(10) NOT NULL CHECK(LENGTH(TRIM(mobile))=10)
)
```

The user table belongs to the `auth` schema.

To access it

```
SELECT * FROM user;
-- OR
SELECT * FROM auth.user;
```

The `public` schema is the second element in the search path

```
-- you can set public
SET search_path TO public;
```

The public schema is not a special schema, you can `drop` it too.

`PostgreSQL schemas and previleges:`

User can only access objects in the schema that they own (`USAGE previlege`).

To allow users to access the objects in the schema that they do not own, you must grant the `USAGE` previlege of the schema to the users.

```
-- syntax
GRANT ON SCHEMA schema_name
TO role_name;

-- example
GRANT USAGE ON SCHEMA auth
TO pkumar;
```

To allow users to create objects in the schema that they do not own, you need to grant them the `CREATE` privilege of the schema to the users:

```
-- syntax
GRANT CREATE ON SCHEMA schema_name
TO user_name

-- example
GRANT CREATE ON SCHEMA auth
TO pkumar;
```

`Note:`

By default, every user has the `CREATE` and `USAGE` on the public schema.

Create schema:

The `CREATE SCHEMA` statement allows you to create a new `schema` in the `current` database.

```
-- syntax
CREATE SCHEMA [IF NOT EXISTS] schema_name;

-- You can also create schema for a user
CREATE SCHEMA [IF NOT EXISTS]
```

```
AUTHORIZATION username;
-- In this case, the schema will have the same name as the username.
```

`Note:` To execute the `CREATE SCHEMA` statement, you must have the `CREATE` privilege in the current database.

`To return all schemas from the current database:`

```
SELECT *
FROM pg_catalog.pg_namespace
ORDER BY nspname;
```

`Using CREATE SCHEMA statement to create a schema for a user example:`

First, create a new role with named `pkumar`

```
CREATE ROLE pkumar
LOGIN
PASSWORD '12345';
```

Second, create a schema for `pkumar`.

```
CREATE SCHEMA AUTHORIZATION pkumar;
-- OR
CREATE SCHEMA IF NOT EXISTS demo AUTHORIZATION pkumar;
```

Alter schema:

The `ALTER SCHEMA` statement allows you to change the definition of a `schema`.

`Rename schema:`

```
ALTER SCHEMA schema_name
RENAME TO new_name;
```

`Note`

That to execute this statement, you must be the owner of the `schema` and you must have the `CREATE` privilege for the database.

`Change the owner of schema:`

```
ALTER SCHEMA schema_name
OWNER TO { new_owner | CURRENT_USER | SESSION_USER};
```

`Examples:`

```sql
-- Rename schema [demo] to [demo1`]
ALTER SCHEMA demo
RENAME TO demo1;

-- Change the owner
ALTER SCHEMA demo
OWNER TO postgres;
```

Drop schema

The `DROP SCHEMA` removes a schema and all of its `objects` from a `database`.

```sql
-- syntax
DROP SCHEMA [IF EXISTS] schema_name
[ CASCADE | RESTRICT ];
```

- use `CASCADE` to delete schema and all of its objects, and in turn, all objects that depend on those objects.
- If you want to `delete schema only` when it is `empty`, you can use the `RESTRICT` option.
- By default, the `DROP SCHEMA` uses the `RESTRICT` option.

`Note:`

To execute the `DROP SCHEMA`statement, you must be the `owner` of the schema that you want to drop or a `superuser`.

`Example:`

```sql
-- drop schema demo
DROP SCHEMA IF EXISTS demo;

-- drop multiple schema
DROP SCHEMA IF EXISTS demo, auth, private;

-- drop schema if not empty
DROP SCHEMA blogs CASCADE;
```

Roles

`PostgreSQL` uses the concept of `roles` to represent `user` accounts. It doesn't use the concept of `users` like other database systems.

Typically, roles that can log in to the PostgreSQL server are called login `roles`. They are equivalent to `user` accounts in other database systems.

When `roles` contain other roles, they are referred to as `group roles`.

`Note:`

PostgreSQL combined the users and groups into roles since version 8.1

CREATE ROLE statement

Use `CREATE ROLE` statement.

```
-- syntax
CREATE ROLE role_name
```

When you create a `role`, it is valid for all databases within the database server (or client).

`Example:`

```
-- create role pkumar
CREATE ROLE pkumar;
```

`To retrieve all roles:`

```
-- display all rolename
SELECT rolname FROM pg_roles;

-- In psql,
-- you can use the \du command to show all roles that you create including the
postgres role in the current PostgreSQL server:
\du
```

`Note:`

- The roles whose names start with `pg_` are system roles.
- The `postgres` is a `superuser` role created by the `PostgreSQL` installer.

`Login access to a role`

To allow the login access to a role in the PostgreSQL server, you need to add the `LOGIN` attribute to it.

```
-- syntax
CREATE ROLE role_name
WITH -- optional
options;

-- options
CREATE ROLE role_name WITH
    LOGIN -- Allow the role to loing in to db,
    PASSWORD 'my_password' -- Set password for role,
    VALID UNTIL '2025-12-31' -- Set expiration date.,
    INHERIT -- Inherit privilege from other roles.,
    CREATEDB -- Allow the role to create new db. ,
    CREATEROLE -- Allow the role to create new role,
    SUPERUSER -- Grants the role all privilege.
    CONNECTION LIMIT connection_count;
```

`Create login role:`

Creating `pkumar` role that has the login privilege and initial password;

```
CREATE ROLE pkumar
LOGIN
PASSWORD '12345';

-- login in psql
psql -U pkumar -d blogs
```

`Create superuser role:`

The `superuser` role has all permissions within the `PostgreSQL` server.

Notice that only a `superuser` role can create another `superuser` role.

```
-- syntax
CREATE ROLE role_name
SUPERUSER
LOGIN
PASSWORD 'user_defined_passwor';
```

`Create roles with database creation permission:`

If you want to create roles that have the `database` creation `privilege`, you can use the `CREATEDB` attribute:

```
-- syntax
CREATE ROLE role_name
CREATEDB
```

```
LOGIN
PASSWORD 'user_defined_password';
```

`Create roles with a validity period:`

To set a date and time after which the role's password is no longer valid, you use the `VALID UNTIL` attribute:

```
-- syntax
VALID UNTIL 'timestamp';

-- example
CREATE ROLE pkumar
WITH
CREATEDB -- createdb privilege
LOGIN
PASSWORD '1234' -- user defined password
VALID UNTIL '2026-01-01'; -- valid only till '1 jan 26'
```

After one second tick in 2026, the password of pkumar is no longer valid.

`Create role with connection limit:`

To specify the number of concurrent connections a role can make, you use the `CONNECTION LIMIT` attribute:

```
-- syntax
CONNECTION LIMIT connection_count;

-- example
CREATE ROLE pkumar
WITH -- optional
CREATEDB -- create db privilege
LOGIN
PASSWORD '12345' -- user defined password
VALID UNTIL '2026-01-01' -- valid until 2026
CONNECTION LIMIT 100 -- 100 concurrent connections
;
```

GRANT

The `GRANT` statement to grant privileges on database object to a role.

After creating a `role` with the `LOGIN` attribute, the role can `log in` to the `PostgreSQL` database server.

However, it cannot do anything to the database objects like `tables`, `views`, `functions`, etc. For example, the `role` cannot select data from a `table` or `execute` a specific `function`.

To allow a `role` to interact with `database` objects, you need to `grant` privileges on the database objects to the role using the `GRANT` statement.

```
-- syntax
GRANT privilege_list | ALL
ON table_name
ON role_name;
```

- `Object Privileges:`
    - `SELECT:` Read data from a table, view, or foreign table.
    - `INSERT:` Insert new rows into a table.
    - `UPDATE:` Modify existing rows in a table.
    - `DELETE:` Delete rows from a table.
    - `TRUNCATE:` Remove all rows from a table.
    - `REFERENCES:` Create foreign key constraints referencing the table.
    - `TRIGGER:` Create triggers on the table.
    - `CREATE:` Create objects within the schema (e.g., tables, views, functions).
    - `CONNECT:` Connect to the database.
    - `TEMPORARY:` Create temporary tables.
    - `EXECUTE:` Execute functions.
    - `USAGE:` Grant the right to use a schema, sequence, language, or type.
    - `SET:` Set session parameters.
    - `ALTER:` SYSTEM: Alter system-wide parameters.
    - `MAINTAIN:` Perform maintenance operations on the object.
    - `USAGE:`(on a column): Use the column in expressions (e.g., in WHERE clauses).
- `Schema Privileges:`
    - `USAGE:` Use the schema and its objects.
    - `CREATE:` Create objects within the schema.
- `Database Privileges:`
    - `CONNECT:` Connect to the database.
    - `TEMPORARY:` Create temporary tables within the database.
    - `CREATE:` Create objects within the database.
- `Role Privileges:`

  The name of the `role` itself: `Grants` membership in the role, allowing the grantee to inherit privileges from the role.
- `Other Privileges:`
    - `ALL PRIVILEGES:` Grant all available privileges for the object type.
    - `WITH GRANT OPTION:` Allow the grantee to further grant the privilege to other roles.

`Grant SELECT, INSERT,UPDATE AND DELETE Privileges on the 'posts' table to the role 'pkumar'`

```
GRANT
SELECT, INSERT, UPDATE, DELETE -- privileges
ON posts -- table
TO pkumar;
```

`Grant all privileges on a table to a role:`

```
GRANT ALL
ON posts
TO pkumar;
```

`Grant all privileges on all tables in a schema to a role:`

```
GRANT ALL
ON ALL TABLES
IN SCHEMA "public"
TO pkumar;
```

`Grant SELECT privileges on all tables to a role:`

```
GRANT SELECT
ON ALL TABLES
IN SCHEMA "public"
TO pkumar;
```

REVOKE

The `REVOKE` statement revokes previously granted privileges on database objects from a `role`.

```
-- syntax
REVOKE privilege | ALL
ON TABLE table_name | ALL TABLES IN SCHEMA schema_name
FROM role_name;
```

`Check privileges:`

```
-- syntax
SELECT grantee, privilege_type
FROM information_schema.role_table_grants
WHERE table_name = 'table_name';
```

Grant ALL previlege to the role 'pkumar' on actor table:

```
GRANT ALL
ON actor
TO pkumar;
```

Revoke select previlege from a role:

```
-- revoking select permission from pkumar user
REVOKE SELECT
ON TABLE actor
FROM pkumar;
```

Revoke all on the actor:

```
REVOKE ALL
ON TABLE actor
FROM pkumar
```

Revoke Privileges on All Objects of a Schema:

```
-- syntax
REVOKE SELECT ON ALL TABLES IN SCHEMA schema_name FROM role_name;

-- example
REVOKE SELECT
ON ALL TABLES IN SCHEMA 'public'
FROM pkumar;
```

Revoke Privileges on Functions, Sequences:

```
-- syntax revoke on function
REVOKE EXECUTE ON FUNCTION function_name(arg1_type, arg2_type) FROM role_name;

-- syntax revoke on sequences
REVOKE USAGE, SELECT ON SEQUENCE sequence_name FROM role_name;
```

`Revoke Admin Privileges on a Database:`

```
REVOKE CONNECT ON DATABASE database_name FROM role_name;
```

Role membership

In PostgreSQL, a group role is a role that serves as a container for other individual roles.

`Create a group role:`

```
-- syntax
CREATE ROLE group_role_name;

-- create admin role
CREATE ROLE admin;
```

`Admin a role to a group role:`

```
-- syntax
GRANT group_role TO role;

-- example
GRANT
admin -- group role
TO
pkumar; -- role

-- If you don't want the role `pkumar` to inherit the privileges of its group
roles, you can use NOINHERIT attribute;
CREATE ROLE sales
WITH LOGIN NOINHERIT
PASSWORD '1234';
```

`Remove a role to a group role:`

```
-- syntax
REVOKE group_role FROM role;

-- example
REVOKE admin FROM pkumar;
```

PL/pgSQL

It a procedural language allow you to add many procedural elements

`such as:`

- Control structures
- Looping
- Conditions
- Complex computation

to extend standard SQL.

It allows you to develop complex function and stored procedures in PostgreSQL that may not be possible using pain SQL.

`Overview:`

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

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

PL/pgSQL is designed to:

- Create user-defined `functions, stored procedure`, and `triggers`.
- Extend statndard `SQL` by adding `control structures` such as `if-else, case` and `loop` statements.
- In herit all user-defined `functions, operators` and `types`.

`Advantage of using PL/pgSQL:`

`SQL` is a query language that allows you to effectively manage data in the database. However, PostgreSQL only can execute `SQL` statements individually.

It means that you have multiple statements, and you need to execute them one by one.

For exmple:

- First, send 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 computation and network overheads.

To resolve this issue, `PostgreSQL` uses `PL/pgSQL`.

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

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.

`Disadvandages:`

- Slower in software development because PL/pgSQL requires specialized skills that many developers do not possess.
- Difficult to manage versions and hard to debug.
- May not be portable to other database management systems.