MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

DAY – 3

TOPIC: POSTGRES CONFIGURATION

MIND MAP

PostgresSQL User and Role Management

- **User vs Role**
  - **Cluster**
    - Running instance of postgresSQL backend
    - Can have multiple on one computer (different ports)
    - Independent
    - Not multiple instance working together
  - **User**
    - Equivalent to create role + Login permission
    - Able to login
    - Ex: CREATE USER username WITH PASSWORD 'password'
  - **Role**
    - Not able to login (by default)
    - Can be granted login permission (WITH LOGIN)
    - Ex: Create Role role_name
    - Advantage: Grouping permissions
    - Essential for Authentication
    - Own database objects (table, functions , etc.)
    - Have cluster-level privileges (attributes)
    - Can be granted privileges on objects
    - Can grant privileges to others (inheritance)
    - Postgres sql terminology for users and groups
    - User roles: allowed to log in
    - Group roles: not allowed to log in
    - Create user/group are aliases for create role
  - **Role Attributes**
    - Predefined server-level setting
    - Boolean (**superuser, create database)**
    - Stored in pg_catalog.pg_roles table
    - Default for new role: inherit privileges, unlimited connections
    - Key attributes (postgesql 15+)
      - BYPASSRLS
      - CONNECTION LIMIT

- REPLICATION
- INHERIT
- CREATEROLE
- CREATEDB
- LOGIN
- General Management Role (non-superuser)
  - Recommended for most services
  - Can CREATE ROLE, CREATE DATABASE
  - May have ministrations (e.g some extensions)
- SUPERUSER
  - Required in every cluster
  - Name typically 'postgres' (process owner)
  - Bypass all privilege checks (except login)
  - Treat like root user (linux)
  - Cloud provides often don't provide direct access

- **Creating SQL**
  - Create User username with password 'password'
  - Create Role role_name with login / no login
- **Database**
  - Co-mingled with roles
  - Must be owned by role
- **Objects**
  - Contained within databases
  - Owned by role

- **Privileges (Access Rights)**
  - Access rights to database and objects
  - Ex: SELECT, UPDATE, USAGE, DELETE
  - Granted / Revoked by authority (e.g CREATE ROLE privilege)
  - Explicit GRANT / REVOKE apply only to existing objects
  - Applying Privileges
    - GRANT [privilege] ON [object type] TO [role]
    - Syntax: Grant / Revoke webpage
    - Privilege description: DDL priv page
- **Privilege Management Strategies**
  - Role Inheritance
    - Preferred way to manage roles in multi-tenant DB
    - Create group role (no login) with privileges
    - Grant membership to group roles for user role (with login)
    - User roles inherit all privileges of member roles

- Object Ownership (crucial)
  - Role that create an object becomes its owner
  - Owner has special, superuser-like privileges on that object
  - Principle of least privilege: Owner must grant access to others
  - Challenge
    - Cumbersome with many creators / object
  - Sol: Centralized Object Creation
    - Use a single group role (e.g app role) to create all objects
    - Manage privileges from one role (more manageable)
    - Run migration scripts as the app role
- Default Privileges
  - Tell postgresql what privileges to grant on *future* objects
  - Set by the role that will create and own the object
  - Ex:
    - GRANT SELECT ON TABLE TO PBLIC FOR ROLE [CREATOR_ROLE]
    - Automates access, avoid manual granting for each new object
- Four Options for Privilege Management
  - **Explicitly grant every time (cumbersome)**
  - **Individual developers set own default privileges**
  - **Application role sets default privileges (recommended)**
  - **Use predefined roles (e.g pg_read_all_data)**

- **Authentication & Connection**
  - Pg_hba.conf (Host based Authentication)
    - Firewall rile set for Postgresql
    - Specifies host, roles, database, authentication method
    - Crucial for connecting to any database
    - Managed by hosting provider/consoles typically
  - Authentication Methods
    - Avoid 'trust' (dangerous, rarely viable)
    - Move to SCRAM-SHA-256 for password
- **Public Schema & Public Role**
  - Default for new databases
  - New users/roles inherit permission from public role
  - Users can select/create objects in public schema by default
  - Not good practice (can be messy)
    - Public Role

            - All role are granted memberships (cannot delete/change)
            - Privileges can be managed on database/schemas
            - Default privileges (postges 14 and below)
                - CONNECT
                - USAGE
                - TEMPORARY
                - EXECUTE
                - CREATE (on public schema) – Security problem
            - Redefined Roles (postgres 14+)
                - E.g pg_read_all_data (cluster level setting)
                - Applied across the cluster
                - Growing in number to reduce superuser reliance
            - Default privileges (postges 15+)
                - CONNECT
                - USAGE
                - TEMPORARY
                - EXECUTE
                - NO CREATE on public schema
            - Security best practice
                - Remove all privileges on public schema from public role
                - Explicitly grant connect / usage to desired roles
    - Best pra: Revoke Privileges
        - Revoke create on schema pubic from public
        - Revoke all on database db_name form public
- **Search path**
    - Parameter defining schema search order
    - SHOW search_path displays: $uesr.public
    - Search order
        - 1. Schema with same name as logged in user
        - 2. Public schema (if user schema not present)
    - Allows omitting fully qualified table name (select * from actor instead of public actor)
    - Value can be changed
- **Creating Roles**
    - Create Role app_read_only

        - Grant connect on database db_name to app_read_only
        - Create schema app_schema
        - Grant usage on schema app_schema to app_read_only
        - Grant usage on all tables in schema app_schema to app_read_only
    - Create Role app_read_only
        - Create Role app read_write
        - Grant connect on database db_name to app_readwrite
        - Grant insert, update, delete on all tables in schema app_schema to app_readwrite
    - Benefits: Centralized permission management for application / users
- **Granting Roles to Users**
    - Grant role_name To user_name
    - Convenient for managing application groups
    - Changes to role apply to all assigned users

PostgreSQL Roles and Privileges: A Comprehensive Guide to Security Management

1. Roles in Detail

In PostgreSQL, the concept of a "role" is an elegant unification of what many other database systems distinguish as "users" and "groups." A single role entity can act as:

- **An Individual Database User:** A role can be an entity that can connect to the database, possessing a username and often a password. This is the most common way individual human users or application users are represented.
  - **Script Example:**

```
-- Create a login role for an individual user
CREATE ROLE alice WITH LOGIN PASSWORD 'strong_password_alice';
```

- **A Group of Users:** A role can serve as a logical collection of other roles. This is immensely powerful for simplifying privilege management. Instead of granting the same permissions to multiple individual users, you grant permissions to a group role, and then simply make the individual users members of that group role.
  - **Script Example:**

```
-- Create a group role for read-only access
CREATE ROLE analytics_team;


-- Grant the analytics_team group role to individual user roles
GRANT analytics_team TO alice;
GRANT analytics_team TO bob;
```

- **A Specific Application or System:** Roles can represent background processes, applications, or external systems that interact with the database, allowing their access rights to be managed distinctly.

Role Attributes (Cluster-Level Privileges):

Roles can possess specific attributes that define their global capabilities within the entire PostgreSQL cluster. These are not tied to specific database objects but to the role itself.

- **SUPERUSER**: Grants all possible privileges within the database cluster, bypassing all permission checks. This attribute should be used with extreme caution and only granted when absolutely necessary.
  - **Script Example:** ALTER ROLE charlie SUPERUSER;
- **CREATEDB**: Allows the role to create new databases.
  - **Script Example:** ALTER ROLE charlie CREATEDB;
- **CREATEROLE**: Allows the role to create, modify, and drop other roles. This implicitly grants the ability to add or remove members from roles.
  - **Script Example:** ALTER ROLE charlie CREATEROLE;
- **LOGIN**: Allows the role to connect to the database server. Roles without this attribute cannot initiate a session, making them ideal for group roles.
  - **Script Example:** CREATE ROLE app_user WITH LOGIN PASSWORD 'app_pass';
- **REPLICATION**: Allows the role to connect to the server in streaming replication mode or to create and manage replication slots. Essential for setting up standby servers.
  - **Script Example:** CREATE ROLE repl_user WITH LOGIN REPLICATION PASSWORD 'repl_pass';
- **BYPASSRLS**: Bypasses all Row-Level Security (RLS) policies. Like SUPERUSER, use with extreme care.
  - **Script Example:** ALTER ROLE auditor BYPASSRLS;
- **INHERIT / NOINHERIT (Default is INHERIT):** Controls whether a role automatically inherits the privileges of roles it is a member of. If NOINHERIT, a user must explicitly use SET ROLE or SET SESSION AUTHORIZATION to assume the privileges of a group role they are a member of.
  - **Script Example:** ALTER ROLE alice NOINHERIT; (Alice would then need to SET ROLE analytics_team; to use its privileges).
- **CONNECTION LIMIT n**: Specifies the maximum number of concurrent connections for a role.
  - **Script Example:** ALTER ROLE guest_user CONNECTION LIMIT 5;
- **PASSWORD '...'**: Sets the password for login roles.
  - **Script Example:** ALTER ROLE alice PASSWORD 'new_strong_password';

- **VALID UNTIL 'timestamp'**: Sets an expiration date for the role's password or the role itself.
  - **Script Example:** CREATE ROLE temp_user WITH LOGIN PASSWORD 'temp_pass' VALID UNTIL '2025-12-31 23:59:59+10'; (Sydney timezone)

Role Membership and Inheritance:

Roles can be granted membership in other roles using the GRANT command. This establishes a hierarchy where a member role inherits the privileges of its granted role(s).

- **Script Example:**

-- Grant the 'reporting_team' group role to 'bob'

GRANT reporting_team TO bob;

-- Grant 'data_entry_team' to 'alice' with the ability for Alice to further grant it

GRANT data_entry_team TO alice WITH ADMIN OPTION;

The WITH ADMIN OPTION allows the grantee (alice) to grant or revoke membership in data_entry_team to other roles, and to drop data_entry_team itself.

2. Special System Roles and Default Users

PostgreSQL comes with a few predefined roles and a default user:

- **Default User/Role:**
  - When you initialize a PostgreSQL cluster using initdb, a superuser role is created with the same name as the operating system user that executed initdb. This is typically postgres if you ran it as the postgres OS user. This role is a superuser and has full control over the database cluster.
- **Pseudo-Roles (Not true roles, but concepts):**
  - **PUBLIC**: This is a special implicit "role" that automatically includes every other role. By default, some privileges (e.g., CONNECT on new databases, CREATE on the public schema) are granted to PUBLIC, meaning they are available to everyone. This is often the first place to review and restrict for security.
    - **Script Example:**

-- Revoke CREATE privilege on the 'public' schema from PUBLIC

REVOKE CREATE ON SCHEMA public FROM PUBLIC;

-- Now, only schema owners or explicitly granted roles can create objects in public

- **CURRENT_USER**: The name of the role currently executing the query.
- **SESSION_USER**: The name of the user that initiated the current database session.

3. Privileges: Defining Actions on Objects

Privileges define the specific actions a role can perform on a database object. They are the granular control mechanisms that govern access to tables, views, functions, etc.

- **Ownership Privileges:**
  - The role that creates a database object automatically becomes its **owner**.
  - The owner implicitly has all possible privileges on that object, including the right to GRANT those privileges to others and REVOKE them.
  - Ownership can be transferred using ALTER object_type object_name OWNER TO new_owner;. This is crucial for managing permissions if the original owner leaves or changes roles.
- **GRANT and REVOKE Commands:**
  - **GRANT privilege_type ON object_type object_name TO role_name [WITH GRANT OPTION];**: Assigns privileges. WITH GRANT OPTION allows the recipient to further grant the privilege.
  - **REVOKE privilege_type ON object_type object_name FROM role_name [CASCADE | RESTRICT];**: Removes privileges. CASCADE revokes the privilege from anyone who received it via the revokee (dangerous if not understood); RESTRICT (default) prevents revocation if it would break a dependency.
- **Levels of Granularity:** Privileges can be granted at various levels in the object hierarchy, providing very fine-grained control:
  - **Database Level:** CONNECT, CREATE (for schemas), TEMPORARY/TEMP.
    - **Script Example:** GRANT CONNECT ON DATABASE mydatabase TO app_user;
  - **Schema Level:** CREATE (for objects within the schema), USAGE (to access objects within the schema).
    - **Script Example:** GRANT USAGE ON SCHEMA reports_schema TO reporting_team;

        - GRANT CREATE ON SCHEMA staging_schema TO data_loader_role;
    - **Table Level:** SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER.
        - **Script Example:** GRANT SELECT, INSERT ON sales.orders TO order_entry_app;
    - **Column Level (for SELECT, INSERT, UPDATE, REFERENCES):** Allows granting permissions on individual columns.
        - **Script Example:** GRANT SELECT (customer_id, customer_name) ON customers TO analytics_team;
    - **View Level:** SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER (if the view is updatable).
        - **Script Example:** GRANT SELECT ON customer_view TO external_api_user;
    - **Function/Procedure Level:** EXECUTE.
        - **Script Example:** GRANT EXECUTE ON FUNCTION calculate_total(int) TO accounting_role;
    - **Sequence Level:** USAGE, SELECT, UPDATE.
        - **Script Example:** GRANT USAGE, SELECT ON SEQUENCE order_id_seq TO order_entry_app;
    - **Tablespace Level:** CREATE.
        - **Script Example:** GRANT CREATE ON TABLESPACE fast_storage TO dev_team;
    - **Foreign Data Wrapper/Server/User Mapping Level:** USAGE.
- **Default Privileges (ALTER DEFAULT PRIVILEGES):** This powerful command allows you to define the privileges that *newly created objects* will automatically have. This simplifies ongoing security management, especially in applications where new tables are frequently created.
    - **Syntax:**

```
ALTER DEFAULT PRIVILEGES [ FOR ROLE target_role ] [ IN SCHEMA target_schema ]

GRANT privilege_type ON object_type TO role_name;
```

- **Script Example:**

```
-- Any tables created by 'app_owner' in the 'public' schema will automatically
-- grant SELECT, INSERT, UPDATE, DELETE to 'app_user_role'.
ALTER DEFAULT PRIVILEGES FOR ROLE app_owner IN SCHEMA public
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO app_user_role;

-- Any functions created by 'admin_role' in any schema will automatically
-- grant EXECUTE to 'public'.
ALTER DEFAULT PRIVILEGES FOR ROLE admin_role
GRANT EXECUTE ON FUNCTIONS TO PUBLIC;
```

4. Security Management and Essentials

Effective database security goes beyond just roles and privileges; it encompasses a holistic approach:

- **Principle of Least Privilege (PoLP):**
  - **Concept:** This is the cornerstone of robust security. Users and applications should only be granted the absolute minimum privileges required to perform their specific tasks. Avoid granting SUPERUSER or ALL PRIVILEGES unnecessarily.
  - **Benefit:** Minimizes the attack surface. If a compromised account only has limited privileges, the potential damage is contained.
- **Role-Based Access Control (RBAC):**
  - **Concept:** Implement RBAC by designing your roles around job functions or application components, rather than individual users. Assign appropriate privileges to these function-based roles. Then, grant these functional roles to individual user roles.
  - **Benefit:** Simplifies access management, ensures consistent security policies across groups of users, and makes it easier to onboard/offboard users.
- **Strong Authentication:**
  - **Concept:** Verify the identity of users and applications connecting to the database.

- **Mechanisms:**
  - **Strong Passwords:** Enforce complex, unique passwords for all login-enabled roles. Consider password expiration policies.
  - **pg_hba.conf Configuration:** This file controls client authentication. It defines which hosts can connect, which database they can connect to, which user they can connect as, and what authentication method is required.
    - **Common Authentication Methods:**
      - md5: Password authentication using MD5 hashing.
      - scram-sha-256: Stronger password authentication (recommended over MD5).
      - trust: Allows anyone to connect without a password (**highly insecure for production**).
      - ident: Uses the client's operating system user name.
      - peer: Uses the client's operating system user name for local connections.
      - cert: Client certificate authentication (very strong, requires SSL).
      - gssapi, ssi, ldap, radius, pam: Integration with external authentication systems.
  - **Client Certificate Authentication (SSL/TLS):** Provides strong mutual authentication where both the server and client verify each other's identities using digital certificates. This also encrypts data in transit.
- **Script Example (Conceptual pg_hba.conf entries):**

```
○ # TYPE  DATABASE      USER        ADDRESS            METHOD
○ host   all          all         127.0.0.1/32        scram-sha-256
○ hostssl all          app_user      192.168.1.0/24       cert clientcert=verify-
  full
○ host   mydb         admin_role     0.0.0.0/0           reject # Block external
  access to admin
```

- **Regular Auditing:**
  - **Concept:** Monitor and log database activity, including connection attempts, query executions, privilege changes, and potentially data modifications.
  - **Mechanisms:**
    - **PostgreSQL Logging:** Configure postgresql.conf parameters like log_connections, log_disconnections, log_statement = 'all', log_duration, log_min_duration_statement to capture detailed activity.
    - **pg_stat_activity:** View current active sessions.
    - **pg_auditor / Third-Party Tools:** For more comprehensive auditing, consider extensions like pg_auditor or external logging and security information and event management (SIEM) systems.
  - **Benefit:** Detects suspicious behavior, identifies security breaches, and provides evidence for forensics.
- **Network Security:**
  - **Concept:** Secure the network infrastructure around the PostgreSQL server to restrict access to authorized clients and protect data in transit.
  - **Firewall Details:**
    - **Purpose:** A firewall acts as a barrier, controlling incoming and outgoing network traffic based on predefined rules. For a database server, the primary goal is to **restrict access to the PostgreSQL port (default 5432)** to only trusted IP addresses or networks.
    - **Implementation:**
      - **Linux (e.g., ufw or firewalld):**
        - **Enable Firewall:**

```
sudo ufw enable
```

        - **Allow PostgreSQL Default Port (5432) from Specific IP:**

```
sudo ufw allow from 192.168.1.100 to any port 5432
```

        - **Allow PostgreSQL Default Port (5432) from Specific Network:**

```
sudo ufw allow from 192.168.1.0/24 to any port 5432
```

- **Allow SSH (Port 22):**

```
sudo ufw allow ssh
```

- **Deny All Other Incoming:** ufw's default policy is usually deny incoming, allow outgoing.
- **Disable Firewall (Temporarily, for troubleshooting):**

```
sudo ufw disable
```

- **Windows (Windows Defender Firewall):**
  - Through the GUI: "Windows Defender Firewall with Advanced Security" -> "Inbound Rules" -> "New Rule..."
  - Via PowerShell:

```
# Allow inbound on port 5432 for PostgreSQL
New-NetFirewallRule -DisplayName "PostgreSQL (TCP 5432 In)" -Direction Inbound -Protocol TCP -LocalPort 5432 -Action Allow
# Restrict source IP if needed (e.g., from 192.168.1.100)
# New-NetFirewallRule -DisplayName "PostgreSQL (TCP 5432 In - Specific IP)" -Direction Inbound -Protocol TCP -LocalPort 5432 -RemoteAddress 192.168.1.100 -Action Allow
```

- **Disable Firewall (Temporarily):**

```
Set-NetFirewallProfile -Profile Domain,Public,Private -Enabled False
```

(Re-enable with -Enabled True)

- **Cloud Provider Security Groups/Network ACLs:** In cloud environments (AWS Security Groups, Azure Network Security Groups, Google Cloud Firewall Rules), these are the primary methods for network access control.

- **SSL/TLS Encryption:** Always encrypt data in transit between clients and the PostgreSQL server using SSL/TLS. Configure ssl = on in postgresql.conf and set up server certificates and keys. This prevents eavesdropping and man-in-the-middle attacks.

5. Authentication Process and Levels

The authentication process in PostgreSQL is managed by the pg_hba.conf file, which specifies how clients are allowed to connect based on their connection type, database, user, and source IP address.

Authentication Process Flow:

1. A client attempts to connect to the PostgreSQL server.
2. The server receives the connection request and consults pg_hba.conf.
3. It scans pg_hba.conf from top to bottom, looking for the first record that matches the connection's type (local, host, hostssl, hostnossl), requested database, client user, and client IP address.
4. Once a matching record is found, the specified authentication method is applied.
5. If authentication succeeds, the connection is established. If it fails, the connection is rejected.

Levels of Authentication:

PostgreSQL provides various authentication methods, offering different levels of security:

- **Password-Based Authentication:**
    - **md5**: Uses MD5 hashing for passwords. Historically common, but **less secure than SCRAM due to MD5's vulnerabilities**.
    - **scram-sha-256**: **Highly recommended for password authentication.** Uses Salted Challenge Response Authentication Mechanism (SCRAM) with SHA-256 for stronger, more secure password hashing and authentication. It protects against brute-force attacks and eavesdropping.
- **Operating System User-Based Authentication:**
    - **ident**: On Unix-like systems, connects to an ident server on the client machine to get the client's OS username. If it matches the PostgreSQL user, authentication succeeds. Less common now.
    - **peer**: Similar to ident, but for local connections only. It checks if the OS user running the client process matches the PostgreSQL role name. Secure for local connections.

- **Certificate-Based Authentication:**
    - **cert**: Requires the client to present an SSL certificate. This provides strong client identity verification and is highly secure, especially when combined with SSL encryption. It can be configured for verify-ca (check CA chain) or verify-full (check hostname in certificate against connection).
- **External Service Integration:**
    - **gssapi**: Generic Security Services API, often used with Kerberos for single sign-on.
    - **ssi**: Simple Security Interface, primarily for internal Postgres-to-Postgres communication.
    - **ldap**: Lightweight Directory Access Protocol. Authenticates against an LDAP directory (e.g., Active Directory). Ideal for integrating with existing enterprise identity management systems.
    - **radius**: Remote Authentication Dial-In User Service. Authenticates against a RADIUS server.
    - **pam**: Pluggable Authentication Modules. Allows integration with various authentication schemes configured via PAM (e.g., system passwords, smart cards).
- **No Authentication (for Specific Use Cases, generally insecure):**
    - **trust**: Assumes anyone who can connect is authorized. **Extremely dangerous and should only be used in highly controlled, isolated development environments.**
    - **reject**: Explicitly rejects any connection attempts matching the record. Useful for blocking specific users or IP ranges.

After Post Installation Configuration Step By Step

The provided text details crucial post-installation setup steps for a software, likely PostgreSQL, focusing on ensuring the system can find its newly installed components. This involves configuring shared libraries and environment variables.

Here's a detailed breakdown with example scripts:

1. Shared Libraries

Shared libraries are essential dynamic-link libraries that programs need to run. If your operating system isn't on the list that automatically finds new shared libraries (FreeBSD, Linux, NetBSD, OpenBSD, Solaris), you need to tell it where to look.

When is this necessary? You'll know if you encounter an error like:

psql: error in loading shared libraries

libpq.so.2.1: cannot open shared object file: No such file or directory

Methods to set the shared library search path:

a. Using LD_LIBRARY_PATH (Most Widely Used)

This method involves setting the LD_LIBRARY_PATH environment variable to the directory where your new shared libraries are located (e.g., /usr/local/pgsql/lib).

- **For Bourne-compatible shells (sh, ksh, bash, zsh):**

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib

export LD_LIBRARY_PATH
```

- **For C shells (csh, tcsh):**

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Important: Replace /usr/local/pgsql/lib with the actual directory you specified with --libdir during installation.

Where to put these commands: To make these settings permanent for your user, add them to your shell's startup file. Common files include:

- ~/.bash_profile (for Bash users, executed when you log in)
- ~/.bashrc (for Bash users, executed for every new shell)
- /etc/profile (for all users on the system, requires root privileges)

Example for ~/.bash_profile (or ~/.bashrc):

```
# Add PostgreSQL shared library path
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

b. Using LD_RUN_PATH (Before Building)

On some systems, it might be preferable to set the LD_RUN_PATH environment variable *before* building the software. This embeds the library search path directly into the compiled binaries, so you don't need to set LD_LIBRARY_PATH at runtime. This method is generally more robust but requires foresight during the build process. The exact usage depends on your build system.

c. Cygwin Specifics

If you are on Cygwin, you have two options:

- **Add the library directory to your PATH:** This makes the system search for .dll files in that directory.
- **Move the .dll files into the bin directory:** This simplifies finding the libraries as the bin directory is usually already in the PATH.

d. System-Specific ldconfig (Linux, FreeBSD, NetBSD, OpenBSD)

For faster runtime linker performance, especially on Linux, FreeBSD, NetBSD, and OpenBSD, you can use the ldconfig command as a root user. This command creates the necessary links and cache to the most recent shared libraries found in the specified directories.

- **On Linux (as root):**

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(Replace /usr/local/pgsql/lib with your actual library directory.)

- **On FreeBSD, NetBSD, and OpenBSD (as root):**

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

Consult the manual page for ldconfig (man ldconfig) for more details on its options and behavior.

2. Environment Variables

Setting environment variables makes using the installed software more convenient.

a. Adding the bin Directory to PATH

If you installed the software into a non-standard location like /usr/local/pgsql (which is not typically searched for programs by default), you should add its bin directory to your system's PATH environment variable. This allows you to run commands like psql directly without specifying their full path.

- **For Bourne-compatible shells (sh, ksh, bash, zsh):**

```
PATH=/usr/local/pgsql/bin:$PATH

export PATH
```

- **For C shells (csh, tcsh):**

```
set path = ( /usr/local/pgsql/bin $path )
```

Where to put these commands: Similar to LD_LIBRARY_PATH, add these to your shell's startup file (~/.bash_profile, ~/.bashrc, or /etc/profile).

Example for ~/.bash_profile (or ~/.bashrc):

```
# Add PostgreSQL bin directory to PATH

PATH=/usr/local/pgsql/bin:$PATH

export PATH
```

b. Setting MANPATH for Documentation

If you installed the man documentation into a location that isn't searched by default, you'll need to update your MANPATH environment variable so your system can find the documentation pages.

- **For Bourne-compatible shells (sh, ksh, bash, zsh):**

```
MANPATH=/usr/local/pgsql/share/man:$MANPATH

export MANPATH
```

Where to put these commands: Add these to your shell's startup file (~/.bash_profile, ~/.bashrc, or /etc/profile).

Example for ~/.bash_profile (or ~/.bashrc):

```
# Add PostgreSQL man page path

MANPATH=/usr/local/pgsql/share/man:$MANPATH

export MANPATH
```

c. PGHOST and PGPORT (PostgreSQL Specific)

These environment variables are specific to PostgreSQL and are used by client applications to connect to the database server. They override the compiled-in default host and port.

- **PGHOST:** Specifies the hostname or IP address of the database server. This is particularly convenient if you're running client applications remotely.
- **PGPORT:** Specifies the port number on which the database server is listening (default is 5432).

While not strictly required (you can often specify these via command-line options), setting PGHOST is convenient for users who frequently interact with a remote database.

- **For Bourne-compatible shells (sh, ksh, bash, zsh):**

```
PGHOST=your_database_server_hostname_or_ip

export PGHOST

PGPORT=5432 # or your custom port

export PGPORT
```

- **For C shells (csh, tcsh):**

Code snippet

```
setenv PGHOST your_database_server_hostname_or_ip

setenv PGPORT 5432 # or your custom port
```

Example for ~/.bash_profile (or ~/.bashrc):

```
# PostgreSQL connection settings

PGHOST=localhost # Or your database server's hostname/IP

PGPORT=5432     # Or the port your PostgreSQL server is listening on

export PGHOST

export PGPORT
```

Summary of common shell startup file modifications (e.g., ~/.bash_profile):

```
# --- PostgreSQL Post-Installation Setup ---
```

# 1. Shared Libraries (if not on FreeBSD, Linux, NetBSD, OpenBSD, Solaris)

```
# Replace /usr/local/pgsql/lib with your --libdir path

LD_LIBRARY_PATH=/usr/local/pgsql/lib
```

export LD_LIBRARY_PATH

# 2. Environment Variables

# Add PostgreSQL bin directory to PATH for easy command execution

# Replace /usr/local/pgsql/bin with your --bindir path

PATH=/usr/local/pgsql/bin:$PATH

export PATH

# Add PostgreSQL man page path for documentation

# Replace /usr/local/pgsql/share/man with your man directory path

MANPATH=/usr/local/pgsql/share/man:$MANPATH

export MANPATH

# Optional: PostgreSQL connection settings for client applications

# Set PGHOST to your database server's hostname or IP

# Set PGPORT to the port your PostgreSQL server is listening on

# PGHOST=your_database_server_hostname_or_ip

# PGPORT=5432

# export PGHOST

# export PGPORT

# --- End PostgreSQL Setup ---

After modifying your shell startup file, you'll need to either log out and log back in, or source the file to apply the changes immediately:

source ~/.bash_profile # Or ~/.bashrc, or /etc/profile if you edited that

By following these steps and adjusting the paths to match your installation, you'll ensure that your system can properly locate and utilize the newly installed software.

Database, Table, and Column Privileges and Restrictions – Tailor fit Roles for Tasks and Responsibilities

In order for roles to use database objects (tables, views, columns, functions, etc...), they must be granted access privileges to them.

The GRANT command defines these essential privileges.

We'll go over a few examples to get the essence of its use.

Creating databases

Since log_user was granted the CREATEDB and CREATEROLE attributes, we can use this role to create a test database named trial.

```
postgres=> CREATE DATABASE trial:

CREATE DATABASE
```

In addition to creating a new ROLE:

```
postgres=> CREATE ROLE db_user WITH LOGIN PASSWORD 'scooby';

CREATE ROLE
```

Finally, log_user will connect to the new trial database:

```
postgres=> c trial;

Password for user log_user:

You are now connected to database "trial" as user "log_user".

trial=>
```

Notice the prompt changed to the name 'trial' indicating that we are connected to that database.

Let's utilize log_user to CREATE a mock table.

```
trial=> CREATE TABLE another_workload(

trial(> id INTEGER,

trial(> first_name VARCHAR(20),

trial(> last_name VARCHAR(20),

trial(> sensitive_info TEXT);

CREATE TABLE
```

Role log_user recently created a helper role, db_user. We require db_user to have limited privileges for table another_workload.

Undoubtedly, the sensitive_info column should not be accessed by this role. INSERT, UPDATE, and DELETE commands should not be granted at this time either, until db_user meets certain expectations.

However, db_user is required to issue SELECT queries. How can we limit this roles abilities within the another_workload table?

First let's examine the exact syntax found in the PostgreSQL GRANT command docs, at the table level.

```
GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )

[, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }

ON [ TABLE ] table_name [, ...]

TO role_specification [, ...] [ WITH GRANT OPTION ]
```

Next, we implement the requirements set forth for role db_user, applying specific syntax.

```
trial=> GRANT SELECT (id, first_name, last_name) ON TABLE another_workload TO db_user;

GRANT
```

Notice just after the SELECT keyword, we listed the columns that db_user can access. Until changed, should db_user attempt SELECT queries on the sensitive_info column, or any other command for that matter, those queries will not be executed.

With db_user logged in, we'll put this into practice, attempting a SELECT query to return all columns and records from the table.

```
trial=> SELECT * FROM another_workload;

ERROR: permission denied for relation another_workload
```

Column sensitive_info is included in this query. Therefore, no records are returned to db_user.

But db_user can SELECT the allowable columns

```
trial=> SELECT id, first_name, last_name

trial-> FROM another_workload;

id | first_name | last_name
```

```
-----+------------+-----------

10 | John | Morris

191 | Jannis | Harper

2 | Remmy | Rosebuilt

(3 rows)
```

That works just fine.

We will test INSERT, UPDATE, and DELETE commands as well.

```
trial=> INSERT INTO another_workload(id,first_name,last_name,sensitive_info)

VALUES(17,'Jeremy','Stillman','key code:400Z');

ERROR: permission denied for relation another_workload

trial=> UPDATE another_workload

trial-> SET id = 101

trial-> WHERE id = 10;

ERROR: permission denied for relation another_workload

trial=> DELETE FROM another_workload

trial-> WHERE id = 2;;

ERROR: permission denied for relation another_workload
```

By not assigning INSERT, UPDATE, or DELETE commands to db_user, the role is denied access to using them.

With the plethora of available options, configuring your role is virtually limitless. You can make them fully functional, able to execute any command, or as constrained as your requirements dictate.

Actionable Takeaways

- Roles are provided access privileges to database objects via the GRANT command.
- Database objects and commands against those objects, is highly configurable within the PostgreSQL environment.