**PostgreSQL Essentials: Must-Know Features & Tools to Maximize Developer and DBA Productivity**

Table of Contents

[Brief Overview 1](#_Toc206649374)

[1 Basic Tools: "Every Journey Begins with the Right Tools" 4](#_Toc206649375)

[1.1 PGAdmin 4](#_Toc206649376)

[1.2 PSQL and PGCLI 12](#_Toc206649377)

[2 Effective Schema Management: "Building the Foundation" 17](#_Toc206649378)

[2.1 Custom Types 17](#_Toc206649379)

[2.2 Function vs Procedure 22](#_Toc206649380)

[2.3 Function Volatility Classification 24](#_Toc206649381)

[2.4 Security Definer vs Security Invoker 26](#_Toc206649382)

[2.5 Exceptions 27](#_Toc206649383)

[3 Monitoring & Performance: "Keeping Your Database Healthy" 29](#_Toc206649384)

[3.1 HypoPG 29](#_Toc206649385)

[3.2 pg\_stat\_statements 34](#_Toc206649386)

[3.3 Plprofiler 40](#_Toc206649387)

[4 Data Management Features: "Scaling for Growth" 43](#_Toc206649388)

[4.1 Partition Management 43](#_Toc206649389)

[4.2 pg\_partman 50](#_Toc206649390)

[4.3 pg\_cron 55](#_Toc206649391)

[4.4 pg\_repack 57](#_Toc206649392)

[4.5 Foreign Data Wrappers (FDWs) 61](#_Toc206649393)

[5 Development & Quality Tools: "Ensuring Code Quality" 68](#_Toc206649394)

[5.1 Orafce 68](#_Toc206649395)

[5.2 plpgsql\_check 70](#_Toc206649396)

[6 Conclusion 79](#_Toc206649397)

## 

## Brief Overview

PostgreSQL is a powerful, open-source relational database management system (RDBMS) known for its robustness, extensibility, and compliance with SQL standards. PostgreSQL is a powerful open-source database, but its true potential is often underused due to lack of awareness around its built-in capabilities and ecosystem tools. This session focuses on the minimum essential features, tools, extensions, and best practices that can significantly improve the day-to-day productivity of both developers and DBAs. Whether you’re building applications, writing queries, or managing database infrastructure, this session gives you the practical knowledge needed to work smarter and faster with PostgreSQL. By mastering these features and tools, developers and DBAs can significantly improve their productivity when working with PostgreSQL. This knowledge will enable them to design efficient database schemas, optimize query performance, ensure data security, and maintain high availability of PostgreSQL databases in various environments.

#### PostgreSQL Database Developer – Roles & Responsibilities

A PostgreSQL Developer focuses on designing, writing, and optimizing SQL code, schemas, and database-side logic.  
**Key Responsibilities:**

* SQL & PL/pgSQL Development
  + Implement business logic in database layers.
  + Write complex queries, stored procedures, triggers, and functions.
  + Maintain coding standards for SQL.
  + Performance Tuning
* Schema Design & Data Modeling
  + Create normalized and optimized table structures.
  + Define primary/foreign keys, constraints, and indexes.
* Ensure efficient database access patterns and performance.
* Database migration, Data Migration
* ETL

#### PostgreSQL DBA (Database Administrator) – Roles & Responsibilities

A Postgres DBA ensures availability, performance, security, and recoverability of the database.

**Key Responsibilities:**

* Database Installation & Configuration
* Plan storage, CPU, memory needs & Upgrades
* Install PostgreSQL binaries, configure database parameters (shared\_buffers, work\_mem, wal\_buffers etc.)
* Backup & Recovery
* High Availability & Replication
* Configure streaming replication, logical replication, failover tools (Patroni, repmgr).
* Monitoring & Maintenance
* Database Maintenance (VACUUM, ANALYZE, and bloat removal)
* Security & Compliance
* Manage roles, privileges, and SSL
* Enforce encryption and audit logging.

#### How Responsibilities Shift in the Cloud

When PostgreSQL runs in managed cloud services (RDS, Aurora, CloudSQL, Azure PostgreSQL), the responsibility boundary shifts:

|  |  |  |
| --- | --- | --- |
| Area | On-Prem | DBA Cloud (Managed PostgreSQL) |
| Infrastructure Setup | Install OS, PostgreSQL binaries, tune kernel | Handled by Cloud Provider |
| Patching & Upgrades | DBA responsibility | Cloud automates minor patching, DBAs plan major upgrades |
| Backups | Custom setup (pgBackRest, cron jobs) | Automated snapshots, point-in-time restore provided |
| High Availability | Setup replication, failover, cluster tools | Built-in HA options (multi-AZ, read replicas) |
| Scaling | Manual (hardware changes, tuning) | ertical/horizontal scaling via UI/CLI |
| Monitoring | Custom Prometheus, pgBadger, Nagios | Cloud metrics (CloudWatch, Azure Monitor, Stackdriver) |
| Security | Configure firewalls, SSL, encryption, auditing | Shared responsibility:  Cloud secures infra, DBA secures data/roles |
| Performance Tuning | Still DBA role (queries, indexes, params allowed) | Same role, but limited access to OS-level tuning |
| Disaster Recovery | Custom standby sites, scripts | Cross-region replication / backups configurable via console |
| Cost Optimization | Not a factor (on-prem hardware sunk cost) | Important responsibility (instance sizing, storage, IOPS) |

Whether your PostgreSQL database is on-premises, in the cloud, or in a hybrid environment, this training equips you with essential tools and techniques to enhance your database management capabilities.

#### What you will learn from this training?

The training focuses on providing practical knowledge needed to work smarter and faster with PostgreSQL.

* Essential tools that save time for every Developer/DBA
  + PGAdmin: A powerful GUI tool for database management
  + PGCLI: An enhanced command-line interface for PostgreSQL
* Useful Extension for regular use and helpful for PostgreSQL database enthusiast.
  + HypoPG: For testing hypothetical indexes without creating them
  + pg\_stat\_statements: For query performance monitoring
  + PLProfiler: For profiling PL/pgSQL code
  + Orafce: For Oracle compatibility functions
  + plpgsql\_check: For semantic checks on PL/pgSQL code
  + pg\_repack: For online table reorganization
  + Database Lab Engine: For quick database cloning
  + pgbench: For database performance bench marking

## 1 Basic Tools: "Every Journey Begins with the Right Tools"

### 1.1 PGAdmin

pgAdmin is a powerful, open-source administration and management Graphical User Interface (GUI) tool for PostgreSQL databases. It provides enterprise-grade features to both novice and expert users, increases productivity and reduces human error. Because it is GUI, you get benefits of:

* User-friendly interface for database management.
* Visual representation of database objects.
* Reduced learning curve compared to command-line tools.
* Intuitive navigation through database structures.

#### Key Features That Make pgAdmin Essential

Key features include a user-friendly graphical interface, SQL query tools with syntax highlighting and autocompletion, database object management (creating, deleting, and modifying databases, tables, views, etc.), and robust backup and restore functionalities. pgAdmin also offers features for monitoring database performance, managing user roles and permissions, and scheduling maintenance tasks. You get active community support, regular updates, bug fixes, security patches and extensive documentation.

#### Managing Servers and connections

pgAdmin allows users to manage servers and connections through its graphical user interface. Use the *Server* dialog to describe a connection to a server.

*Note: you must ensure that the pg\_hba.conf file of the server from which you are connecting allows connections from the host of the client.*

**Server Groups:**

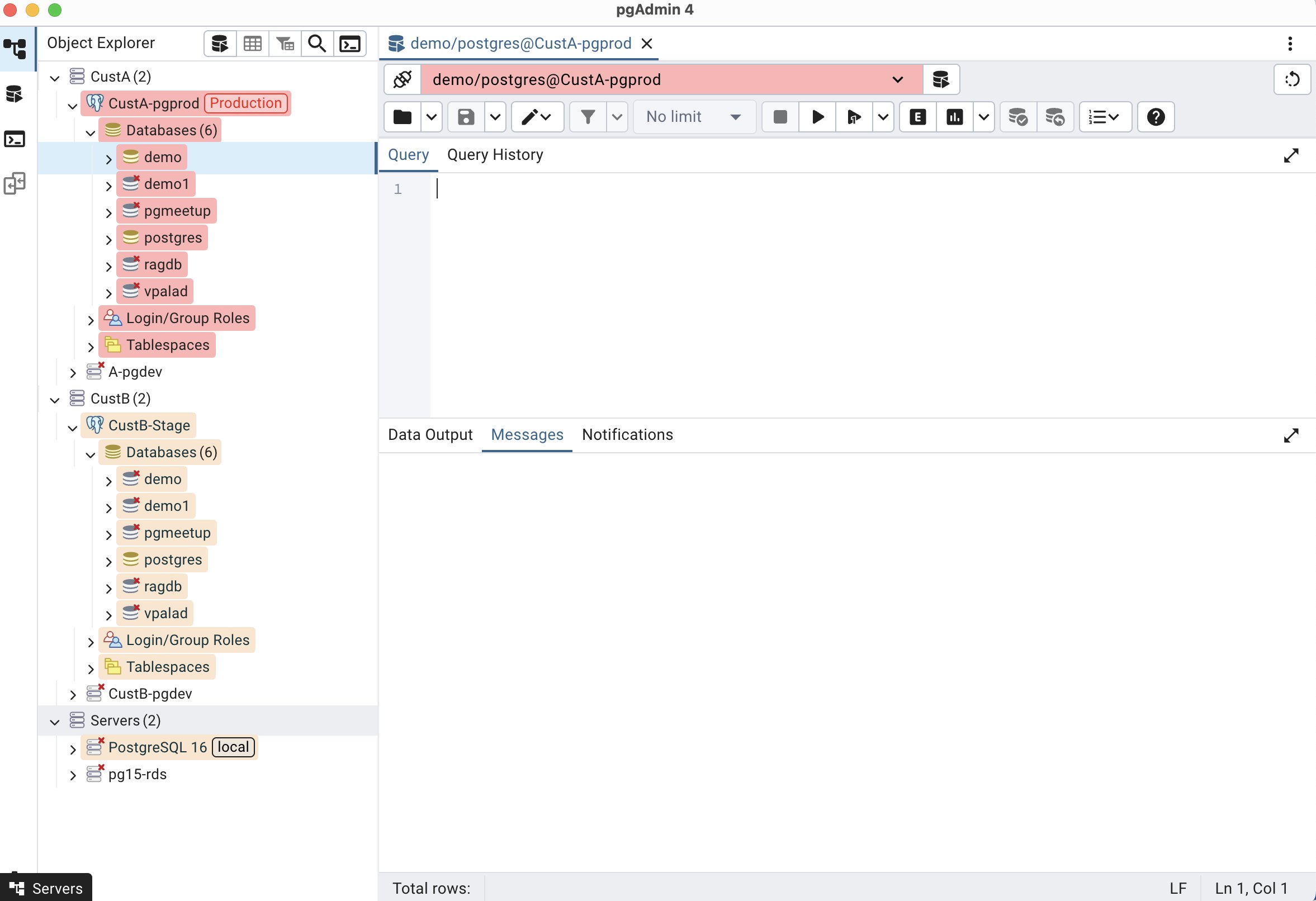
You can organize your server connections by creating server groups. Right-click on the "Servers" node and select "Server Group" from the "Create" menu to create a new group. Then, you can drag and drop registered servers into these groups.

**Tags:**

Use the table in the *Tags* tab to add tags. The tags will be shown on the right side of a server node label in the object explorer tree.

Click on the *+* button to add a new tag. Some of the parameters are:

* *Text* field to specify the tag name.
* *Color* field to select the accent color of the tag.
* Click the *Save* button to save your work.



#### Server Health Monitoring and Dashboards

pgAdmin offers built-in dashboards that provide a high-level overview of your PostgreSQL server and database performance. While not a full-fledged business intelligence (BI) tool for complex data analysis and custom visualizations, pgAdmin's dashboards are useful for specific monitoring and diagnostic use cases:

Server Health Monitoring:

* The main dashboard provides real-time insights into server activity, including CPU usage, memory consumption, disk I/O, and active connections. This allows administrators to quickly identify potential resource bottlenecks or unusual activity.

Database Performance Overview:

* Dashboards for individual databases display key metrics such as transaction rates, query execution times, and lock contention. This helps in understanding the workload on specific databases and identifying slow queries or contention issues.

Ad-hoc Performance Analysis:

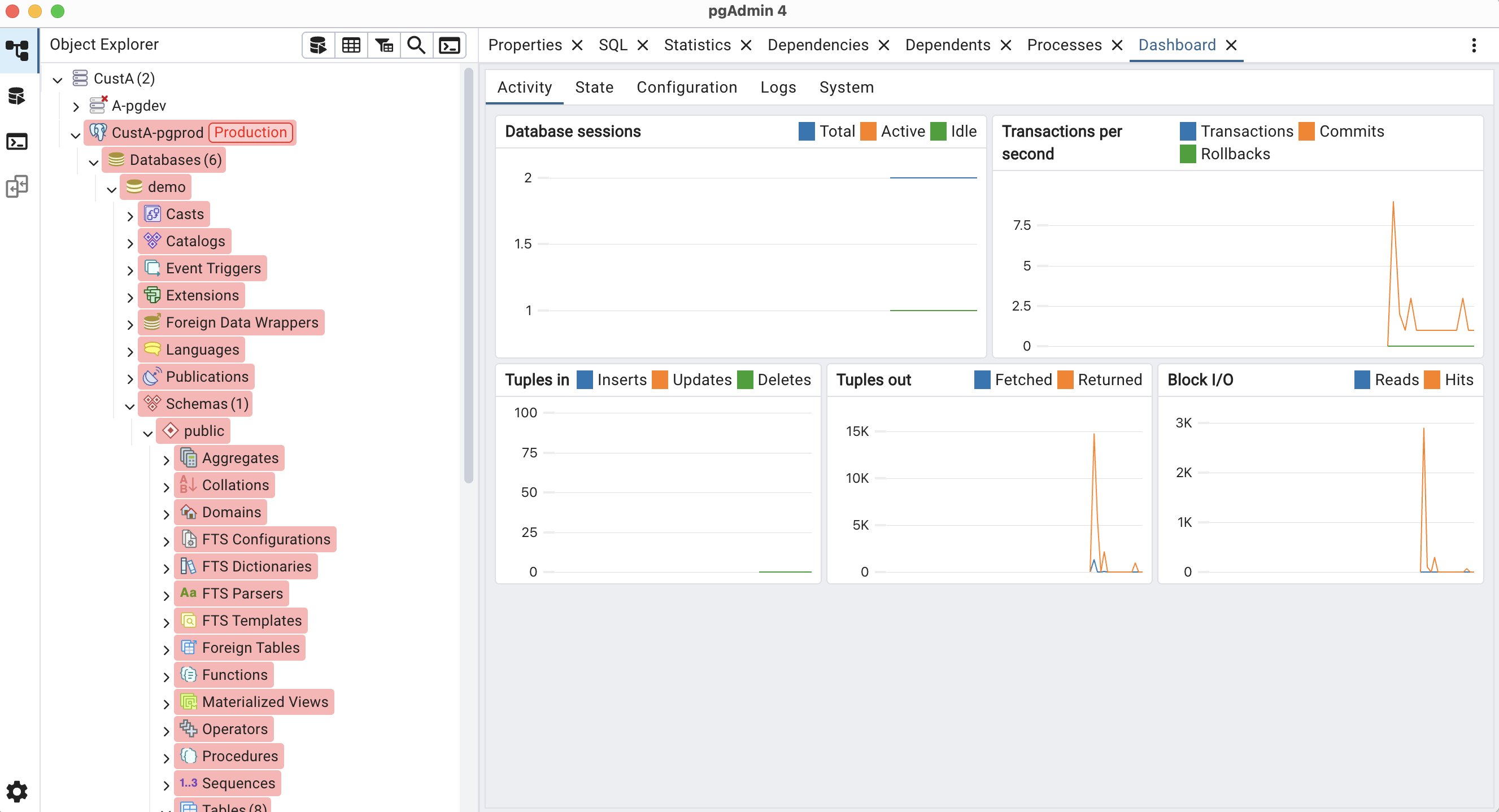
* Users can customize the dashboard settings to focus on specific metrics or timeframes, enabling ad-hoc analysis of performance trends or the impact of recent changes.

Connection and Session Management:

* The dashboard provides information about active connections and sessions, including details about connected users, client applications, and the queries being executed. This aids in monitoring user activity and troubleshooting connection-related issues.

Quick Diagnostics:

* When a performance issue arises, the pgAdmin dashboards offer a quick way to assess the current state of the server and databases, providing immediate clues for further investigation.



**Server activity** shows the server or database information in 4 different tabs.

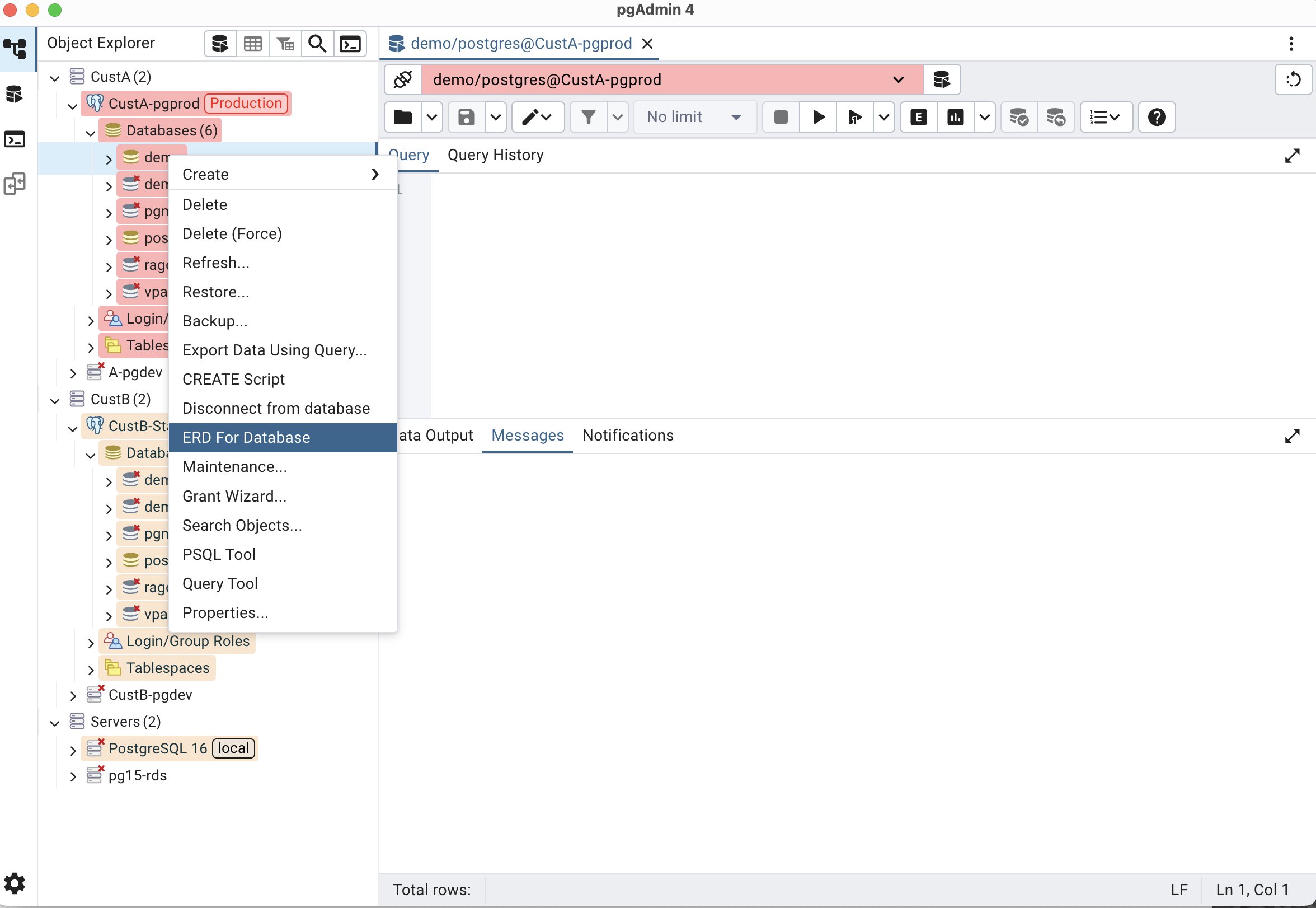
* Sessions
* Locks
* Prepared Transactions
* Configuration (only shown for Servers)

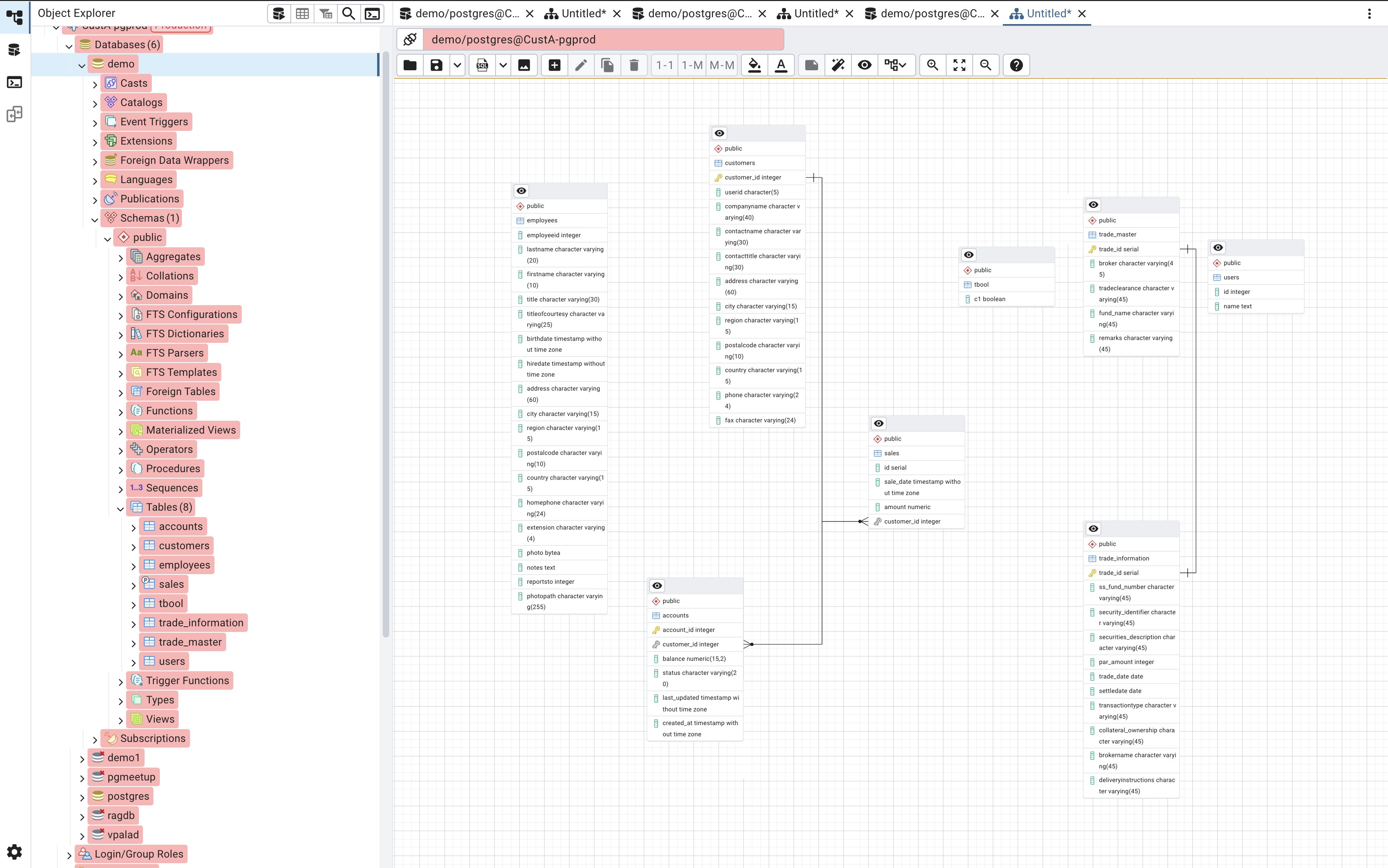


#### Generating ER diagram

Generating an ERD from an Existing Database/Table:

* **Open pgAdmin and Connect:** Launch pgAdmin and connect to your PostgreSQL server.
* **Navigate to Database/Table:** In the browser tree, locate the specific database or table for which you want to generate the ERD.
* **Generate ERD:**
  + **For an entire database:** Right-click on the database name and select "ERD for database".
  + **For a specific table and its related tables:** Right-click on the desired table and select "ERD for Table". This option will include related tables based on foreign key relationships. You can usually control the recursion depth for related tables if needed.





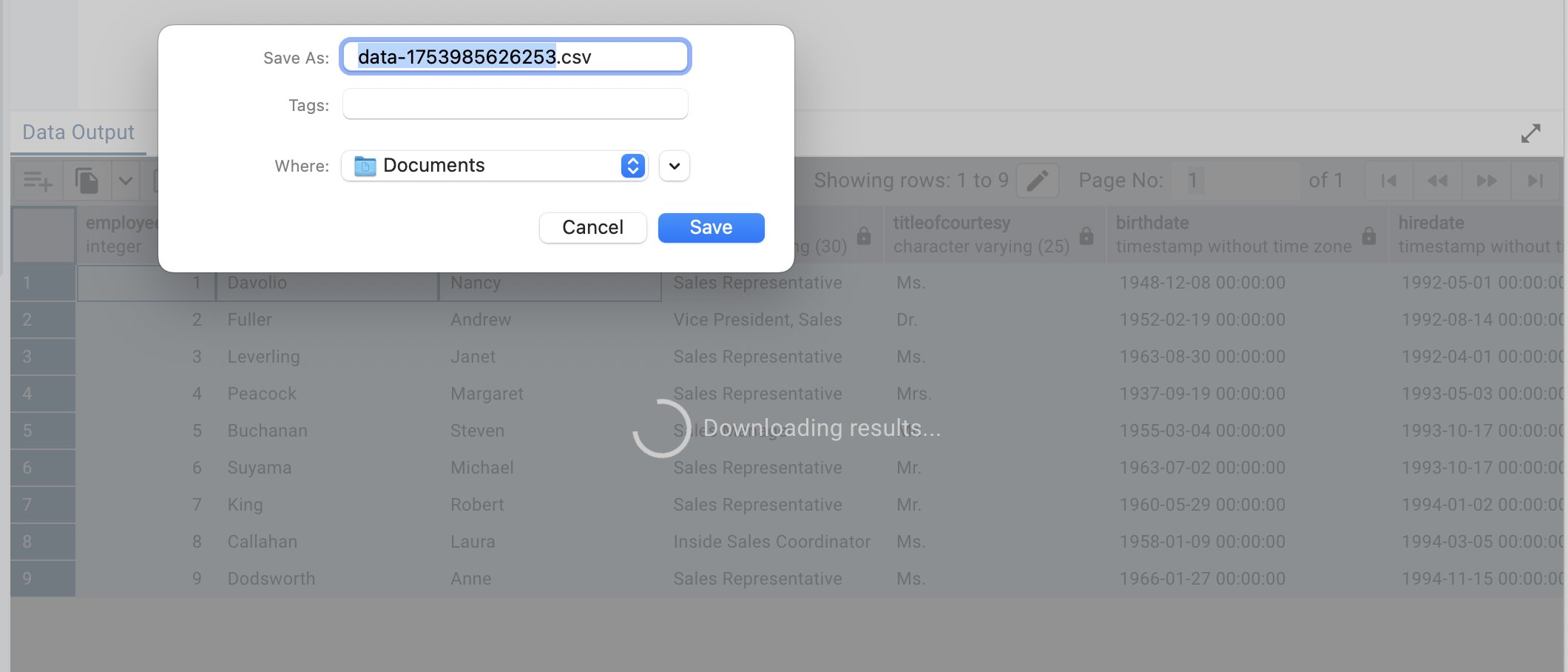
#### Different ways for Data Export & Import

**Option1:**

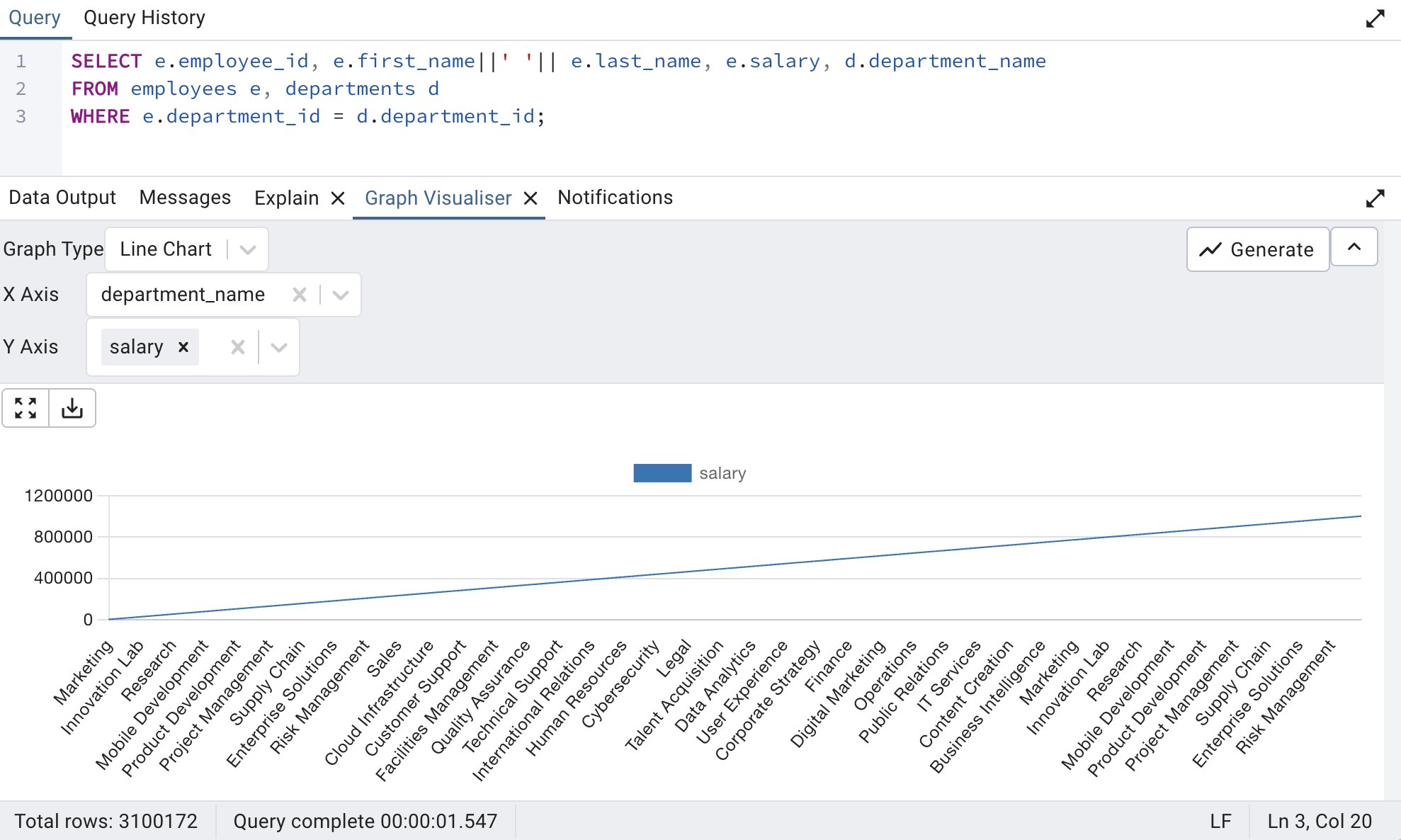


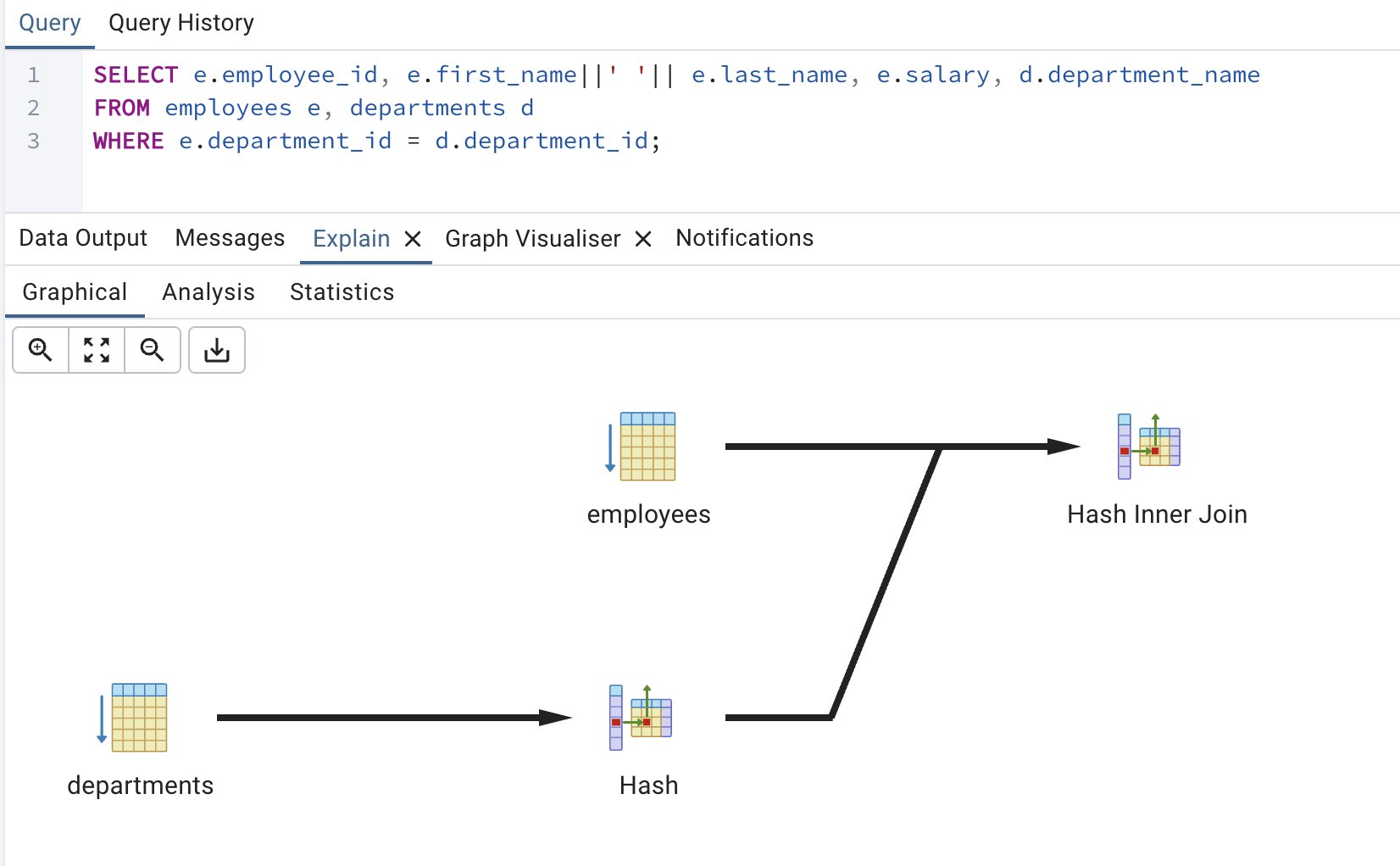
**Option2:**





#### Graph Visualiser and Explain

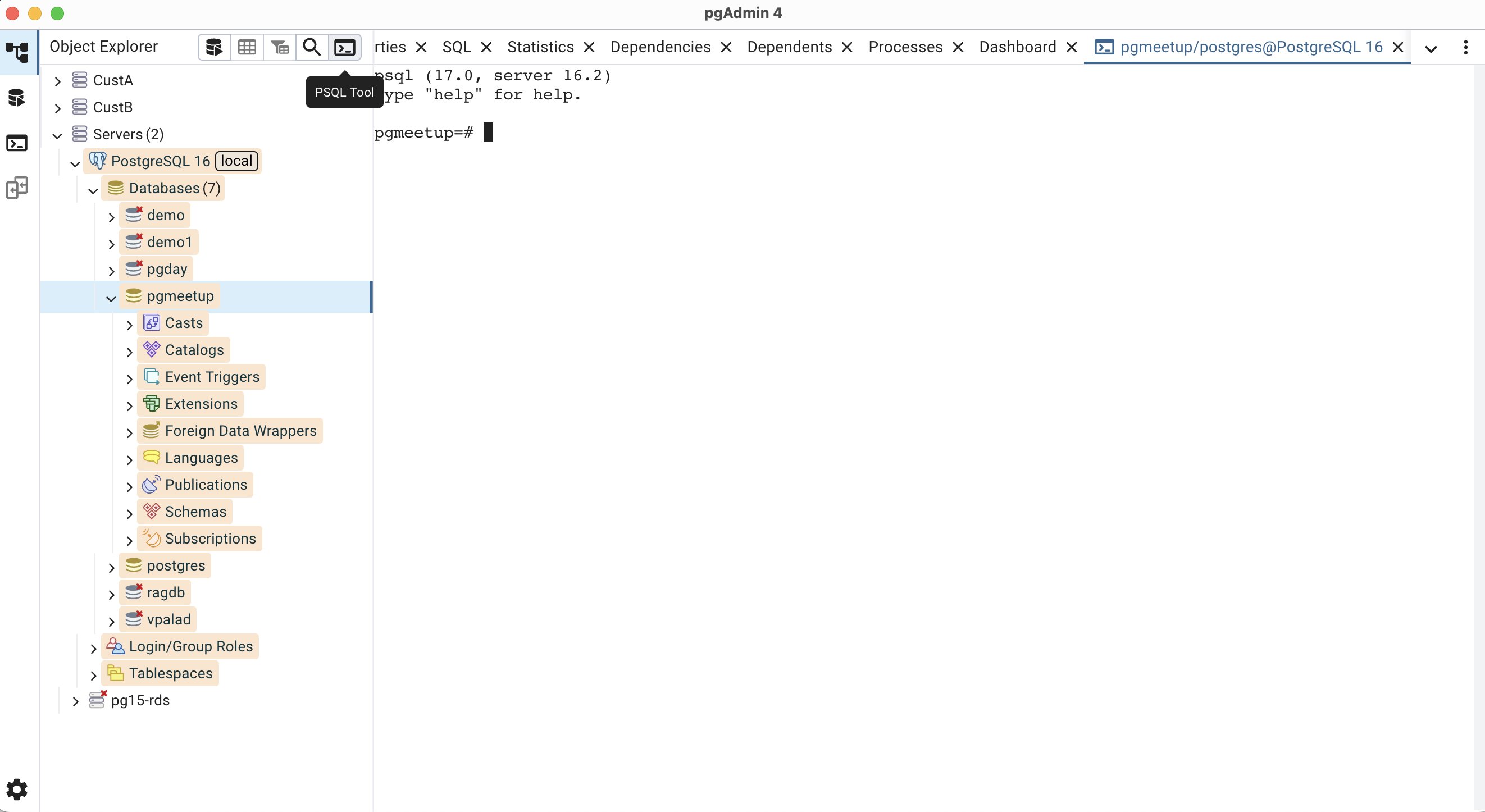




#### Connect to psql client from PGAdmin

**Access the PSQL Tool**:

* Once connected to a server in pgAdmin, you can find the PSQL Tool. It is usually available from the "Tools" menu or by right-clicking on a database or server in the object explorer and selecting "PSQL Tool".
* Alternatively, you can often find a dedicated "PSQL tool" button at the top of the object explorer.



### 1.2 PSQL and PGCLI

#### PSQL

psql is the command-line interface (CLI) or terminal-based front-end for the PostgreSQL database system. It allows users to interact with PostgreSQL databases directly from the command line.

#### Identify background catalog query run by PSQL commands:

* In PostgreSQL's psql interactive terminal, \E is a command-line option, not a meta-command used within a psql session.
* The \E option, when used at the command line when starting psql, is equivalent to --echo-hidden.  
   Its purpose is to echo the actual queries generated by \d and other backslash commands. This allows users to observe the underlying SQL statements that psql executes to perform its various meta-commands, which can be useful for understanding psql's internal operations or for learning how to construct similar SQL queries directly.
* For example, if you were to run psql -E and then use \dt to list tables, psql would not only display the table list but also print the SELECT query it used to retrieve that information from the database's system catalogs.

-- Connect to database with \E  
psql -d pgday -E  
  
-- Now execute \dt and see the background catalog query  
pgday=# \dt  
/\*\*\*\*\*\*\*\* QUERY \*\*\*\*\*\*\*\*\*/  
SELECT n.nspname as "Schema",  
  c.relname as "Name",  
  CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHEN 'i' THEN 'index' WHEN 'S' THEN 'sequence' WHEN 't' THEN 'TOAST table' WHEN 'f' THEN 'foreign table' WHEN 'p' THEN 'partitioned table' WHEN 'I' THEN 'partitioned index' END as "Type",  
  pg\_catalog.pg\_get\_userbyid(c.relowner) as "Owner"  
FROM pg\_catalog.pg\_class c  
     LEFT JOIN pg\_catalog.pg\_namespace n ON n.oid = c.relnamespace  
     LEFT JOIN pg\_catalog.pg\_am am ON am.oid = c.relam  
WHERE c.relkind IN ('r','p','')  
      AND n.nspname <> 'pg\_catalog'  
      AND n.nspname !~ '^pg\_toast'  
      AND n.nspname <> 'information\_schema'  
  AND pg\_catalog.pg\_table\_is\_visible(c.oid)  
ORDER BY 1,2;  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
                     List of relations  
 Schema |            Name             | Type  |   Owner  
--------+-----------------------------+-------+------------  
 public | pl\_profiler\_saved           | table | plprofiler  
 public | pl\_profiler\_saved\_callgraph | table | plprofiler  
 public | pl\_profiler\_saved\_functions | table | plprofiler  
 public | pl\_profiler\_saved\_linestats | table | plprofiler  
(4 rows)

Enhance your psql client experience using psqlrc file:

* **~/.psqlrc** is the default user configuration file for [psql](https://pgpedia.info/p/psql.html).
* On UNIX/UNIX-like systems, this file will be in the user's home directory.
* On Windows, the user's configuration file is named %APPDATA%\postgresql\psqlrc.conf.
* Search psqlrc file is found or can be created in the user’s home directory.
* Ref:
  + Get more details → <https://pgpedia.info/p/psqlrc.html>
  + <https://www.digitalocean.com/community/tutorials/how-to-customize-the-postgresql-prompt-with-psqlrc-on-ubuntu-14-04>

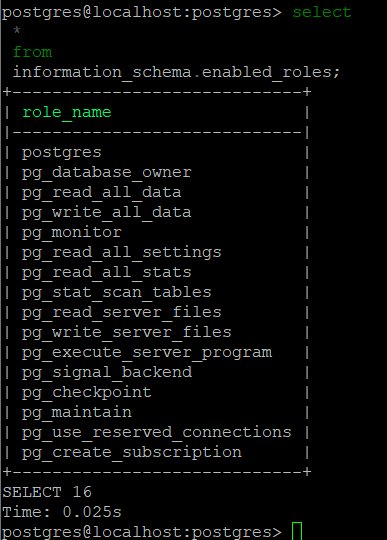
#### PGCLI

PGCLI is an interactive command-line interface for PostgreSQL that enhances the user experience compared to the default psql client. While both pgcli and psql allow interaction with PostgreSQL databases, pgcli offers several features aimed at improving productivity and readability.

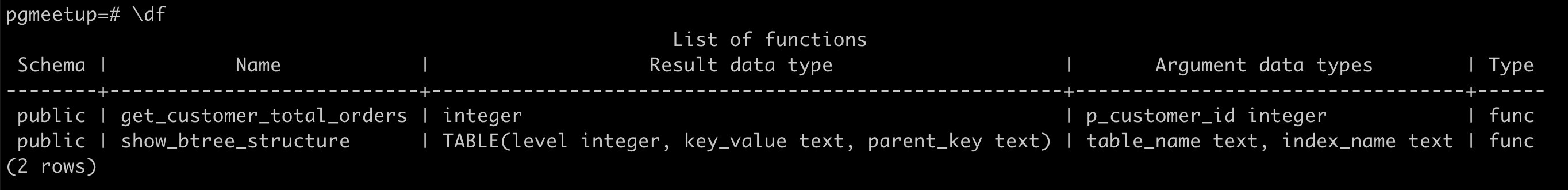
* PGCLI is a [Python package](https://pypi.org/project/pgcli/) that is used as an interactive command-line interface for the PostgreSQL database server.

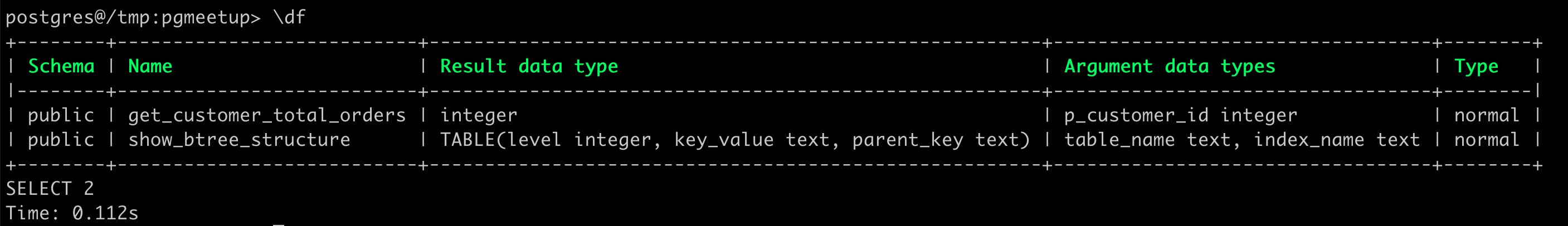
**Key Features of PGCLI**

* **Auto-completion:** Provides real-time suggestions for SQL keywords, table names, column names, and other database objects as the user types. This includes context-sensitive "smart completion."
* **Syntax Highlighting:** Color-codes SQL commands for improved readability and easier identification of different elements within a query.
* **Multi-Line Editing:** Facilitates writing and editing complex, multi-line SQL queries within the terminal, offering a more flexible input experience than psql's default single-line input.



* **Pretty Printing of Tabular Data:** Presents query results in a well-formatted, easy-to-read table layout.
  + psql vs pgcli:





* Primitive Support for psql Backslash Commands: Allows the use of some familiar psql meta-commands within pgcli.
* **Fuzzy Filter on History:** Enables quick searching and retrieval of previously executed commands from the history.
* **Auto Switch to Vertical Output:** Automatically adjusts the display of results to a vertical format when a row exceeds the terminal width, enhancing readability for wide tables.

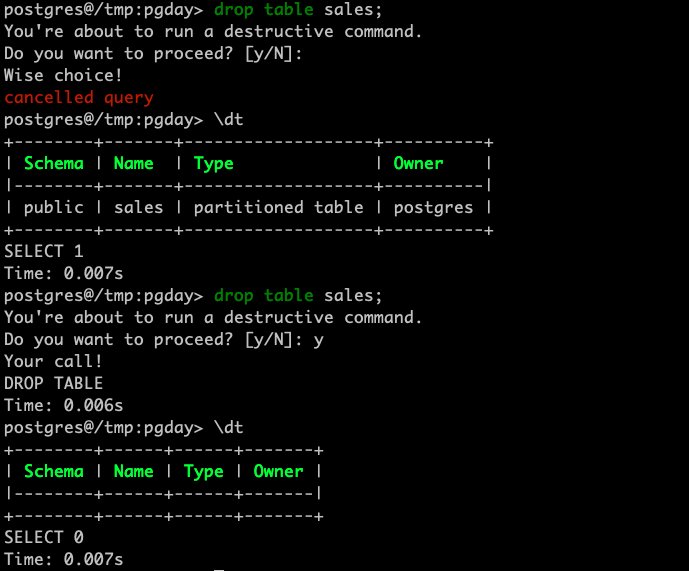
**Destructive Command Warning!**

* Nice and cool built-in feature of PGCLI.

-- Create parent table  
CREATE TABLE sales (  
    id serial,  
    sale\_date timestamp NOT NULL,  
    amount decimal,  
    customer\_id int  
) PARTITION BY RANGE (sale\_date);



* Run a DROP TABLE command.



**Differences from psql:**

The primary distinction lies in pgcli's focus on user-friendliness and interactive features that psql either lacks or requires more manual configuration to achieve. While psql is a robust and widely used client, pgcli provides out-of-the-box enhancements like:

* **Real-time Auto-completion:** psql requires pressing the Tab key for completion, whereas pgcli offers suggestions as you type.
* **Built-in Syntax Highlighting:**psql does not have native syntax highlighting and would require external tools or terminal configurations to achieve a similar effect.
* **Streamlined Multi-line Editing:**pgcli's multi-line mode is more intuitive for complex queries compared to psql's default behavior, which can prematurely execute queries on newline or semicolon input.
* **Enhanced Output Formatting:**pgcli's pretty printing and automatic vertical output for wide results offer a more visually appealing and readable experience than psql's standard output.

In essence, pgcli aims to provide a more modern and interactive command-line experience for PostgreSQL users, particularly those who prefer working within the terminal but desire features commonly found in graphical database clients.

**Ref:**

* <https://github.com/dbcli/pgcli>
* <https://www.dbi-services.com/blog/simplifying-postgresql-management-a-guide-to-install-and-use-pgcli/>

## 2 Effective Schema Management: "Building the Foundation"

### 2.1 Custom Types

##### Domains vs Types

Domains add constraints and validation rules to existing data types while maintaining the same structure, whereas Types create entirely new data structures with their own components and can include custom operations and methods.

**Using Domain (for email validation):**

**-- Create a domain for email**

CREATE DOMAIN email\_address AS VARCHAR(255)

CHECK (VALUE ~ '^[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$');

**-- Use the domain in tables**

CREATE TABLE users (

id serial PRIMARY KEY,

name text,

email email\_address -- Using the domain

);

**-- This will work**

INSERT INTO users (name, email) VALUES ('John', 'john@example.com');

**-- This will fail (invalid email)**

INSERT INTO users (name, email) VALUES ('John', 'not-an@email');

**Using Custom Type (for complex data):**

**-- Create a custom type for address**

CREATE TYPE address AS (

street VARCHAR(100),

city VARCHAR(50),

state VARCHAR(2),

zip VARCHAR(10)

);

**-- Use the type in tables**

CREATE TABLE customers (

id serial PRIMARY KEY,

name text,

delivery\_address address,

billing\_address address

);

**-- Insert using the custom type**

INSERT INTO customers (name, delivery\_address)

VALUES ('John', **('123 Main St', 'Boston', 'MA', '02108')**);

Key Difference:

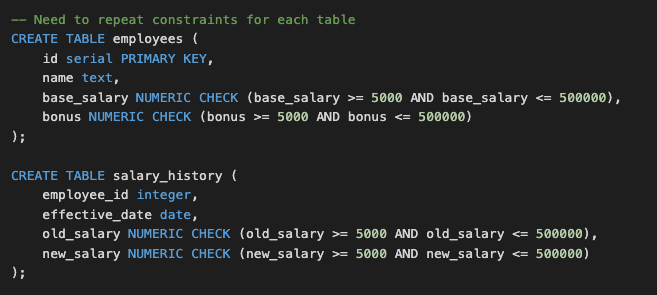
* Domain (email\_address) adds validation to existing type
* Custom Type (address) creates new data structure with multiple components

##### Domains vs Check constraints

Use Case: Employee Salary Management System

You want to enforce consistent salary ranges across different positions, tracking salary changes over time, and managing departmental budgets. The system must prevent invalid salary entries (below $5,000 or above $500,000) while providing flexibility for different types of compensation including base salary and bonuses. Additionally, the organization requires the ability to audit salary changes, understand historical trends, and maintain compliance with salary policies. The system should support organizational structure through departments and their respective budgets, while ensuring all salary-related data remains consistent and validated across the entire database.

**Using Check Constraints:**



**-- Need to repeat constraints for each table**

CREATE TABLE employees (

id serial PRIMARY KEY,

name text,

base\_salary NUMERIC CHECK (base\_salary >= 5000 AND base\_salary <= 500000),

bonus NUMERIC CHECK (bonus >= 5000 AND bonus <= 500000)

);

CREATE TABLE salary\_history (

employee\_id integer,

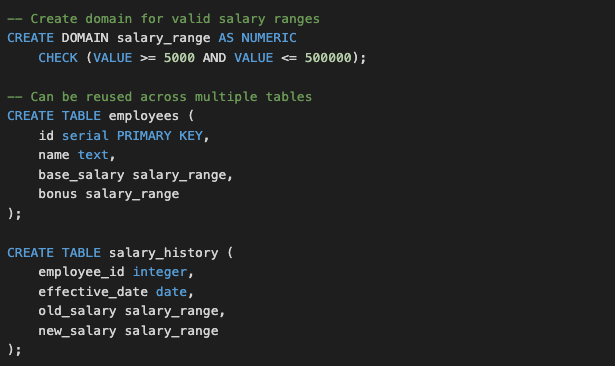
effective\_date date,

old\_salary NUMERIC CHECK (old\_salary >= 5000 AND old\_salary <= 500000),

new\_salary NUMERIC CHECK (new\_salary >= 5000 AND new\_salary <= 500000)

);

**Using Domain:**

****

**-- Create domain for valid salary ranges**

CREATE DOMAIN salary\_range AS NUMERIC

CHECK (VALUE >= 5000 AND VALUE <= 500000);

**-- Can be reused across multiple tables**

CREATE TABLE employees (

id serial PRIMARY KEY,

name text,

base\_salary salary\_range,

bonus salary\_range

);

CREATE TABLE salary\_history (

employee\_id integer,

effective\_date date,

old\_salary salary\_range,

new\_salary salary\_range

);

**-- Shows constraints for the positive\_integer domain**

SELECT

t.typname as domain\_name,

c.conname as constraint\_name,

pg\_get\_constraintdef(c.oid) as constraint\_definition

FROM pg\_type t

JOIN pg\_constraint c ON t.oid = c.contypid

WHERE t.typname = 'positive\_integer';

domain\_name | constraint\_name | constraint\_definition

--------------+--------------------+-----------------------------------------------------------------------

salary\_range | salary\_range\_check | CHECK (((VALUE >= (5000)::numeric) AND (VALUE <= (500000)::numeric)))

(1 row)

**--Check Domain**

pgday=> \dD+ salary\_range

List of domains

Schema | Name | Type | Collation | Nullable | Default | Check | Access privileges | Description

--------+--------------+---------+-----------+----------+---------+-------------------------------------------------------------+-------------------+-------------

public | salary\_range | numeric | | | | CHECK (VALUE >= 5000::numeric AND VALUE <= 500000::numeric) | |

(1 row)

**--Add new Domain constraint**

ALTER DOMAIN salary\_range ADD CONSTRAINT new\_cons\_range CHECK (VALUE >= 10000::numeric AND VALUE <= 600000::numeric) **NOT VALID**;

**NOT VALID** in PostgreSQL, it means the constraint will be created but PostgreSQL will skip validating existing data in the table against this constraint. Only new or updated rows will be checked against the constraint.

The **NOT VALID** option is useful when:

You want to add constraints to large tables without checking existing data

You want to avoid the performance impact of validating all existing rows

You're confident existing data meets the constraint but want to enforce it for new data

pgday=> SELECT

t.typname as domain\_name,

c.conname as constraint\_name,

pg\_get\_constraintdef(c.oid) as constraint\_definition

FROM pg\_type t

JOIN pg\_constraint c ON t.oid = c.contypid

WHERE t.typname = 'salary\_range';

domain\_name | constraint\_name | constraint\_definition

--------------+--------------------+----------------------------------------------------------------------------------

salary\_range | salary\_range\_check | CHECK (((VALUE >= (5000)::numeric) AND (VALUE <= (500000)::numeric)))

salary\_range | new\_cons\_range | CHECK (((VALUE >= (10000)::numeric) AND (VALUE <= (600000)::numeric))) NOT VALID

(2 rows)

**--Drop Domain constraint**

ALTER DOMAIN positive\_integer

DROP CONSTRAINT positive\_integer\_check;

Choosing Domains:

* Maintenance Performance
  + Domains are more efficient to maintain as changes only need to be made once
  + Check constraints require updating multiple tables individually, which can be time-consuming and error-prone
* Database Object Management
  + Domains create additional database objects that need to be tracked
  + This might slightly increase the size of system catalogs
  + The impact is negligible in most practical applications

The choice between domains and check constraints should primarily be based on code organization and maintainability rather than performance considerations, as the performance difference is minimal in real-world applications.

### 2.2 Function vs Procedure

Functions and procedures allow the bundling of complex SQL statements and procedural logic into a single, reusable unit. This encapsulates business rules within the database, ensuring data integrity and consistency across applications.

**-- CREATE FUNCTION**

postgres=> \h CREATE FUNCTION

Command: CREATE FUNCTION

Description: define a new function

Syntax:

CREATE [ OR REPLACE ] FUNCTION

name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default\_expr ] [, ...] ] )

[ RETURNS rettype

**| RETURNS TABLE ( column\_name column\_type [, ...] ) ]**

{ LANGUAGE lang\_name

| TRANSFORM { FOR TYPE type\_name } [, ... ]

| WINDOW

**| { IMMUTABLE | STABLE | VOLATILE }**

| [ NOT ] LEAKPROOF

| { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }

| { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }

| PARALLEL { UNSAFE | RESTRICTED | SAFE }

| COST execution\_cost

| ROWS result\_rows

| SUPPORT support\_function

| SET configuration\_parameter { TO value | = value | FROM CURRENT }

| AS 'definition'

| AS 'obj\_file', 'link\_symbol'

| sql\_body

} ...

URL: https://www.postgresql.org/docs/16/sql-createfunction.html

**-- CREATE PROCEDURE**

postgres=> \h CREATE PROCEDURE

Command: CREATE PROCEDURE

Description: define a new procedure

Syntax:

CREATE [ OR REPLACE ] PROCEDURE

name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default\_expr ] [, ...] ] )

{ LANGUAGE lang\_name

| TRANSFORM { FOR TYPE type\_name } [, ... ]

| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER

| SET configuration\_parameter { TO value | = value | FROM CURRENT }

| AS 'definition'

| AS 'obj\_file', 'link\_symbol'

| sql\_body

} ...

URL: <https://www.postgresql.org/docs/16/sql-createprocedure.html>

Key Differences:

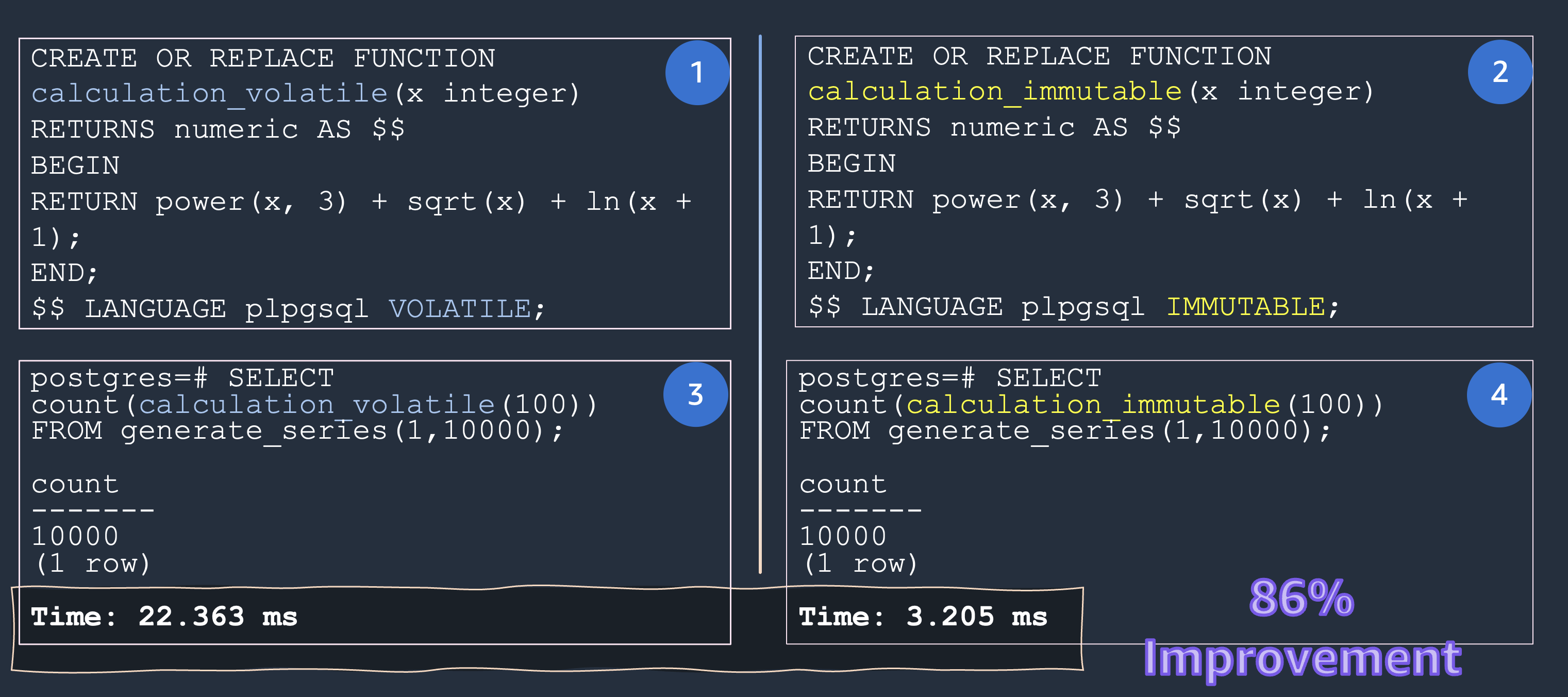
* Return Values
  + Function → Must return something (scalar, set, record, etc.).
  + Procedure → Cannot return a value directly, but can use OUT parameters to pass values back.
* Transaction Control
  + Function → Cannot contain transaction control (COMMIT, ROLLBACK).
  + Procedure → Can manage transactions inside (COMMIT, ROLLBACK allowed).
* Volatility Categories
  + Function → Can be declared IMMUTABLE, STABLE, or VOLATILE (optimizer uses this).
  + Procedure → No such classification; assumed to be side-effecting and not optimizable in queries.
* Usage Context
  + Function → Can be used inside SQL expressions (e.g., in SELECT, WHERE, JOIN, triggers).
  + Procedure → Cannot be used inside SQL expressions; only called explicitly with CALL.

### 2.3 Function Volatility Classification

PostgreSQL implements a volatility classification system that categorizes functions into three distinct levels, as detailed in the following table.

|  |  |  |
| --- | --- | --- |
| **Volatility Classification** | **Use Case** | **Examples** |
| IMMUTABLE | Functions that always return the same output for the same input arguments (such as mathematical calculations) | -- String Functions  length()  upper()  lower()  substr()  replace()  trim()  -- Mathematical Functions  abs()  round()  ceiling()  floor()  mod()  power() |
| STABLE | Functions that return consistent results within a single table scan but might vary across transactions | -- Current Transaction Functions  current\_user  current\_schema  session\_user  current\_timestamp  -- Aggregate Functions  --(when used with non-volatile inputs)  count()  sum()  avg()  min()  max() |
| VOLATILE | Functions that can return different results even with the same inputs | -- Time/Date Functions  timeofday()  clock\_timestamp()  -- Random Functions  random()  gen\_random\_uuid()  -- Sequence Functions  nextval()  currval()  lastval() |

**Performance benefit for properly categorizing volatility classification:**



**--1**

CREATE OR REPLACE FUNCTION calculation\_volatile(x integer)

RETURNS numeric AS $$

BEGIN

RETURN power(x, 3) + sqrt(x) + ln(x + 1);

END;

$$ LANGUAGE plpgsql VOLATILE;

**--2**

CREATE OR REPLACE FUNCTION calculation\_immutable(x integer)

RETURNS numeric AS $$

BEGIN

RETURN power(x, 3) + sqrt(x) + ln(x + 1);

END;

$$ LANGUAGE plpgsql IMMUTABLE;

**--3**

postgres=# SELECT count(calculation\_volatile(100))

FROM generate\_series(1,10000);

count

-------

10000

(1 row)

**--4**

postgres=# SELECT count(calculation\_immutable(100))

FROM generate\_series(1,10000);

count

-------

10000

(1 row)

Properly categorizing functions as VOLATILE, IMMUTABLE, or STABLE allows PostgreSQL’s optimizer to make informed decisions, leading to more efficient query plans. VOLATILE functions offer freshness, STABLE functions balance predictability with flexibility, and IMMUTABLE functions enable the highest degree of optimization through result caching and precomputation. Understanding and using these categories effectively can greatly enhance the performance and scalability of PostgreSQL-based applications.

**Ref:**

<https://aws.amazon.com/blogs/database/volatility-classification-in-postgresql/>

### 2.4 Security Definer vs Security Invoker

In PostgreSQL, SECURITY DEFINER and SECURITY INVOKER are options that determine the privilege context under which a function or procedure executes.

SECURITY INVOKER:

* This is the default behavior for functions and procedures in PostgreSQL.
* When a function or procedure is defined with SECURITY INVOKER, it executes with the privileges of the user who calls or invokes it.
* This means that if the calling user does not have the necessary permissions to access underlying tables or perform certain operations within the function, the function's execution will fail.
* SECURITY INVOKER adheres to the principle of least privilege, as it ensures that operations are performed only with the permissions explicitly granted to the invoking user.

SECURITY DEFINER:

* When a function or procedure is defined with SECURITY DEFINER, it executes with the privileges of the user who owns or created the function.
* This allows a less-privileged user to execute operations that require higher privileges, as long as the function owner has those privileges.
* This can be useful for creating controlled interfaces for specific, privileged operations, such as an audit logging function that writes to a restricted table.
* **Caution:** SECURITY DEFINER functions require careful design and security considerations, as a poorly designed function could potentially be exploited to grant unintended access or privileges. It is crucial to ensure that such functions are as simple as possible and do not allow for arbitrary code execution or privilege escalation.

In summary:

* SECURITY INVOKER: Executes with the privileges of the caller. This is the default and generally more secure.
* SECURITY DEFINER: Executes with the privileges of the owner. Use with caution and only when necessary for specific, controlled scenarios.

### 2.5 Exceptions

In PostgreSQL, exceptions are anomalous or exceptional circumstances that arise during the execution of PL/pgSQL code, such as stored procedures or functions, and disrupt the normal flow of execution. These can include errors like division by zero, constraint violations (e.g., unique constraint violation), insufficient privileges, or invalid data types.

##### Avoid Exceptions

Avoiding exceptions in PostgreSQL, particularly within PL/pgSQL functions, is often recommended due to performance and resource considerations. While exceptions are a valid mechanism for error handling, their overuse can lead to inefficiencies.

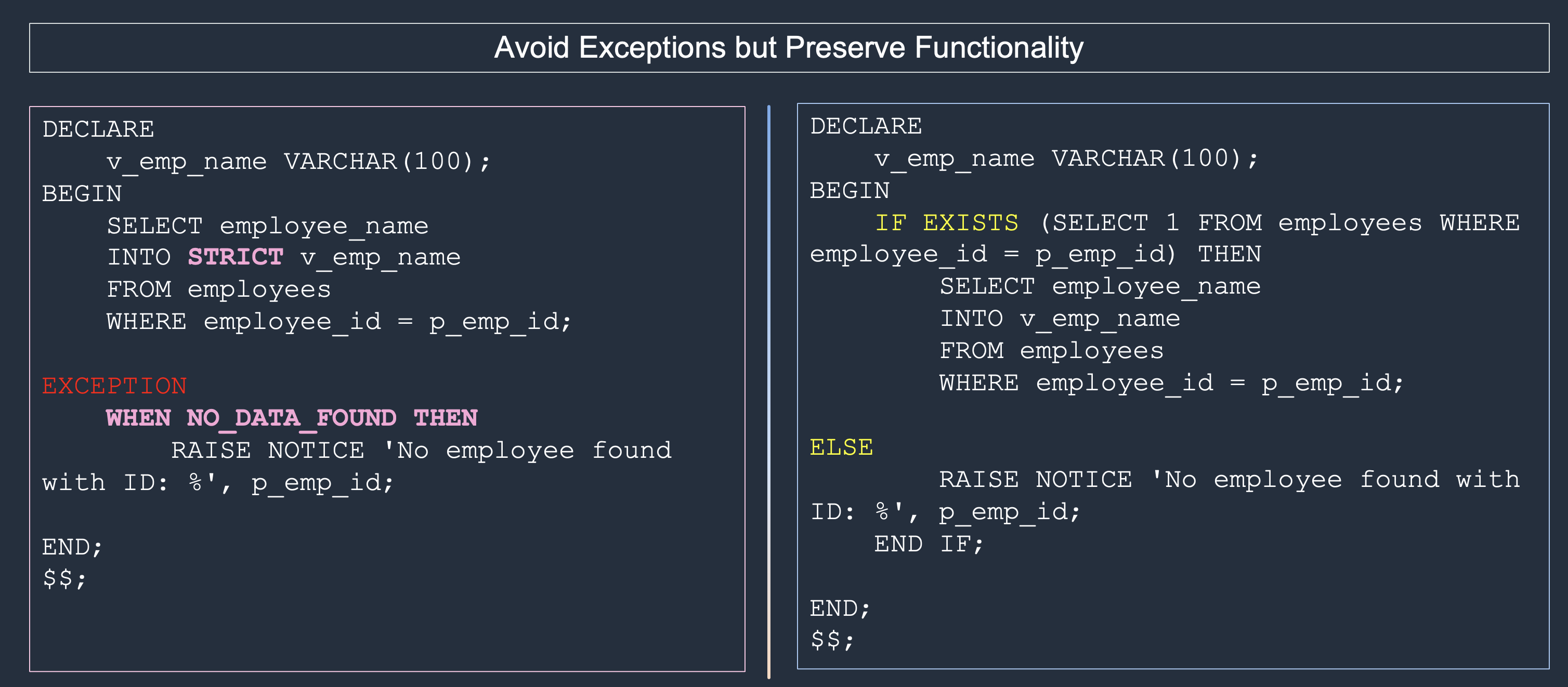
Why to Avoid Overusing Exceptions:

* **Performance Overhead:**

**Sub-transactions and Savepoints:** Each EXCEPTION block in PL/pgSQL implicitly establishes a sub-transaction and sets a savepoint. If an error occurs and is caught, the changes within that block are rolled back to the savepoint before the EXCEPTION block's code executes. This process of creating and managing sub-transactions and savepoints introduces overhead, especially in high-transaction environments.**Resource**

* **Consumption:**

Frequent use of EXCEPTION blocks, especially in high-transaction environments, can consume more resources due to the overhead associated with managing savepoints and potential subtransactions. This can impact overall database performance.



**-- Block with EXCEPTION**

DECLARE

v\_emp\_name VARCHAR(100);

BEGIN

SELECT employee\_name

INTO **STRICT** v\_emp\_name

FROM employees

WHERE employee\_id = p\_emp\_id;

EXCEPTION

**WHEN NO\_DATA\_FOUND THEN**

RAISE NOTICE 'No employee found with ID: %', p\_emp\_id;

END;

$$;

**-- Block with EXCEPTION REMOVED and Refactored**

DECLARE

v\_emp\_name VARCHAR(100);

BEGIN

IF EXISTS (SELECT 1 FROM employees WHERE employee\_id = p\_emp\_id) THEN

SELECT employee\_name

INTO v\_emp\_name

FROM employees

WHERE employee\_id = p\_emp\_id;

ELSE

RAISE NOTICE 'No employee found with ID: %', p\_emp\_id;

END IF;

END;

$$;

While exception handling provides a mechanism to gracefully manage errors and prevent program crashes, its overuse can lead to performance and architectural drawbacks.

## 3 Monitoring & Performance: "Keeping Your Database Healthy"

### 3.1 HypoPG

Indexes are critical for improving query performance in PostgreSQL. They reduce the amount of data scanned, speed up lookups, joins, and aggregations.

Imagine you have a complex database with multiple tables, and you need to optimize a specific query that involves joining several tables and applying various filtering conditions. The query is critical for your application, and you want to improve its performance by creating appropriate indexes.

But… choosing the right index is hard:

* Which column(s) should I index?
* Should I use single-column or composite indexes?
* Will the index be used by the query planner?
* Problem with trial-and-error:
* Large indexes take time to build and lock resources.
* Wrong choices can degrade performance instead of improving it.

This is where HypoPG helps:

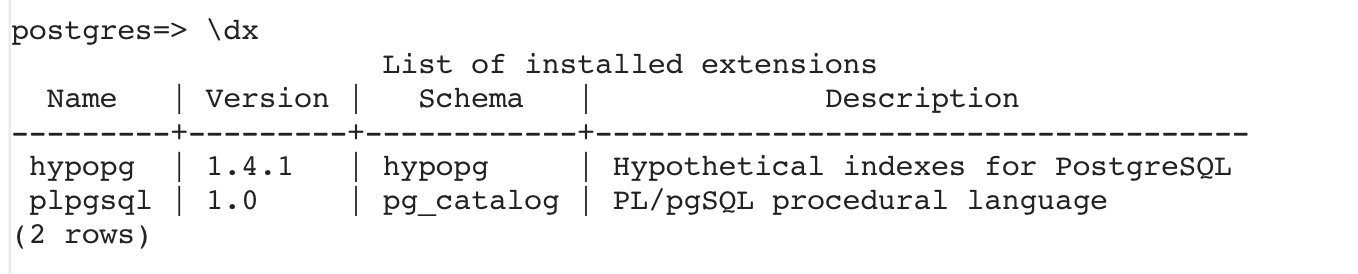
* Allows you to create hypothetical indexes that exist only in memory that are visible to the Postgres query planner.
* have no resource cost (CPU, disk, memory)
* PostgreSQL’s query planner treats them as if they were real, so you can check plans.
* Helps DBAs and developers evaluate indexing strategies safely before committing.
* Instant to create
* Works only with Explain not with Explain Analyze because this doesn’t physically exists
* This only exists in Current Session
* Index size estimation

Here's a clear example of using HypoPG in PostgreSQL:

First, install and create the extension:

CREATE EXTENSION hypopg;

To confirm that the extension is installed successfully, run the following command:  
  
\dx 



#### Create hypothetical indexes in your queries using HypoPG

To create a hypothetical index, use the **hypopg\_create\_index** function.

1.Initial Setup and Base Query Analysis:

-- Create test table with 1M rows

CREATE TABLE testable AS

SELECT id, 'line ' || id val

FROM generate\_series(1,1000000) id;

ANALYZE testable;

-- Check initial query plan

EXPLAIN SELECT \* FROM testable WHERE id < 1000;

No surprise, a sequential scan is the only way to go.

2. Enable HypoPG and Create First Hypothetical Index:

SET hypopg.enabled = on;

-- Create hypothetical index on id

SELECT hypopg\_create\_index('CREATE INDEX ON testable (id)');

-- Check new query plan

EXPLAIN SELECT \* FROM testable WHERE id < 1000;

Output with Hypothetical Index ..

Note : We also notice that the hypothetical index creation is more or less 1ms.

Note : Some information from the CREATE INDEX statement will be ignored, such as the index name if provided. Some of the ignored information will be handled in a future release.

HypoPG gives us in EXPLAIN:

When you create a hypothetical index and rerun EXPLAIN, you will see:

1. Plan changes (e.g., Seq Scan → Index Scan)

2. Cost reduction (lower estimated cost = planner believes its faster)

You can check the available hypothetical indexes in your own backend:

SELECT \* FROM hypopg\_list\_indexes ;

indexrelid | index\_name | schema\_name | table\_name | am\_name

------------+--------------------------+-------------+------------+---------

13630 | <13630>btree\_testable\_id | public | testable | btree

And of course, this hypothetical index is not used in an EXPLAIN ANALYZE:

EXPLAIN ANALYZE SELECT \* FROM testable WHERE id < 1000 ;

QUERY PLAN

----------------------------------------------------------------------------------------------------------------------------

Gather (cost=1000.00..11735.33 rows=870 width=15) (actual time=0.722..54.277 rows=999 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Parallel Seq Scan on testable (cost=0.00..10648.33 rows=362 width=15) (actual time=22.319..39.430 rows=333 loops=3)

Filter: (id < 1000)

Rows Removed by Filter: 333000

Planning Time: 0.078 ms

Execution Time: 54.364 ms

(8 rows)

Let’s test with a new query pattern :

EXPLAIN SELECT \* FROM testable

WHERE id < 1000 and val LIKE 'line 100000%';

QUERY PLAN

---------------------------------------------------------------------------------------------

Index Scan using "<13630>btree\_testable\_id" on testable (cost=0.05..37.45 rows=1 width=15)

Index Cond: (id < 1000)

Filter: (val ~~ 'line 100000%'::text)

(3 rows)

Our hypothetical index is still used, but an index on id and val should help this query. Also, as there’s a wildcard on the right-side of the LIKE pattern, the operator class text\_pattern\_ops is needed.

SELECT hypopg\_create\_index('CREATE INDEX ON testable (id, val text\_pattern\_ops)');

EXPLAIN SELECT \* FROM testable

WHERE id < 1000 and val LIKE 'line 100000%';

QUERY PLAN

------------------------------------------------------------------------------------------------------

Index Only Scan using "<13631>btree\_testable\_id\_val" on testable (cost=0.05..34.94 rows=1 width=15)

Index Cond: ((id < 1000) AND (val ~>=~ 'line 100000'::text) AND (val ~<~ 'line 100001'::text))

Filter: (val ~~ 'line 100000%'::text)

(3 rows)

PostgreSQL decides to use our new index!

To play around with different index Use cases , use **hypopg\_hide\_index(oid)** to hide one of the indexes:

SELECT hypopg\_hide\_index(13631);

hypopg\_hide\_index

-------------------

t

(1 row)

pgday=# EXPLAIN SELECT \* FROM testable

WHERE id < 1000 and val LIKE 'line 100000%';

QUERY PLAN

---------------------------------------------------------------------------------------------

Index Scan using "<13630>btree\_testable\_id" on testable (cost=0.05..37.45 rows=1 width=15)

Index Cond: (id < 1000)

Filter: (val ~~ 'line 100000%'::text)

(3 rows)

HypoPG provides a quick estimation of potential index sizes, which can be valuable for planning purposes. Let's compare these estimates with actual index sizes:

1. View estimated sizes of hypothetical indexes:

pgday=# SELECT indexname,pg\_size\_pretty(hypopg\_relation\_size(indexrelid)) FROM hypopg();

indexname | pg\_size\_pretty

------------------------------+----------------

<13630>btree\_testable\_id | 22 MB

<13631>btree\_testable\_id\_val | 37 MB

(2 rows)

2. Create actual indexes for comparison:

pgday=# CREATE INDEX ON testable (id);

CREATE INDEX

pgday=# CREATE INDEX ON testable (id, val text\_pattern\_ops);

CREATE INDEX

3. Compare sizes of actual indexes:

pgday=# SELECT relname,pg\_size\_pretty(pg\_relation\_size(oid))

FROM pg\_class WHERE relkind = 'i' AND relname LIKE '%testable%';

relname | pg\_size\_pretty

---------------------+----------------

testable\_id\_idx | 21 MB

testable\_id\_val\_idx | 30 MB

(2 rows)

Note: HypoPG's estimates are intentionally slightly higher than actual sizes. This prevents the query planner from preferring hypothetical indexes over existing real indexes, which would be counterproductive.

4.Reset all hypothetical indexes:

SELECT hypopg\_reset();

5. Verify that all hypothetical indexes are removed:

SELECT \* FROM hypopg\_list\_indexes ;

indexrelid | index\_name | schema\_name | table\_name | am\_name

------------+------------+-------------+------------+---------

(0 rows)

#### Limitation of HYPOPG:

Only btree hypothetical indexes are supported

No hypothetical indexes on expression

No hypothetical indexes on predicate

Index size estimation could be improved

#### Best Use Cases

1. Complex query optimization
2. Testing multiple index combinations
3. Quick performance evaluation
4. Index strategy planning
5. Large table optimizations

Although this solution doesn’t apply to every use case, it’s an important tool to add to your toolbox, allowing you quickly evaluate how to get the most out of PostgreSQL.

### 3.2 pg\_stat\_statements

PostgreSQL's pg\_stat\_statements is like having a black box recorder for your database - it tracks every query, showing you exactly what's happening under the hood. Think of it as your database's fitness tracker, measuring how your queries perform in the wild.

**Quick Setup**

1. **Quick Setup**

**-- Enable in postgresql.conf**

shared\_preload\_libraries = 'pg\_stat\_statements'

**-- Create extension**

CREATE EXTENSION pg\_stat\_statements;

**-- Basic configuration parameters**

pg\_stat\_statements.max = 5000 -- Number of statements tracked

pg\_stat\_statements.track = all -- Statements to track

pg\_stat\_statements.track\_utility = on -- Track utility commands

**-- Reset statistics**  
SELECT pg\_stat\_statements\_reset();  
  
select \* from pg\_stat\_statements;  
  
**--create extension if not exists**  
create extension if not exists pg\_stat\_statements;  
  
**--Create table to run few queries aganist this to get the pg\_stat\_statements table to be populated.**  
  
CREATE TABLE salesinfo (  
    id SERIAL PRIMARY KEY,  
    customer\_id INTEGER,  
    product\_id INTEGER,  
    sale\_date DATE,  
    amount DECIMAL(10,2),  
    region VARCHAR(50),  
    payment\_mode VARCHAR(20),  
    status VARCHAR(20)  
);  
  
  
CREATE OR REPLACE FUNCTION generate\_sales\_data(n INTEGER) RETURNS void AS $$  
BEGIN  
    FOR i IN 1..n LOOP  
        INSERT INTO salesinfo (  
            customer\_id,  
            product\_id,  
            sale\_date,  
            amount,  
            region,  
            payment\_mode,  
            status  
        )  
        SELECT  
            floor(random() \* 1000 + 1)::int, -- customer\_id between 1-1000  
            floor(random() \* 100 + 1)::int,  -- product\_id between 1-100  
            current\_date - (random() \* 365)::int, -- random date within last year  
            (random() \* 1000)::decimal(10,2), -- amount between 0-1000  
            (ARRAY['North', 'South', 'East', 'West', 'Central'])[floor(random() \* 5 + 1)], -- random region  
            (ARRAY['Cash', 'Credit Card', 'UPI', 'Net Banking'])[floor(random() \* 4 + 1)], -- random payment mode  
            (ARRAY['Completed', 'Pending', 'Failed', 'Refunded'])[floor(random() \* 4 + 1)] -- random status  
        ;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;  
  
  
**-- Insert 1 million records**  
SELECT generate\_sales\_data(1000000);  
  
  
--CREATE INDEX idx\_sales\_date ON salesinfo(sale\_date);  
--CREATE INDEX idx\_sales\_region ON salesinfo(region);  
--CREATE INDEX idx\_sales\_status ON salesinfo(status);  
  
**-- Install extension**  
CREATE EXTENSION pg\_stat\_statements;  
  
**-- Configure in postgresql.conf**  
pg\_stat\_statements.max = 5000        -- Maximum number of statements tracked  
pg\_stat\_statements.track = all       -- Track all statements (top, all, none)  
pg\_stat\_statements.track\_utility = on -- Track utility commands  
pg\_stat\_statements.save = on         -- Save stats across restarts  
  
--few example queries to run.  
  
-- Query 1: Group by with aggregation  
  
SELECT   
    region,  
    status,  
    COUNT(\*),  
    SUM(amount),  
    AVG(amount)  
FROM salesinfo  
GROUP BY region, status;  
  
-- Query 2: Date range with conditions  
  
SELECT \*  
FROM salesinfo  
WHERE sale\_date BETWEEN current\_date - interval '30 days' AND current\_date  
AND amount > 500  
AND status = 'Completed';  
  
-- Query 3: Complex aggregation  
  
SELECT \* FROM (SELECT   
    DATE\_TRUNC('month', sale\_date) as month,  
    region,  
    payment\_mode,  
    COUNT(\*) as total\_sales,  
    SUM(amount) as total\_amount,  
    AVG(amount) as avg\_amount  
FROM salesinfo  
GROUP BY   
    DATE\_TRUNC('month', sale\_date),  
    region,  
    payment\_mode  
ORDER BY month DESC) a LIMIT 20;  
  
-- Query 4: Window functions  
  
SELECT   
    \*,  
    AVG(amount) OVER (PARTITION BY region) as region\_avg,  
    RANK() OVER (PARTITION BY region ORDER BY amount DESC) as amount\_rank  
FROM salesinfo  
WHERE sale\_date >= current\_date - interval '90 days';  
  
-- Query 5: Subqueries  
  
SELECT   
    s.region,  
    s.status,  
    s.amount,  
    (SELECT AVG(amount) FROM salesinfo WHERE region = s.region) as region\_avg  
FROM salesinfo s  
WHERE amount > (  
    SELECT AVG(amount) FROM salesinfo  
)  
LIMIT 1000;  
  
**2. Common Monitoring Queries**  
  
**-- Top time-consuming queries**  
  
SELECT                 
substring(query, 1, 50) as short\_query,  
    round(total\_exec\_time::numeric, 2) as total\_time,  
    calls,  
    round(mean\_exec\_time::numeric, 2) as mean\_time,  
    round((100 \* total\_exec\_time / sum(total\_exec\_time::numeric) over ())::numeric, 2) as percentage FROM pg\_stat\_statements ORDER BY total\_time DESC  
LIMIT 10;  
  
**--To see thecolumns present in pg\_stat\_statements table.**  
  
\d pg\_state\_statements  
  
-- **Most frequently called queries**  
SELECT  
substring(query, 1, 50) as short\_query,  
    calls,  
    round(total\_exec\_time::numeric, 2) as total\_time,  
    round(mean\_exec\_time::numeric, 2) as mean\_time,  
    rows as total\_rows FROM pg\_stat\_statements ORDER BY calls DESC  
LIMIT 10;  
  
**-- Queries with highest average time**  
  
SELECT  
substring(query, 1, 50) as short\_query,  
    round(mean\_exec\_time::numeric, 2) as mean\_time,  
    calls,  
    round(total\_exec\_time::numeric, 2) as total\_time FROM pg\_stat\_statements WHERE calls > 100  -- Minimum calls threshold  
ORDER BY mean\_time DESC  
LIMIT 10;  
  
  
  
4. Advanced Analysis  
**-- Query IO Statistics**  
SELECT  
substring(query, 1, 50) as short\_query,  
    shared\_blks\_hit,  
    shared\_blks\_read,  
    local\_blks\_hit,  
    local\_blks\_read,  
    temp\_blks\_read,  
    temp\_blks\_written FROM pg\_stat\_statements ORDER BY shared\_blks\_read DESC  
LIMIT 10;  
  
-- Cache Hit Ratio per Query  
SELECT  
substring(query, 1, 50) as short\_query,  
    calls,  
    shared\_blks\_hit,  
    shared\_blks\_read,  
    round(shared\_blks\_hit::numeric /  
nullif(shared\_blks\_hit + shared\_blks\_read, 0) \* 100, 2) as cache\_hit\_ratio FROM pg\_stat\_statements WHERE shared\_blks\_hit + shared\_blks\_read > 0  
ORDER BY cache\_hit\_ratio DESC;  
  
**-- Create monitoring view**  
CREATE VIEW query\_stats AS  
SELECT   
    queryid,  
    calls,  
    total\_exec\_time,  
    mean\_exec\_time,  
    rows,  
    shared\_blks\_hit,  
    shared\_blks\_read,  
    query FROM pg\_stat\_statements WHERE calls > 100;  
      
**-- Find slow queries**  
SELECT query,   
       total\_exec\_time/calls as avg\_time,  
       calls,  
       rows/calls as avg\_rows FROM pg\_stat\_statements WHERE total\_exec\_time/calls > 100  -- milliseconds  
ORDER BY avg\_time DESC;  
  
**-- IO intensive queries**  
SELECT query,  
       shared\_blks\_read + shared\_blks\_written as total\_blocks FROM pg\_stat\_statements ORDER BY total\_blocks DESC;

### 3.3 Plprofiler

**PostgreSQL's** PLProfiler is an extension that helps developers and DBAs dive deep into function or procedure performance and behavior. Whether you're tracking down that elusive performance bottleneck or optimizing your database's efficiency, PLProfiler serves as your go-to tool for detailed analysis.

#### Why Use PLProfiler?

PLProfiler shines in several critical areas. It's particularly effective for performance optimization, helping you identify slow-running code segments with precision. During debugging sessions, it provides invaluable insights into execution paths. For code analysis, it offers line-by-line execution statistics, making it easier to understand complex functions. Additionally, its resource usage tracking capabilities help you monitor and optimize database performance.

#### Best Practices and Tips

When working with PLProfiler, focus on profiling specific functions rather than entire applications – this keeps your analysis targeted and meaningful. Keep your profiling sessions short to minimize overhead and maintain accurate results. Always document your findings and improvements; this creates a valuable knowledge base for future optimizations. Remember to use PLProfiler alongside other monitoring tools for a comprehensive view of your database's performance. Finally, maintain good housekeeping by regularly cleaning up profiling data to prevent unnecessary resource consumption.

**-- Install extension**  
CREATE EXTENSION plprofiler;  
  
**-- Verify installation**  
SELECT \* FROM pg\_extension WHERE extname = 'plprofiler';  
  
**-- Reset local profiling data**  
SELECT pl\_profiler\_reset\_local();  
  
**-- Get line statistics (local session)**  
SELECT \* FROM pl\_profiler\_linestats\_local();  
  
**-- Check if enabled locally**  
SELECT pl\_profiler\_get\_enabled\_local();  
  
**-- Enable profiling locally**  
SELECT pl\_profiler\_set\_enabled\_local(true);  
  
**--create a sample function and execute them**  
  
CREATE OR REPLACE FUNCTION public.test\_function()  
RETURNS void AS $$  
BEGIN  
    FOR i IN 1..1000 LOOP  
        PERFORM pg\_sleep(0.001);  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;  
  
select \* from public.test\_function();  
  
CREATE OR REPLACE FUNCTION public.test\_function\_v2()  
RETURNS integer AS $$  
DECLARE  
    counter integer := 0;  
BEGIN  
    FOR i IN 1..1000 LOOP  
        counter := counter + 1;  
        PERFORM pg\_sleep(0.001);  
    END LOOP;  
    RETURN counter;  
END;  
$$ LANGUAGE plpgsql;  
  
select \* from public.test\_function\_v2();  
  
**--queries to analyze profiling data**  
  
WITH function\_summary AS (  
    SELECT   
        p.proname,  
        COUNT(DISTINCT l.line\_number) as lines\_of\_code,  
        SUM(l.exec\_count) as total\_executions,  
        SUM(l.total\_time)/1000000.0 as total\_time\_ms,  
        MAX(l.longest\_time)/1000000.0 as max\_line\_time\_ms  
    FROM pl\_profiler\_linestats\_local() l  
    JOIN pg\_proc p ON l.func\_oid = p.oid  
    GROUP BY p.proname  
)  
SELECT   
    proname as "Function",  
    lines\_of\_code as "LOC",  
    total\_executions as "Executions",  
    round(total\_time\_ms::numeric, 2) as "Total Time (ms)",  
    round((total\_time\_ms/NULLIF(total\_executions,0))::numeric, 4) as "Avg Time (ms)",  
    round(max\_line\_time\_ms::numeric, 2) as "Max Line Time (ms)"  
FROM function\_summary  
ORDER BY total\_time\_ms DESC;  
  
**-- Get line-level statistics**  
SELECT   
    p.proname as function\_name,  
    l.line\_number,  
    l.exec\_count,  
    (l.total\_time/1000000.0)::numeric(10,2) as total\_time\_ms,  
    (l.longest\_time/1000000.0)::numeric(10,2) as longest\_time\_ms  
FROM pl\_profiler\_linestats\_local() l  
JOIN pg\_proc p ON l.func\_oid = p.oid where p.proname ilike '%v2%'  
ORDER BY l.total\_time DESC;  
  
  
CREATE OR REPLACE FUNCTION public.test\_function\_v3()  
RETURNS void AS $$  
BEGIN  
    FOR i IN 1..1000 LOOP  
        PERFORM pg\_sleep(0.001);  
    END LOOP;  
    perform test\_function\_v4();  
END;  
$$ LANGUAGE plpgsql;  
  
  
  
CREATE OR REPLACE FUNCTION public.test\_function\_v4()  
RETURNS integer AS $$  
DECLARE  
    counter integer := 0;  
BEGIN  
    FOR i IN 1..1000 LOOP  
        counter := counter + 1;  
        PERFORM pg\_sleep(0.001);  
    END LOOP;  
    RETURN counter;  
END;  
$$ LANGUAGE plpgsql;  
  
select \* from public.test\_function\_v3();  
  
**-- nested function calls if any**  
  
SELECT   
    c.stack[array\_upper(c.stack, 1)] as current\_function\_oid,  
    c.stack[array\_upper(c.stack, 1) - 1] as caller\_function\_oid,  
    p.proname as function\_name,  
    c.call\_count,  
    (c.us\_self/1000000.0)::numeric(10,2) as self\_time\_ms,  
    (c.us\_children/1000000.0)::numeric(10,2) as children\_time\_ms,  
    (c.us\_total/1000000.0)::numeric(10,2) as total\_time\_ms,  
    array\_length(c.stack, 1) as call\_depth  
FROM pl\_profiler\_callgraph\_local() c  
JOIN pg\_proc p ON c.stack[array\_upper(c.stack, 1)] = p.oid  
WHERE array\_length(c.stack, 1) > 0  
ORDER BY c.us\_total DESC;

## 4 Data Management Features: "Scaling for Growth"

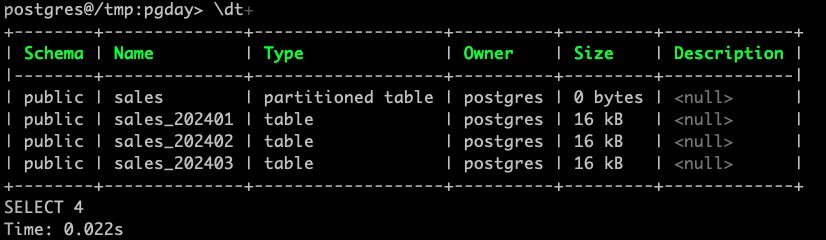
### 4.1 Partition Management

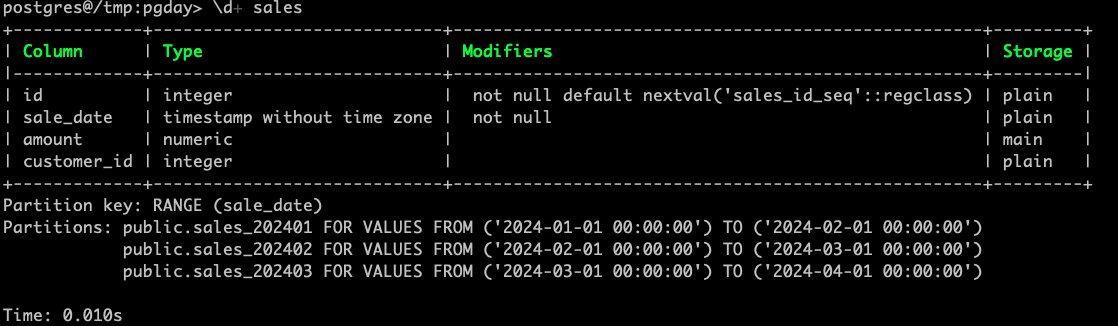
#### Understanding PostgreSQL Partitions

* Partitioning refers to splitting what is logically one large table into smaller physical pieces.
* Each partition stores a subset of the data as defined by its partition bounds.
* All rows inserted into a partitioned table will be routed to the appropriate one of the partitions based on the values of the partition key column(s). Updating the partition key of a row will cause it to be moved into a different partition if it no longer satisfies the partition bounds of its original partition. All partitions must have the same columns as their partitioned parent.
* The partitioned table itself is a “virtual” table having no storage of its own. Instead, the storage belongs to partitions, which are otherwise-ordinary tables associated with the partitioned table.
* **NOTE:**
  + It is not possible to turn a regular table into a partitioned table or vice versa.
  + Partitions (child tables) may have their own indexes, constraints and default values, distinct from those of other partitions.

-- Create parent table  
CREATE TABLE sales (  
    id serial,  
    sale\_date timestamp NOT NULL,  
    amount decimal,  
    customer\_id int  
) PARTITION BY RANGE (sale\_date);  
  
-- Manually create partitions for each month  
CREATE TABLE sales\_202401   
    PARTITION OF sales   
    FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');  
      
CREATE TABLE sales\_202402   
    PARTITION OF sales   
    FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');  
      
CREATE TABLE sales\_202403   
    PARTITION OF sales   
    FOR VALUES FROM ('2024-03-01') TO ('2024-04-01');  
  
-- Insert statements for above table  
INSERT INTO sales (sale\_date, amount, customer\_id)  
VALUES  
    ('2024-01-01', 100.00, 1),  
    ('2024-01-20', 150.00, 2),  
    ('2024-02-01', 200.00, 3),  
    ('2024-02-25', 120.00, 4),  
    ('2024-02-02', 180.00, 5),  
    ('2024-03-20', 90.00, 6),  
    ('2024-03-02', 110.00, 7),  
    ('2024-01-15', 130.00, 8),  
    ('2024-03-30', 140.00, 9),  
    ('2024-03-10', 160.00, 10),  
    ('2024-02-11', 111.00, 11),  
    ('2024-03-21', 111.00, 11),  
    ('2024-03-22', 111.00, 11),          
    ('2024-03-19', 222.00, 12);

* Verify the table structure.





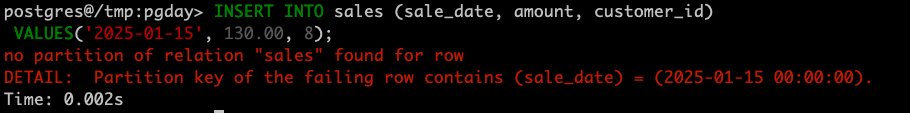
#### Problem 1: Problem of partition creation manually.

* PostgreSQL declarative partitioning does not create any child partitions automatically, including the default.
* User has to create past (for existing data) and future (for new data) partitions manually.

##### Default Partition

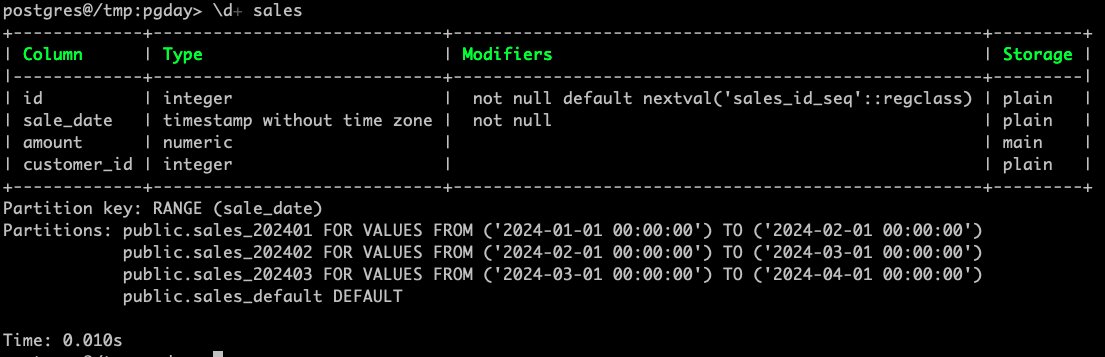
* If you insert data that does not match the existing partitions boundaries, then you get error.
* You get because each partition is associated with a lower and upper bounds which is the constraint for the partition and the constraint violates the data which is outside the range.

-- Insert statements for above table  
INSERT INTO sales (sale\_date, amount, customer\_id)  
VALUES('2025-01-15', 130.00, 8);



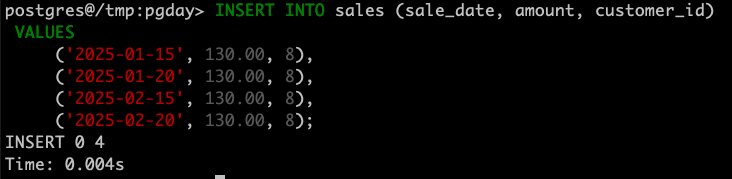
* Create default partition for data that doesn't match existing partitions boundaries.

-- Default partition creation  
CREATE TABLE sales\_default   
    PARTITION OF sales DEFAULT;



* Insert the data now and it should be successful.

-- Prepare insert statements for above table  
INSERT INTO sales (sale\_date, amount, customer\_id)  
VALUES  
    ('2025-01-15', 130.00, 8),  
    ('2025-01-20', 130.00, 8),  
    ('2025-02-15', 130.00, 8),  
    ('2025-02-20', 130.00, 8);



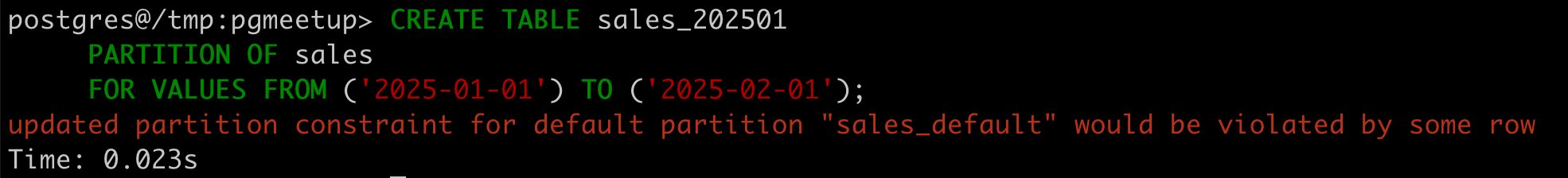
* Check the data in main table and also in child tables.

pgday=> select count(1) from sales;  
 count  
-------  
    18  
(1 row)  
  
pgday=> select count(1) from sales\_202401;  
 count  
-------  
     3  
(1 row)  
  
pgday=> select count(1) from sales\_202402;  
 count  
-------  
     4  
(1 row)  
  
pgday=> select count(1) from sales\_202403;  
 count  
-------  
     7  
(1 row)  
  
pgday=> select count(1) from sales\_default;  
 count  
-------  
     4  
(1 row)  
  
pgday=> select \* from sales\_202401;  
+----+---------------------+--------+-------------+  
| id | sale\_date           | amount | customer\_id |  
|----+---------------------+--------+-------------|  
| 1  | 2024-01-01 00:00:00 | 100.00 | 1           |  
| 2  | 2024-01-20 00:00:00 | 150.00 | 2           |  
| 8  | 2024-01-15 00:00:00 | 130.00 | 8           |  
+----+---------------------+--------+-------------+  
SELECT 3  
Time: 0.008s  
  
pgday=> select \* from sales\_default;  
+----+---------------------+--------+-------------+  
| id | sale\_date           | amount | customer\_id |  
|----+---------------------+--------+-------------|  
| 15 | 2025-01-15 00:00:00 | 130.00 | 8           |  
| 16 | 2025-01-20 00:00:00 | 130.00 | 8           |  
| 17 | 2025-02-15 00:00:00 | 130.00 | 8           |  
| 18 | 2025-02-20 00:00:00 | 130.00 | 8           |  
+----+---------------------+--------+-------------+  
SELECT 4  
Time: 0.007s

#### Problem 2: Problem of Default partition data movement.

* You can't create new partitions for data that is already in Default partition.

pgday=> CREATE TABLE sales\_202501  
    PARTITION OF sales  
    FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');



* You can always create new partitions which do not violate Default partition.

pgday=> CREATE TABLE sales\_202507  
    PARTITION OF sales  
    FOR VALUES FROM ('2025-07-01') TO ('2025-08-01');

**Manual solution to resolve data movement from default to actual partition?**

BEGIN;  
  
-- 1. Create a TEMP table  
CREATE TEMP TABLE **temp\_default**(LIKE **sales\_default**);  
  
-- 2. Load and Delete data from Default partition to TEMP table  
WITH **partition\_data** AS (  
    DELETE FROM **sales\_default** RETURNING \*  
)  
INSERT INTO **temp\_default** SELECT \* FROM **partition\_data**;  
  
/\*  
postgres@qbusinessdb:pgday> select \* from temp\_default;  
+----+---------------------+--------+-------------+  
| id | sale\_date           | amount | customer\_id |  
|----+---------------------+--------+-------------|  
| 15 | 2025-01-15 00:00:00 | 130.00 | 8           |  
| 16 | 2025-01-20 00:00:00 | 130.00 | 8           |  
| 17 | 2025-02-15 00:00:00 | 130.00 | 8           |  
| 18 | 2025-02-20 00:00:00 | 130.00 | 8           |  
+----+---------------------+--------+-------------+  
SELECT 4  
Time: 0.228s  
  
postgres@qbusinessdb:pgday> select \* from sales\_default;  
+----+-----------+--------+-------------+  
| id | sale\_date | amount | customer\_id |  
|----+-----------+--------+-------------|  
+----+-----------+--------+-------------+  
SELECT 0  
Time: 0.224s  
\*/  
  
-- 3. Create required partitons (child tables)  
CREATE TABLE sales\_202501   
    PARTITION OF sales   
    FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');  
  
CREATE TABLE sales\_202502   
    PARTITION OF sales   
    FOR VALUES FROM ('2025-02-01') TO ('2025-03-01');  
  
-- 4. Insert to child tables from TEMP table  
INSERT INTO sales\_202501  
SELECT \* FROM temp\_default WHERE sale\_date>= '2025-01-01'::DATE AND sale\_date< '2025-02-01'::DATE;  
  
INSERT INTO sales\_202502  
SELECT \* FROM temp\_default WHERE sale\_date>= '2025-02-01'::DATE AND sale\_date< '2025-03-01'::DATE;  
  
-- 5. Drop TEMP table and Commit  
DROP TABLE temp\_default;  
COMMIT;  
  
/\*  
postgres@qbusinessdb:pgday> select \* from sales\_202501;  
+----+---------------------+--------+-------------+  
| id | sale\_date           | amount | customer\_id |  
|----+---------------------+--------+-------------|  
| 15 | 2025-01-15 00:00:00 | 130.00 | 8           |  
| 16 | 2025-01-20 00:00:00 | 130.00 | 8           |  
+----+---------------------+--------+-------------+  
SELECT 2  
Time: 0.225s  
  
postgres@qbusinessdb:pgday> select \* from sales\_202502;  
+----+---------------------+--------+-------------+  
| id | sale\_date           | amount | customer\_id |  
|----+---------------------+--------+-------------|  
| 17 | 2025-02-15 00:00:00 | 130.00 | 8           |  
| 18 | 2025-02-20 00:00:00 | 130.00 | 8           |  
+----+---------------------+--------+-------------+  
SELECT 2  
Time: 0.224s  
  
postgres@qbusinessdb:pgday> select \* from sales\_default;  
+----+-----------+--------+-------------+  
| id | sale\_date | amount | customer\_id |  
|----+-----------+--------+-------------|  
+----+-----------+--------+-------------+  
SELECT 0  
Time: 0.228s  
\*/

#### Problem 3: Problem of partition maintenance.

Maintenance of partitions like adding, detaching old partions, archiving data is another challenge when working with partition tables.

So, to summarize the problems with manual partition table management:

* Manual partition creation for every month!
* Time-consuming repetitive work
* Prone to human errors in date ranges
* Need to remember to create future partitions
* Large amounts of data in the default partition
* Maintenance Overhead
* Future Partition Planning
* Old Partition Cleanup

PostgreSQL has several extension which can help overcome above issues.

### 4.2 pg\_partman

**pg\_partman** is an extension for PostgreSQL that helps automate partition creation and maintenance. It significantly simplifies partition management, which can be complex and time-consuming when done manually.

-- Verify installed extensions  
pgday=> \dx

* Drop the table and initialize the pg\_partman for child table creation.

-- DROP table  
DROP TABLE sales;  
  
-- Create the Parent table  
CREATE TABLE sales (  
    id serial,  
    sale\_date timestamp NOT NULL,  
    amount decimal,  
    customer\_id int  
) PARTITION BY RANGE (sale\_date);  
  
/\*  
**-- Does not create the child tables**  
postgres@qbusinessdb:pgday> \d sales  
+-------------+-----------------------------+-----------------------------------------------------+  
| Column      | Type                        | Modifiers                                           |  
|-------------+-----------------------------+-----------------------------------------------------|  
| id          | integer                     |  not null default nextval('sales\_id\_seq'::regclass) |  
| sale\_date   | timestamp without time zone |  not null                                           |  
| amount      | numeric                     |                                                     |  
| customer\_id | integer                     |                                                     |  
+-------------+-----------------------------+-----------------------------------------------------+  
Partition key: RANGE (sale\_date)  
  
Time: 1.151s (1 second), executed in: 1.138s (1 second)  
\*/  
  
-- Initialize pg\_partman for monthly partitioning  
SELECT partman.create\_parent(  
    p\_parent\_table := 'public.sales',  
    p\_control := 'sale\_date',  
    p\_type := 'range',  
    p\_interval := '1 month',  
    p\_start\_partition := '2024-01-01',  
    p\_premake := 12  
);



* Creates 12 future partitions as well as the **Default** partition. (**Problem 1 resolved**)
* Insert data now.

-- Prepare insert statements for above table  
INSERT INTO sales (sale\_date, amount, customer\_id)  
VALUES  
    ('2024-01-01', 100.00, 1),  
    ('2024-01-20', 150.00, 2),  
    ('2024-02-01', 200.00, 3),  
    ('2024-02-25', 120.00, 4),  
    ('2024-02-02', 180.00, 5),  
    ('2024-03-20', 90.00, 6),  
    ('2024-03-02', 110.00, 7),  
    ('2024-01-15', 130.00, 8),  
    ('2024-03-30', 140.00, 9),  
    ('2024-03-10', 160.00, 10),  
    ('2024-02-11', 111.00, 11),  
    ('2024-03-21', 111.00, 11),  
    ('2024-03-22', 111.00, 11),          
    ('2024-03-19', 222.00, 12),  
    ('2025-01-15', 130.00, 8),  
    ('2025-01-20', 130.00, 8),  
    ('2025-02-15', 130.00, 8),  
    ('2025-02-20', 130.00, 8);  
  
-- Creating older data for Default partition  
INSERT INTO sales (sale\_date, amount, customer\_id)  
VALUES  
    ('2023-01-10', 130.00, 11),  
    ('2023-01-15', 140.00, 22),  
    ('2023-01-20', 150.00, 33),  
    ('2023-02-20', 160.00, 44);  
  
/\*  
pgday> select \* from sales\_default;  
+----+---------------------+--------+-------------+  
| id | sale\_date           | amount | customer\_id |  
|----+---------------------+--------+-------------|  
| 19 | 2023-01-10 00:00:00 | 130.00 | 11          |  
| 20 | 2023-01-15 00:00:00 | 140.00 | 22          |  
| 21 | 2023-01-20 00:00:00 | 150.00 | 33          |  
| 22 | 2023-02-20 00:00:00 | 160.00 | 44          |  
+----+---------------------+--------+-------------+  
SELECT 4  
Time: 0.234s  
\*/

* The pg\_partman procedure partition\_data\_proc() will automatically clean up your default partition. This procedure does the same steps shown above for native partitioning: moving the data to a temporary table, creating the necessary child tables based on the data found, then moving the data back.

postgres@qbusinessdb:pgday> CALL partman.partition\_data\_proc('public.sales');  
  
Loop: 1, Rows moved: 3  
Loop: 2, Rows moved: 1  
Total rows moved: 4  
Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning data  
CALL  
Time: 2.296s (2 seconds), executed in: 2.296s (2 seconds)  
  
**-- Now no data in Default partition**  
postgres@qbusinessdb:pgday> select \* from sales\_default;  
+----+-----------+--------+-------------+  
| id | sale\_date | amount | customer\_id |  
|----+-----------+--------+-------------|  
+----+-----------+--------+-------------+  
SELECT 0  
Time: 0.234s

* Verify table data and table structure to check new partitions got added and data moved from Default partition. (**Problem 2 resolved**)

postgres@qbusinessdb:pgday> \d+ sales  
+-------------+-----------------------------+-----------------------------------------------------+---------+  
| Column      | Type                        | Modifiers                                           | Storage |  
|-------------+-----------------------------+-----------------------------------------------------+---------|  
| id          | integer                     |  not null default nextval('sales\_id\_seq'::regclass) | plain   |  
| sale\_date   | timestamp without time zone |  not null                                           | plain   |  
| amount      | numeric                     |                                                     | main    |  
| customer\_id | integer                     |                                                     | plain   |  
+-------------+-----------------------------+-----------------------------------------------------+---------+  
Partition key: RANGE (sale\_date)  
Partitions: public.sales\_default DEFAULT  
            public.sales\_p20230101 FOR VALUES FROM ('2023-01-01 00:00:00') TO ('2023-02-01 00:00:00')  
            public.sales\_p20230201 FOR VALUES FROM ('2023-02-01 00:00:00') TO ('2023-03-01 00:00:00')  
            public.sales\_p20240101 FOR VALUES FROM ('2024-01-01 00:00:00') TO ('2024-02-01 00:00:00')  
            public.sales\_p20240201 FOR VALUES FROM ('2024-02-01 00:00:00') TO ('2024-03-01 00:00:00')  
            public.sales\_p20240301 FOR VALUES FROM ('2024-03-01 00:00:00') TO ('2024-04-01 00:00:00')  
            public.sales\_p20240401 FOR VALUES FROM ('2024-04-01 00:00:00') TO ('2024-05-01 00:00:00')  
            public.sales\_p20240501 FOR VALUES FROM ('2024-05-01 00:00:00') TO ('2024-06-01 00:00:00')  
            public.sales\_p20240601 FOR VALUES FROM ('2024-06-01 00:00:00') TO ('2024-07-01 00:00:00')  
            public.sales\_p20240701 FOR VALUES FROM ('2024-07-01 00:00:00') TO ('2024-08-01 00:00:00')  
            public.sales\_p20240801 FOR VALUES FROM ('2024-08-01 00:00:00') TO ('2024-09-01 00:00:00')  
            public.sales\_p20240901 FOR VALUES FROM ('2024-09-01 00:00:00') TO ('2024-10-01 00:00:00')  
            public.sales\_p20241001 FOR VALUES FROM ('2024-10-01 00:00:00') TO ('2024-11-01 00:00:00')  
            public.sales\_p20241101 FOR VALUES FROM ('2024-11-01 00:00:00') TO ('2024-12-01 00:00:00')  
            public.sales\_p20241201 FOR VALUES FROM ('2024-12-01 00:00:00') TO ('2025-01-01 00:00:00')  
            public.sales\_p20250101 FOR VALUES FROM ('2025-01-01 00:00:00') TO ('2025-02-01 00:00:00')  
            public.sales\_p20250201 FOR VALUES FROM ('2025-02-01 00:00:00') TO ('2025-03-01 00:00:00')  
            public.sales\_p20250301 FOR VALUES FROM ('2025-03-01 00:00:00') TO ('2025-04-01 00:00:00')  
            public.sales\_p20250401 FOR VALUES FROM ('2025-04-01 00:00:00') TO ('2025-05-01 00:00:00')  
            public.sales\_p20250501 FOR VALUES FROM ('2025-05-01 00:00:00') TO ('2025-06-01 00:00:00')  
            public.sales\_p20250601 FOR VALUES FROM ('2025-06-01 00:00:00') TO ('2025-07-01 00:00:00')  
            public.sales\_p20250701 FOR VALUES FROM ('2025-07-01 00:00:00') TO ('2025-08-01 00:00:00')  
            public.sales\_p20250801 FOR VALUES FROM ('2025-08-01 00:00:00') TO ('2025-09-01 00:00:00')  
            public.sales\_p20250901 FOR VALUES FROM ('2025-09-01 00:00:00') TO ('2025-10-01 00:00:00')  
            public.sales\_p20251001 FOR VALUES FROM ('2025-10-01 00:00:00') TO ('2025-11-01 00:00:00')  
            public.sales\_p20251101 FOR VALUES FROM ('2025-11-01 00:00:00') TO ('2025-12-01 00:00:00')  
            public.sales\_p20251201 FOR VALUES FROM ('2025-12-01 00:00:00') TO ('2026-01-01 00:00:00')  
            public.sales\_p20260101 FOR VALUES FROM ('2026-01-01 00:00:00') TO ('2026-02-01 00:00:00')  
            public.sales\_p20260201 FOR VALUES FROM ('2026-02-01 00:00:00') TO ('2026-03-01 00:00:00')  
            public.sales\_p20260301 FOR VALUES FROM ('2026-03-01 00:00:00') TO ('2026-04-01 00:00:00')  
            public.sales\_p20260401 FOR VALUES FROM ('2026-04-01 00:00:00') TO ('2026-05-01 00:00:00')  
            public.sales\_p20260501 FOR VALUES FROM ('2026-05-01 00:00:00') TO ('2026-06-01 00:00:00')  
            public.sales\_p20260601 FOR VALUES FROM ('2026-06-01 00:00:00') TO ('2026-07-01 00:00:00')  
            public.sales\_p20260701 FOR VALUES FROM ('2026-07-01 00:00:00') TO ('2026-08-01 00:00:00')  
            public.sales\_p20260801 FOR VALUES FROM ('2026-08-01 00:00:00') TO ('2026-09-01 00:00:00')  
  
Time: 1.283s (1 second), executed in: 1.274s (1 second)

**Gaps in child partitions:**

* Now we can see that the new child partitions have been made, the data has been moved to them, and the default partition is empty.
* pg\_partman will only make new partitions based on the NEWEST partition and available data. It will not automatically fill in gaps to avoid potentially expensive automatic maintenance operations. However in many cases, you will be expecting data for these missing child partitions and will want to fill in the gaps.
* pg\_partman has a utility that you can run manually to do this: partition\_gap\_fill.

pgday> SELECT \* FROM partman.partition\_gap\_fill('public.sales');  
+--------------------+  
| partition\_gap\_fill |  
|--------------------|  
| 10 |  
+--------------------+  
SELECT 1  
Time: 0.316s

**Let’s resolve Problem 3:**

* Automatic periodic maintenance.3.3

SELECT \* FROM partman.part\_config;  
  
UPDATE partman.part\_config SET retention = '12 months';  
  
-- Update partition configuration for specific needs  
UPDATE partman.part\_config   
SET   
    retention = '6 months',        -- Keep 36 months of data  
    retention\_schema = 'backup',    -- Store Backup data  
    retention\_keep\_table = true,    -- Keep the table after detaching  
    automatic\_maintenance = 'on',   -- Enable automatic maintenance  
    premake = 3,                    -- Create 3 future partitions  
    infinite\_time\_partitions = true -- Allow infinite future partitions  
WHERE parent\_table = 'public.sales';  
  
CREATE SCHEMA backup;  
UPDATE partman.part\_config SET retention = '3 months', retention\_schema = 'backup';  
  
-- Run the maintenance job  
SELECT partman.run\_maintenance('public.sales');  
  
-- Partition Management and Monitoring:  
-- View current partitions  
SELECT \* FROM partman.show\_partitions('public.sales');  
  
-- Check partition information  
SELECT \* FROM partman.part\_config   
WHERE parent\_table = 'public.sales';  
  
-- View partition sizes  
SELECT   
    partition\_tablename,  
    pg\_size\_pretty(pg\_total\_relation\_size(partition\_schemaname || '.' || partition\_tablename)) as size  
FROM partman.show\_partitions('public.sales');  
  
-- Check row counts in partitions  
SELECT   
    schemaname || '.' || tablename as partition\_name,  
    n\_live\_tup as estimated\_rows  
FROM pg\_stat\_user\_tables  
WHERE tablename LIKE 'sales\_p%'  
ORDER BY tablename;  
  
-- Add indexes to parent table (automatically inherited by partitions)  
CREATE INDEX idx\_sales\_sale\_date ON sales(sale\_date);  
CREATE INDEX idx\_sales\_customer ON sales(customer\_id);  
  
-- Optimize specific partition if needed  
ALTER TABLE sales\_p2024\_01 SET (  
    autovacuum\_vacuum\_scale\_factor = 0.05,  
    autovacuum\_analyze\_scale\_factor = 0.02  
);

### 4.3 pg\_cron

**pg\_cron** is a PostgreSQL extension that provides a cron-based job scheduler directly within the database. It allows users to schedule and execute SQL commands or calls to stored procedures using the familiar cron syntax.

-- Create a maintenance schedule using pg\_cron

CREATE EXTENSION pg\_cron;

--Cron schedule

┌───────────── min (0 - 59)

│ ┌────────────── hour (0 - 23)

│ │ ┌─────────────── day of month (1 - 31) or last day of the month ($)

│ │ │ ┌──────────────── month (1 - 12)

│ │ │ │ ┌───────────────── day of week (0 - 6) (0 to 6 are Sunday to

│ │ │ │ │ Saturday, or use names; 7 is also Sunday)

│ │ │ │ │

│ │ │ │ │

\* \* \* \* \*

Let’s take a scenario to drop local users and schedule to run the function every 5 mins.

**--Dropping local user through PG function**

CREATE OR REPLACE FUNCTION fnc\_drop\_local\_user()

RETURNS void AS $$

DECLARE

rol RECORD;

BEGIN

FOR rol IN SELECT rolname FROM pg\_roles WHERE rolcanlogin AND NOT rolsuper AND rolname NOT IN ('postgres') --Gets all roles which can login

LOOP

BEGIN

EXECUTE 'DROP ROLE '||rol.rolname;

EXCEPTION

WHEN OTHERS THEN

IF SQLERRM LIKE '%cannot be dropped because some objects depend on it' THEN

--EXECUTE 'REASSIGN OWNED BY '||rol.rolname||' TO postgres';

BEGIN

EXECUTE 'GRANT '||rol.rolname||' to postgres';

EXECUTE 'DROP OWNED BY '||rol.rolname;

EXECUTE 'DROP ROLE '||rol.rolname;

EXCEPTION

WHEN OTHERS THEN

RAISE DEBUG 'Error: %', SQLERRM;

END;

ELSE

RAISE DEBUG 'Error: %', SQLERRM;

END IF;

END;

END LOOP;

END;

$$ LANGUAGE plpgsql;

**--Test above function**

postgres=# select fnc\_drop\_local\_user();

fnc\_drop\_local\_user

---------------------

(1 row)

**--At every minute**

SELECT cron.schedule('daily\_delete\_user', '\* \* \* \* \*', 'SELECT fnc\_drop\_local\_user()');

**--Check the jobs scheduled**

select \* from cron.job;

**--Check running jobs**

select \* from cron.job\_run\_details;

**-- unschedule**

SELECT cron.unschedule ('daily\_delete\_user');

-- Schedule daily maintenance at 2 AM

SELECT cron.schedule(

'sales\_partition\_maintenance',

'0 2 \* \* \*',

$$SELECT partman.run\_maintenance(

p\_analyze := true,

p\_jobmon := true

)$$

);

-- Alternative: Manual maintenance run

SELECT partman.run\_maintenance();

--Delete all data job\_run\_details

CREATE OR REPLACE FUNCTION fnc\_delete\_job\_run\_details()

RETURNS void AS $$

BEGIN

DELETE FROM cron.job\_run\_details;

END;

$$ LANGUAGE plpgsql;

--Weekly once at Sunday 1:00 AM

SELECT cron.schedule('weekly\_flush\_job\_run\_details', '00 1 \* \* 7', 'SELECT fnc\_delete\_job\_run\_details()');

SELECT cron.schedule('partman\_maintenance\_job', '\* \* \* \* \*', 'call partman.run\_maintenance\_proc()');

SELECT cron.unschedule ('partman\_maintenance\_job');

Benefits of combining pg\_partman and pg\_cron:

* **Full Automation:** Eliminates the need for manual intervention in partition management.
* **Reduced Administrative Burden:** Frees up database administrators from routine partitioning tasks.
* **Improved Performance:** Ensures that partitions are managed optimally, leading to better query performance and reduced resource consumption.
* **Data Retention and Archiving:** Automates the process of keeping only necessary data online and potentially archiving older data.

### 4.4 pg\_repack

**pg\_repack** is an open-source PostgreSQL extension developed and maintained by the community. It is an **online table reorganization tool** that removes table and index bloat **without requiring downtime** or exclusive locks on tables.

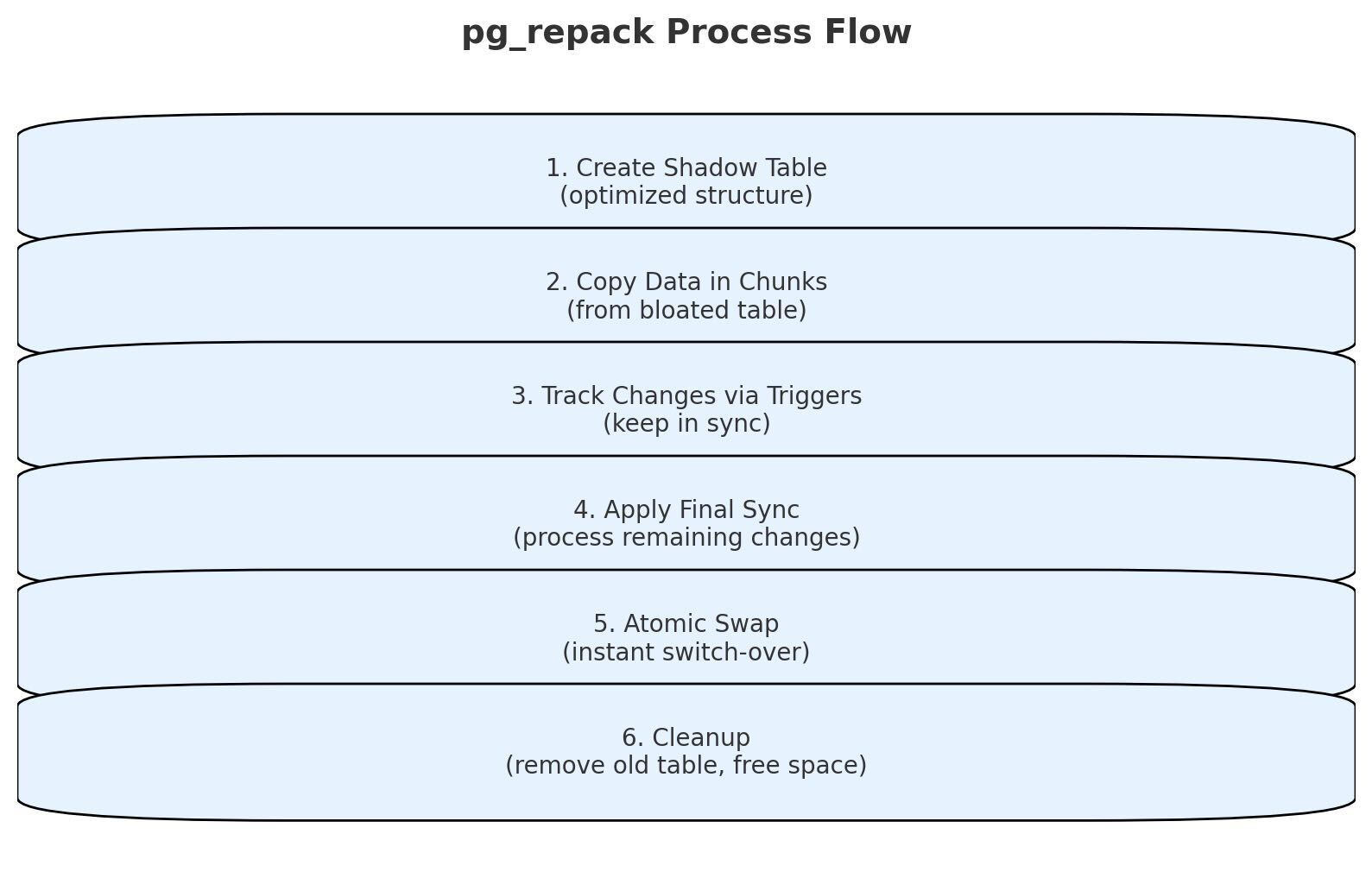
#### Why Do You Need pg\_repack?

* Reclaim disk space after deleting large amounts of data
* Reorder and compact table rows to reduce page usage
* Recover from bloat due to ineffective autovacuum settings
* Improve overall database performance and storage efficiency

#### Key Benefits

* Zero downtime – Tables stay online during processing
* Space reclamation – Frees actual disk space by shrinking table files
* Performance boost – Optimizes data layout and rebuilds indexes
* Online operation – Uses triggers to capture live changes during processing

#### How It Works



#### Feature Comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Feature | Autovacuum | VACUUM FULL | pg\_repack |
| 1 | **Downtime** | None | ❌ Full downtime | ✅ None |
| 2 | **Space reclamation** | ❌ No | ✅ Yes | ✅ Yes |
| 3 | **Locking** | Minimal | ❌ Exclusive | ✅ Minimal |
| 4 | **Performance impact** | Low | ❌ High | ✅ Low |
| 5 | **Index rebuilding** | ❌ No | ✅ Yes | ✅ Yes |
| 6 | **Disk space needed** | None | 2x table size | 2x table size |

#### Demo

-- Create a demo table  
CREATE TABLE demo\_bloat\_test (  
 id SERIAL PRIMARY KEY,  
 data TEXT,  
 status INTEGER DEFAULT 1,  
 created\_at TIMESTAMP DEFAULT NOW()  
);

-- Insert 1 million rows  
INSERT INTO demo\_bloat\_test (data, status)SELECT  
'Sample data row ' || generate\_series,  
 (random() \* 10)::INTEGER  
FROM generate\_series(1, 1000000);

#### Check Initial Table State

SELECT  
'INITIAL STATE' AS phase,  
 n\_live\_tup AS live\_tuples,  
 n\_dead\_tup AS dead\_tuples,  
 ROUND(100.0 \* n\_dead\_tup / GREATEST(n\_live\_tup + n\_dead\_tup, 1), 2) AS dead\_tuple\_pct,  
 ROUND(pg\_total\_relation\_size('demo\_bloat\_test')/1024/1024.0, 2) AS size\_mb,  
 pg\_size\_pretty(pg\_total\_relation\_size('demo\_bloat\_test')) AS human\_sizeFROM pg\_stat\_user\_tables WHERE relname = 'demo\_bloat\_test';

#### Simulate Bloat

-- Update 25% of rows to simulate dead tuples  
UPDATE demo\_bloat\_testSET data = 'Updated batch 1 - ' || data,   
 status = status + 1  
WHERE id % 4 = 0;

#### Option 1: VACUUM or Autovacuum

VACUUM demo\_bloat\_test;

#### Option 2: VACUUM FULL (with downtime)

VACUUM FULL demo\_bloat\_test;

#### Option 3: pg\_repack (online)

pg\_repack -h localhost -U postgres -d pgday -p 5433 -t demo\_bloat\_test

#### Check Table After Repacking

SELECT  
'AFTER UPDATE' AS phase,  
 n\_live\_tup AS live\_tuples,  
 n\_dead\_tup AS dead\_tuples,  
 ROUND(100.0 \* n\_dead\_tup / GREATEST(n\_live\_tup + n\_dead\_tup, 1), 2) AS dead\_tuple\_pct,  
 ROUND(pg\_total\_relation\_size('demo\_bloat\_test')/1024/1024.0, 2) AS size\_mb,  
 pg\_size\_pretty(pg\_total\_relation\_size('demo\_bloat\_test')) AS human\_sizeFROM pg\_stat\_user\_tables WHERE relname = 'demo\_bloat\_test';

#### When to Use Each Tool

* **Autovacuum**  
  Routine daily maintenance. Keeps performance steady and prevents bloat.
* **pg\_repack**  
  Periodic deep cleanup (monthly/quarterly) to reclaim space and optimize layout.
* **VACUUM FULL**  
  Use only during maintenance windows. High impact and requires downtime.

#### Real-World Use Case

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Scenario | Autovacuum | VACUUM FULL | pg\_repack |
| 1 | Table Size | 1TB | 1TB | 1TB |
| 2 | Bloat | 50% | 50% | 50% |
| 3 | Space Reclaimed | ❌ 0 GB | ✅ 500 GB | ✅ 500 GB |
| 4 | Downtime | None | ❌ 2–8 hours | ✅ None |
| 5 | Disk Space Requirement | None | ❌ 2TB required | ✅ 1TB required |
| 6 | Time Taken | Ongoing | ❌ Slow | ✅ ~30 minutes |

**Recommendation:**  
Use **pg\_repack** in **production systems** where **downtime is not acceptable**.

### 4.5 Foreign Data Wrappers (FDWs)

Foreign Data Wrapper (FDW) extensions in PostgreSQL serve as the foundation for connecting to external data sources. These extensions provide the necessary interface and protocols to communicate with different types of external systems. When creating an extension, it's essential to ensure that all required dependencies and client libraries are properly installed on the PostgreSQL server. The extension creation process registers the FDW functionality within PostgreSQL and makes its features available for use. For example, 'postgres\_fdw' enables connectivity to other PostgreSQL databases.

* file\_fdw
* postgres\_fdw
* oracle\_fdw
* mysql\_fdw
* tds\_fdw
* mongodb\_fdw

#### FDW Components

##### Foreign Server

A foreign server represents the connection configuration to an external data source. It acts as a container for connection parameters such as host address, port number, database name, and various performance-related settings. The foreign server definition is crucial as it establishes how PostgreSQL will communicate with the remote system. Multiple foreign tables can share the same server configuration, making it a reusable component in your FDW setup. Server options can be customized to optimize performance, handle timeouts, and manage connection behaviors. The server configuration serves as a bridge between your local PostgreSQL instance and the remote data source.

##### User Mapping

User mapping establishes the authentication and credentials needed to connect to the remote system. It links a local PostgreSQL user to the corresponding authentication credentials for the remote server. This mapping is essential for security as it allows different local users to have different remote access credentials. User mappings can be created for specific users or for all users (PUBLIC), providing flexibility in managing access control. The credentials stored in user mappings are encrypted and managed securely by PostgreSQL. This component ensures that proper authentication is maintained while accessing remote data sources.

##### Foreign Tables

Foreign tables are virtual tables in PostgreSQL that map to actual tables or views in the remote data source. They define the structure and characteristics of the remote data, including column names, data types, and any specific options for data access. Foreign tables can be created manually by specifying the column definitions or imported automatically using IMPORT FOREIGN SCHEMA. They behave similarly to regular PostgreSQL tables but query data from the remote source when accessed. Foreign tables can include options for performance optimization, column mapping, and other behavioral characteristics.

##### FDW OPTIONS Parameters: Server Level OPTIONS

##### fetch\_size

OPTIONS (fetch\_size '1000')

**Purpose**: Controls the number of rows fetched in each batch from the remote server.

* **Default**: 100 rows
* **Impact**: Larger values reduce network round trips but increase memory usage
* **Use Case**:
  + Increase for better performance with large result sets
  + Decrease for memory-constrained environments

**-- Example for large datasets**  
OPTIONS (fetch\_size '5000')

**-- Example for memory-constrained systems**  
OPTIONS (fetch\_size '200')

**use\_remote\_estimate**

   OPTIONS (use\_remote\_estimate 'true')

**Purpose**: Determines whether to use the remote server's cost estimates for query planning.

* **Default**: false
* **Impact**: Affects query optimization and execution plan
* **Use Case**:
  + Enable for better query planning with accurate remote statistics
  + Disable if remote statistics are unreliable

**-- Enable remote cost estimation**  
OPTIONS (use\_remote\_estimate 'true')  
   
**-- Disable remote cost estimation**  
OPTIONS (use\_remote\_estimate 'false')

##### Other Common Server OPTIONS

**Connection Management**

    OPTIONS (  
    connect\_timeout '30',       **-- Connection timeout in seconds**  
    keepalives '1',             **-- Enable TCP keepalives**  
    keepalives\_idle '60',       **-- Idle time before keepalive**  
    keepalives\_interval '10',   **-- Interval between keepalives**  
    keepalives\_count '3'        **-- Number of retries**  
)

**Performance Tuning**

    OPTIONS (  
    batch\_size '1000',          **-- Batch size for modifications**  
    updatable 'true',           **-- Allow updates on foreign tables**  
    truncatable 'true',         **-- Allow TRUNCATE command**  
)

##### Table Level OPTIONS

**1. Basic Table Options**

    CREATE FOREIGN TABLE foreign\_table (  
    id int,  
    name text  
)  
SERVER foreign\_server  
OPTIONS (  
    schema\_name 'public',       **-- Remote schema name**  
    table\_name 'remote\_table',  **-- Remote table name**  
    batch\_size '1000'          **-- Batch size for this table**  
);

**Column Level Options**

CREATE FOREIGN TABLE foreign\_table (  
    id   int OPTIONS (key 'true'),             **-- Mark as key column**  
    name text OPTIONS (not null 'true'),     **-- Mark as not null**  
    data text OPTIONS (column\_name 'data\_remote') **-- Map to different column name**  
)  
SERVER foreign\_server;

#### postgres\_fdw

**--create databses and users as below**  
  
CREATE DATABASE sourcedb;  
CREATE DATABASE targetdb;  
  
CREATE USER sourceuser WITH password 'spassword';  
CREATE USER targetuser WITH password 'tpassword';  
  
  
**--Connect to targetdb and then create the schema and table**  
CREATE SCHEMA targetschema AUTHORIZATION targetuser;  
  
CREATE TABLE targetschema.targetsales (  
    id SERIAL PRIMARY KEY,  
    customer\_id INTEGER,  
    product\_id INTEGER,  
    sale\_date DATE,  
    amount DECIMAL(10,2),  
    region VARCHAR(50),  
    payment\_mode VARCHAR(20),  
    status VARCHAR(20)  
);  
  
alter table targetschema.targetsales owner to targetuser;  
  
CREATE OR REPLACE FUNCTION generate\_sales\_data(n INTEGER) RETURNS void AS $$  
BEGIN  
    FOR i IN 1..n LOOP  
        INSERT INTO targetschema.targetsales (  
            customer\_id,  
            product\_id,  
            sale\_date,  
            amount,  
            region,  
            payment\_mode,  
            status  
        )  
        SELECT  
            floor(random() \* 1000 + 1)::int, -- customer\_id between 1-1000  
            floor(random() \* 100 + 1)::int,  -- product\_id between 1-100  
            current\_date - (random() \* 365)::int, -- random date within last year  
            (random() \* 1000)::decimal(10,2), -- amount between 0-1000  
            (ARRAY['North', 'South', 'East', 'West', 'Central'])[floor(random() \* 5 + 1)], -- random region  
            (ARRAY['Cash', 'Credit Card', 'UPI', 'Net Banking'])[floor(random() \* 4 + 1)], -- random payment mode  
            (ARRAY['Completed', 'Pending', 'Failed', 'Refunded'])[floor(random() \* 4 + 1)] -- random status  
        ;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;  
  
  
**-- Insert 1 million records**  
SELECT generate\_sales\_data(1000000);  
  
select count(\*) from targetschema.targetsales;  
  
create table targetschema.targetsales1 as select \* from targetschema.targetsales;  
  
**-- come back to sourcedb and create extension**  
CREATE SCHEMA sourceschema AUTHORIZATION sourceuser;  
CREATE EXTENSION postgres\_fdw;  
   
**-- Create server connection**  
CREATE SERVER <postgres\_remote>  
    FOREIGN DATA WRAPPER postgres\_fdw  
    OPTIONS (host '<remote\_host>', port '5432', dbname '<remote\_db>');  
   
**-- Create user mapping**  
CREATE USER MAPPING FOR <local\_user>  
    SERVER <postgres\_remote>  
    OPTIONS (user '<remote\_user>', password '<remote\_password>');  
   
**-- Create foreign table**  
CREATE FOREIGN TABLE sourceschema.targetsales (  
    id SERIAL ,  
    customer\_id INTEGER,  
    product\_id INTEGER,  
    sale\_date DATE,  
    amount DECIMAL(10,2),  
    region VARCHAR(50),  
    payment\_mode VARCHAR(20),  
    status VARCHAR(20)  
)  
    SERVER sourcetotarget  
    OPTIONS (schema\_name 'targetschema', table\_name 'targetsales');  
   
 GRANT USAGE ON FOREIGN SERVER <remote\_user> TO sourceuser;  
   
**-- Import single table**  
IMPORT FOREIGN SCHEMA <remote\_schema>  
    LIMIT TO (targetsales1)  
    FROM SERVER <postgres\_remote>  
    INTO <source schema>;  
   
**-- Import all tables from schema**  
IMPORT FOREIGN SCHEMA <remote\_schema>  
    FROM SERVER <postgres\_remote>  
    INTO <source schema>;

#### file\_fdw

7. file\_fdw (PostgreSQL to CSV Files)  
-- Load extension  
CREATE EXTENSION file\_fdw;  
  
-- Create server connection  
CREATE SERVER csv\_server  
    FOREIGN DATA WRAPPER file\_fdw;  
  
-- Create foreign table  
CREATE FOREIGN TABLE csv\_data (  
    id integer,  
    name text,  
    email text  
)  
    SERVER csv\_server  
    OPTIONS (  
        filename '/path/to/file.csv',  
        format 'csv',  
        header 'true'  
    );

PostgreSQL can link to other systems to fetch data via foreign data wrappers (FDWs) . When we fire a query (e.g., SELECT) against a foreign table, the FDW will fetch the result from the external data source and print the output via foreign table.

Below are some popular FDWs:  
a) postgres\_fdw is used to perform DML operations (select, insert, update, delete) with external PostgreSQL servers in PostgreSQL. postgres\_fdw is a contrib module and can be found in the contrib/ folder of PostgreSQL sources. For more information please visit  https://www.postgresql.org/docs/11/postgres-fdw.html.  
b) mysql\_fdw is used to perform DML operations (select, insert, update, delete) with MySQL servers in PostgreSQL. We need to download the source code from GitHub and compile against the PostgreSQL server. For more information please visit https://github.com/EnterpriseDB/mysql\_fdw.  
c) oracle\_fdw is used to perform DML operations (select, insert, update, delete) with Oracle servers in PostgreSQL. For more information please visit https://github.com/laurenz/oracle\_fdw.  
d) mongo\_fdw is is used to perform DML operations (select, insert, update, delete) with MongoDB in PostgreSQL. For more information please visit https://github.com/EnterpriseDB/mongo\_fdw.  
e) hdfs\_fdw is used for the Hadoop distributed file system. It can work with Hive or Spark. It provides the capability to read data from the Hadoop Distributed File System (HDFS) using SQL and use it for online transactional processing (OLTP) and analytical purposes. For more information please visit <https://github.com/EnterpriseDB/hdfs_fdw>.

## 5 Development & Quality Tools: "Ensuring Code Quality"

### 5.1 Orafce

* Stands for **ORA**cle **F**unction **C**ompatibility **E**xtension.Provides Oracle-compatible functions and packages in PostgreSQL.
* This extension allows us to write Oracle-style procedure/function in PostgreSQL.
* The orafce extension provides functions and operators that emulate a subset of functions and packages from an Oracle database. The orafce extension makes it easier for you to port an Oracle application to PostgreSQL. For more information about orafce, see [orafce](https://github.com/orafce/orafce) on GitHub.

#### Use Cases

* Database migration projects from Oracle to PostgreSQL
* Applications requiring Oracle compatibility
* Legacy system modernization
* Cross-platform database deployments
* Maintaining application compatibility during database transitions
* Reducing migration costs and efforts

#### What you get in Orafce extensions?

* DUAL Table
* Data types: VARCHAR2, NVARCHAR2, DATE, NUMBER
* Functions:
  + Date/Time functions: SYSDATE, ADD\_MONTHS, TO\_DATE, TO\_NUMBER, LAST\_DAY, NEXT\_DAY, TRUNC, ROUND
  + Character functions: concat, nvl, nvl2, lnnvl, decode, substr
* Oracle system supplied Packages: dbms\_output, utl\_file, dbms\_utility, dbms\_random
* Operators: || (concatenation), INSTR, SUBSTR
* Full list can be found at <https://github.com/orafce/orafce>

#### Benefits

* Helps in Oracle to PostgreSQL migration scenarios.
* Enables easier transition for applications using Oracle-specific functions.
* Reduces code modifications when migrating from Oracle to PostgreSQL.
* Actively maintained to support newer PostgreSQL releases.
* Regular updates to ensure compatibility with latest PostgreSQL versions.

#### Hands-on Scenarios

##### TRUNC:

* TRUNC function in Oracle can run for VARCHAR, NUMBER or DATE data types.
* PostgreSQL has DATE\_TRUNC similar to Oracle TRUNC for DATE types.

-Oracle  
SELECT TRUNC(SYSDATE, 'MM') FROM DUAL;  
 trunc  
---------------------  
 01-JUN-2025  
(1 row)  
  
SELECT TRUNC(CURRENT\_DATE);  
ERROR:  function trunc(date) does not exist  
LINE 1: SELECT TRUNC(CURRENT\_DATE);  
               ^  
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.  
  
  
-- Would need to write custom logic  
SELECT DATE\_TRUNC('month', CURRENT\_DATE);  
-- Or  
SELECT (DATE\_TRUNC('month', CURRENT\_DATE))::date;  
       date\_trunc  
------------------------  
 2025-06-01 00:00:00+00  
(1 row)  
  
postgres=# SELECT oracle.TRUNC(CURRENT\_DATE, 'MM');  
   trunc  
------------  
 2025-06-01  
(1 row)  
  
set search\_path = oracle, pg\_catalog, "$user", public;  
  
SELECT TRUNC(CURRENT\_DATE, 'MM') FROM DUAL;  
   trunc  
------------  
 2025-06-01  
(1 row)

##### DECODE:

--Oracle  
SQL> SELECT id, decode(name, 'ABC', 'Testing', NULL, 'Empty', 'Not Assigned') vname FROM test;  
 ID   VNAME  
----  --------------  
  1   Testing  
  2   Empty  
  3   Not Assigned  
(3 rows)

--PostgreSQL  
CREATE TABLE test(id INTEGER, name varchar(10));  
  
insert into test values(1, 'ABC');  
insert into test values(2, null);  
insert into test values(3, 'XYZ');  
  
  
  
--PostgreSQL: Converting DECODE in native style  
SELECT id,   
    CASE   
        WHEN name = 'ABC' THEN 'Testing'  
        WHEN name IS NULL THEN 'Empty'  
        ELSE 'Not Assigned'  
      END vname FROM test;

--Using orafce decode  
SELECT id, decode(name, 'ABC', 'Testing', NULL, 'Empty', 'Not Assigned') vname FROM test;  
  vname  
---------  
 Testing  
 Empty  
(2 rows)

### 5.2 plpgsql\_check

The plpgsql\_check extension is a crucial tool for PostgreSQL developers, especially during Oracle to PostgreSQL migrations. This powerful code analyzer acts as an early warning system, identifying semantic issues in your PL/pgSQL code before they escalate into production problems.

**Key benefits:**

• Catches semantic errors missed by PostgreSQL's syntax-only checks

• Identifies issues at the development stage, saving time and resources

• Scans for syntax errors, undefined variables, unused variables, and hidden bugs

• Provides precise error locations and clear explanations for quick fixes

• Particularly valuable for large, complex codebases

By integrating plpgsql\_check into your development workflow, you can:

• Ensure cleaner, more reliable code

• Significantly reduce debugging time

• Streamline the Oracle to PostgreSQL migration process

• Boost overall code quality and maintainability  
  
**Challenges with semantic checks in PostgreSQL**

**Example 1:**

CREATE OR REPLACE FUNCTION calculate\_area(p\_length NUMERIC)

RETURNS INTEGER AS $$

DECLARE

    v\_dt1      DATE;

BEGIN

    SELECT current\_date INTO v\_dt1 FROM DUAL;  /\* Bug1 - table DUAL does not exist in PostgreSQL \*/

    IF p\_length <= 0 THEN

        RETURN 0;

    END IF;

    RETURN p\_length\*p\_width;    /\* Bug2 - variable p\_width is not declared within the function scope \*/

END; $$

LANGUAGE plpgsql;

pgday=# SELECT calculate\_area(4.5);

ERROR:  relation "dual" does not exist

LINE 1: SELECT current\_date            FROM DUAL

                                            ^

QUERY:  SELECT current\_date            FROM DUAL

CONTEXT:  PL/pgSQL function calculate\_area(numeric) line 5 at SQL statement

CREATE OR REPLACE FUNCTION calculate\_area(p\_length NUMERIC)

RETURNS INTEGER AS $$

DECLARE

    v\_dt1      DATE;

BEGIN

    SELECT current\_date INTO v\_dt1 ;

    IF p\_length <= 0 THEN

        RETURN 0;

    END IF;

    RETURN p\_length\*p\_width;    /\* Bug2 - variable p\_width is not declared within the function scope \*/

END; $$

LANGUAGE plpgsql;

pgday=# SELECT calculate\_area(4.5);

ERROR:  column "p\_width" does not exist

LINE 1: p\_length\*p\_width

                 ^

QUERY:  p\_length\*p\_width

CONTEXT:  PL/pgSQL function calculate\_area(numeric) line 11 at RETURN

Example -2 :

CREATE TABLE t1(

a int,

b int

);

CREATE OR REPLACE FUNCTION f2()

RETURNS void

LANGUAGE plpgsql

AS

$$

DECLARE

rec record;

BEGIN

FOR rec IN SELECT \* FROM t1

LOOP

RAISE NOTICE '%', rec.c; /\* Bug1: Columns ”c” does not exist in table “t” \*/

END LOOP;

END;

$$;

/\* No semantic issues found as table "t" is empty and control does not go inside the loop \*/

SELECT f2();

Insert into t1 values (1, 2);

SELECT f2();

##### Common semantic issues that plpgsql\_check can identify

* **Undefined variables**
* **Type mismatches**
* **Control flow problems**
* **Incorrect function calls**
* **Trigger-related issues**
* **Incorrect use of SQL statements**
* **Incorrect use of subqueries**
* **Incorrect use of cursors**
* **Incorrect use of arrays**

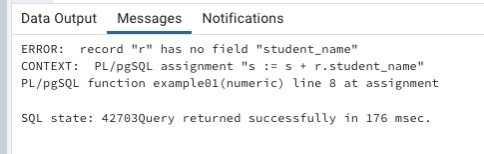
##### Important semantic check functions ****plpgsql\_check\_function plpgsql\_check\_function\_tb****

Create the following table to understand how the plpgsql\_check extensions work with a few PL/pgSQL code examples:

/\* Create a table named as “bigtable” \*/  
CREATE TABLE bigtable(  
    id   integer,  
    name text,  
      constraint bigtable\_pk primary key(id)  
);  
  
/\* Inserting sample data into “bigtable” \*/  
INSERT INTO bigtable  
SELECT i, 'name-'||i  
FROM generate\_series(1, 1000) i;  
  
/\* Sample Function that reads data from the “bigtable” based on the “id” passed \*/  
CREATE OR REPLACE FUNCTION example01(\_id numeric)  
RETURNS numeric AS $$  
DECLARE  
    r record;  
    s numeric DEFAULT 0;  
BEGIN  
    FOR r IN SELECT \* FROM bigtable WHERE id = \_id  
    LOOP  
        s := s + r.student\_name;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;

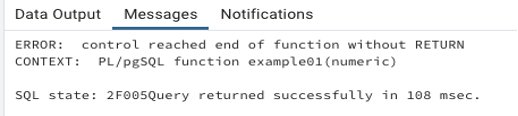
In your first run, pass an input to the function with value 1000:

SELECT example01(1000);



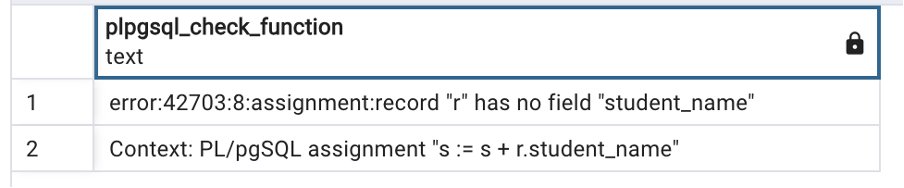
In a second run, change the input parameter to 1001:

SELECT example01(1001);

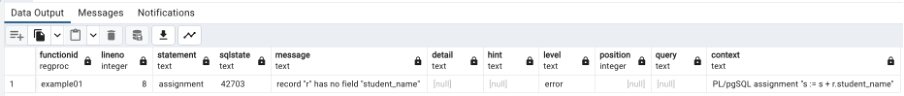


Check the syntax and semantics of the function discussed in the previous section using plpgsql\_check\_function:

SELECT \* FROM plpgsql\_check\_function('example01(numeric)');



SELECT \* FROM plpgsql\_check\_function\_tb('example01(numeric)');



Both plpgsql\_check\_function and plpgsql\_check\_function\_tb don’t reveal all semantic errors by default. You need to pass **fatal\_errors** optional arguments to get the full list of semantic errors within your PL/pgSQL code.

SELECT \* FROM plpgsql\_check\_function\_tb('example01', fatal\_errors=>false);

Let’s fix all the issues reported within the function and compile once again to check if the plpgsql\_check\_function\_tb function reports any other bugs:

CREATE OR REPLACE FUNCTION example01(\_id numeric)

RETURNS numeric

LANGUAGE plpgsql

AS $function$

DECLARE

r record;

s numeric DEFAULT 0;

BEGIN

FOR r IN SELECT \* FROM bigtable WHERE id = \_id

LOOP

s := s + r.id;

END LOOP;

RETURN s;

END;

$function$;

SELECT \* FROM plpgsql\_check\_function\_tb('example01', fatal\_errors=>false);

##### Semantic checks for triggers in PostgreSQL

/\* Create sample table \*/  
CREATE TABLE employees\_tg(  
    id                INTEGER,  
    first\_name VARCHAR(40) NOT NULL,  
    last\_name  VARCHAR(40) NOT NULL,  
      CONSTRAINT employees\_pk PRIMARY KEY(id)  
);  
  
INSERT INTO employees\_tg VALUES(1, 'David', 'Mellon');  
INSERT INTO employees\_tg VALUES(2, 'John', 'Kett');  
INSERT INTO employees\_tg VALUES(3, 'Anjello', 'Pie');  
  
CREATE TABLE audit\_employees\_deleted(  
    id                INTEGER,  
    first\_name VARCHAR(40) NOT NULL,  
    last\_name  VARCHAR(40) NOT NULL,  
      CONSTRAINT audit\_employees\_deleted\_pk PRIMARY KEY(id)  
);

CREATE OR REPLACE FUNCTION fnc\_trg\_delete\_employee()

RETURNS TRIGGER AS $$

BEGIN

INSERT INTO audit\_employees\_deleted

VALUES (NEW.\*);

END;

$$ LANGUAGE plpgsql;

In the preceding trigger function, there are two bugs:

* The RETURN keyword is missing
* DELETE triggers can’t refer to NEW

However, the compilation is successful, and because trigger functions are a special type of functions, you can’t test simply by calling them.

postgres=# SELECT fnc\_trg\_delete\_employee();  
ERROR:  trigger functions can only be called as triggers  
CONTEXT:  compilation of PL/pgSQL function "fnc\_trg\_delete\_employee" near line 1

Let’s create a trigger that is activated when you delete any record from the employees table and calls the trg\_fnc\_delete\_employee() function to perform an insert operation in the audit\_employees\_deleted table:

CREATE TRIGGER trg\_delete\_employee

BEFORE DELETE

ON employees\_tg

FOR EACH ROW

EXECUTE FUNCTION fnc\_trg\_delete\_employee();

##### Test manually to see the semantic issue

DELETE FROM employees\_tg  
WHERE id = 2;  
ERROR:  null value in column "id" of relation "audit\_employees\_deleted" violates not-null constraint  
DETAIL:  Failing row contains (null, null, null).  
CONTEXT:  SQL statement "INSERT INTO audit\_employees\_deleted VALUES (NEW.\*)"  
PL/pgSQL function fnc\_trg\_delete\_employee() line 3 at SQL statement

##### Semantic checks for triggers using plpgsql\_check\_function\_tb

Since trigger function and the trigger are tightly coupled and associated with a table, you need to specify the table name along with the trigger function name

SELECT \* FROM plpgsql\_check\_function\_tb(' fnc\_trg\_delete\_employee()', 'employees\_tg ');

performance\_warnings

SELECT \* FROM plpgsql\_check\_function\_tb('example01', performance\_warnings => true);

drop function example01(\_id numeric);

CREATE OR REPLACE FUNCTION example01(\_id integer)

RETURNS numeric

LANGUAGE plpgsql

STABLE

AS $function$

DECLARE

r record;

s numeric DEFAULT 0;

BEGIN

FOR r IN SELECT \* FROM bigtable WHERE id = \_id

LOOP

s := s + r.id;

END LOOP;

RETURN s;

END;

$function$;

##### Additional utility queries for semantic checks on all code objects

Check all non-trigger plpgsql user created functions:

SELECT p.oid, p.proname, plpgsql\_check\_function(p.oid,fatal\_errors := false)  
FROM pg\_catalog.pg\_namespace n  
JOIN pg\_catalog.pg\_proc p ON pronamespace = n.oid  
JOIN pg\_catalog.pg\_language l ON p.prolang = l.oid  
WHERE l.lanname = 'plpgsql' AND p.prorettype <> 2279;

Check all trigger plpgsql functions:

SELECT p.oid, p.proname, plpgsql\_check\_function(p.oid,fatal\_errors := false)  
FROM pg\_catalog.pg\_namespace n  
JOIN pg\_catalog.pg\_proc p ON pronamespace = n.oid  
JOIN pg\_catalog.pg\_language l ON p.prolang = l.oid  
WHERE l.lanname = 'plpgsql' AND p.prorettype <> 2279;

Check all plpgsql functions (functions or trigger functions with defined triggers):

SELECT (pcf).functionid::regprocedure, (pcf).lineno, (pcf).statement,  
(pcf).sqlstate, (pcf).message, (pcf).detail, (pcf).hint, (pcf).level,  
(pcf)."position", (pcf).query, (pcf).context  
FROM(SELECT  
plpgsql\_check\_function\_tb(pg\_proc.oid, COALESCE(pg\_trigger.tgrelid, 0),fatal\_errors := false) AS pcf  
FROM pg\_proc  
LEFT JOIN pg\_trigger  
ON (pg\_trigger.tgfoid = pg\_proc.oid)  
WHERE  
prolang = (SELECT lang.oid FROM pg\_language lang WHERE lang.lanname = 'plpgsql') AND  
pronamespace <> (SELECT nsp.oid FROM pg\_namespace nsp WHERE nsp.nspname = 'pg\_catalog') AND  
-- ignore unused triggers  
(pg\_proc.prorettype <> (SELECT typ.oid FROM pg\_type typ WHERE typ.typname = 'trigger') OR  
pg\_trigger.tgfoid IS NOT NULL)  
) ss ORDER BY (pcf).functionid::regprocedure::text, (pcf).lineno;

## 6 Conclusion

Based on the document and conversation history, here's the conclusion:

This comprehensive PostgreSQL training document covers essential tools, extensions, and best practices that significantly enhance both developer and DBA productivity. The content progresses logically from fundamental tools like PGAdmin and PSQL to advanced features such as partition management and performance monitoring.

Key takeaways include:

1. Essential tools that provide the foundation for effective database management

2. Performance monitoring and optimization techniques using various extensions

3. Data management solutions for handling large-scale databases

4. Code quality and compatibility tools for maintaining robust applications

5. Testing and validation approaches for ensuring reliability

The document emphasizes practical, hands-on knowledge with real-world examples and use cases. It addresses common challenges in PostgreSQL database management while providing solutions through various tools and extensions. The material is structured to build expertise progressively, making it valuable for both newcomers and experienced database professionals.

The combination of GUI tools, command-line interfaces, monitoring solutions, and management extensions creates a complete toolkit for modern PostgreSQL database management, enabling professionals to work more efficiently while maintaining high standards of performance and reliability.