

Comprehensive PostgreSQL Administration

PostgreSQL 9.6

Course Agenda

- Introduction
- System Architecture
- Installation and Database Clusters
- Configuration
- Creating and Managing Databases
- User Tools - CUI and GUI
- Security
- SQL Primer
- Backup, Recovery and PITR
- Routine Maintenance Tasks
- Data Dictionary
- Loading and Moving Data
- SQL Tuning
- Performance Tuning
- Streaming Replication
- Connection Pooling using pgpool-II
- Table Partitioning
- Extensions
- Monitoring
- Upgrading Best Practices

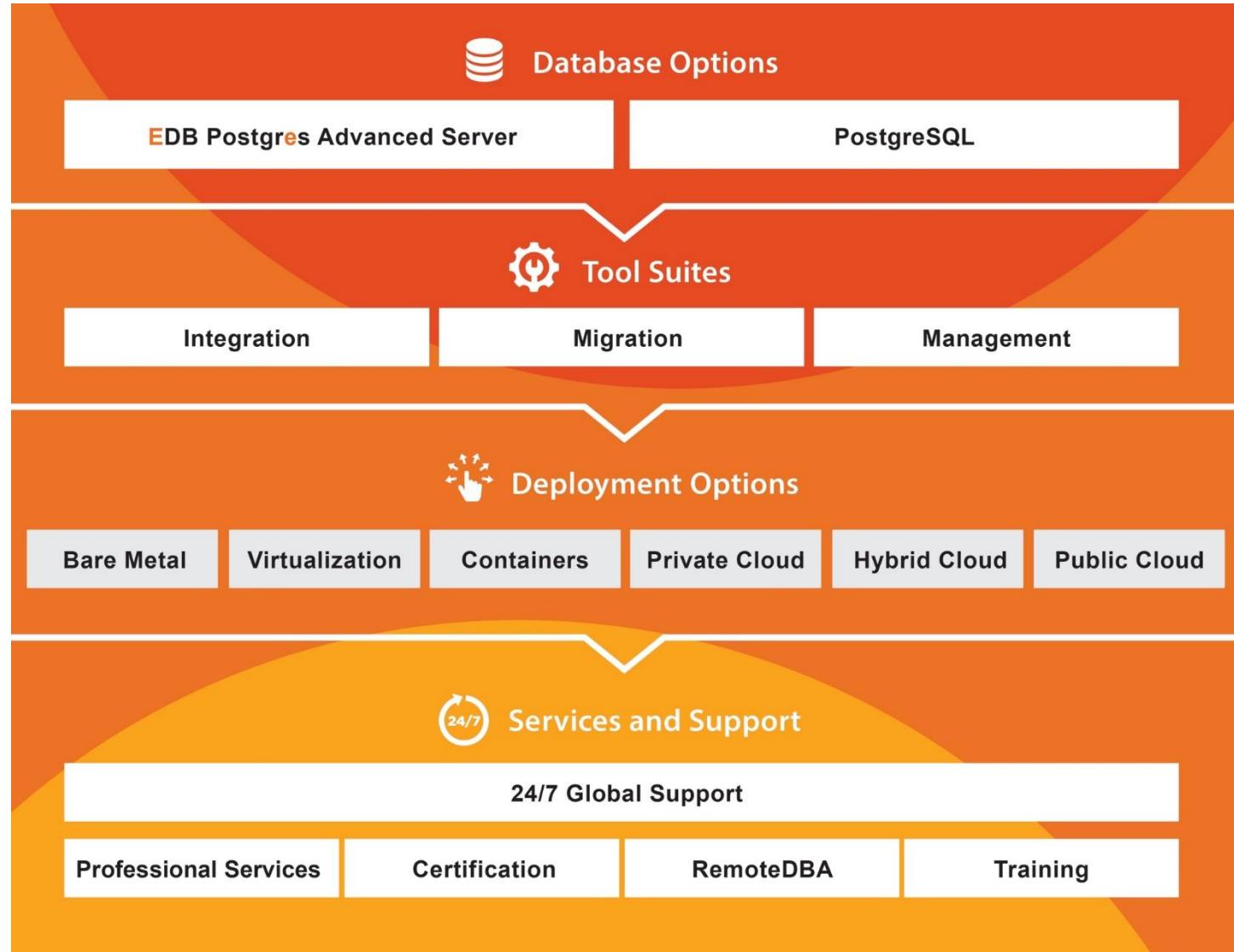
Module 1

Introduction and Architectural Overview

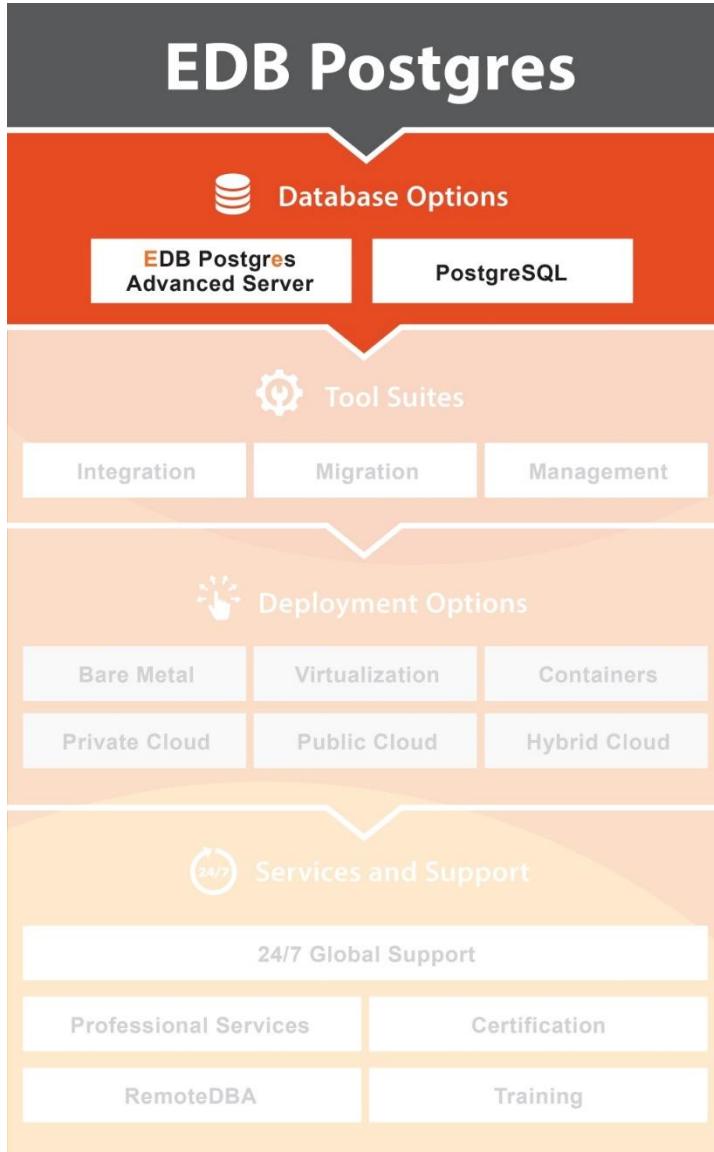
Module Objectives

- EDB Postgres Platform
- History of PostgreSQL
- Major Features
- EDB Postgres Advanced Server Features
- Architectural Overview
- General Database Limits
- Common Database Object Names

EDB Postgres Platform



EDB Postgres Platform – DBMS Options



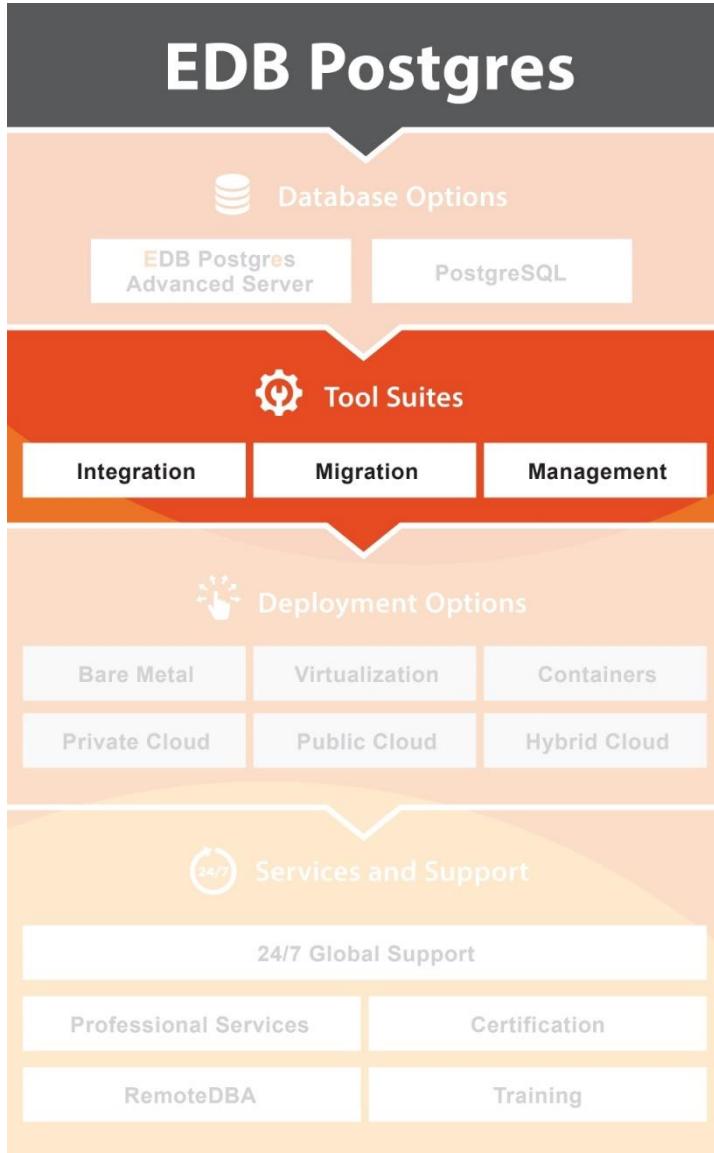
EDB Postgres Advanced Server

- Based on open-source PostgreSQL
- Adds performance, security, and database compatibility for Oracle
- Includes additional developer and DBA features
- EDB Postgres Enterprise:
24x7 support plus management, integration and migration tool suites deployable across many traditional platforms as well as cloud platforms

PostgreSQL

- The world's most advanced open source database
- EDB Postgres Standard:
24x7 support plus management, integration and migration tool suites deployable across many traditional platforms as well as cloud platforms

EDB Postgres Platform – Tool Suites



Integration

- Exchange data across various database management systems in near real time
- EDB Postgres Data Adapters (foreign data wrappers)
- EDB Postgres Replication Server
- EDB Postgres XA Support*

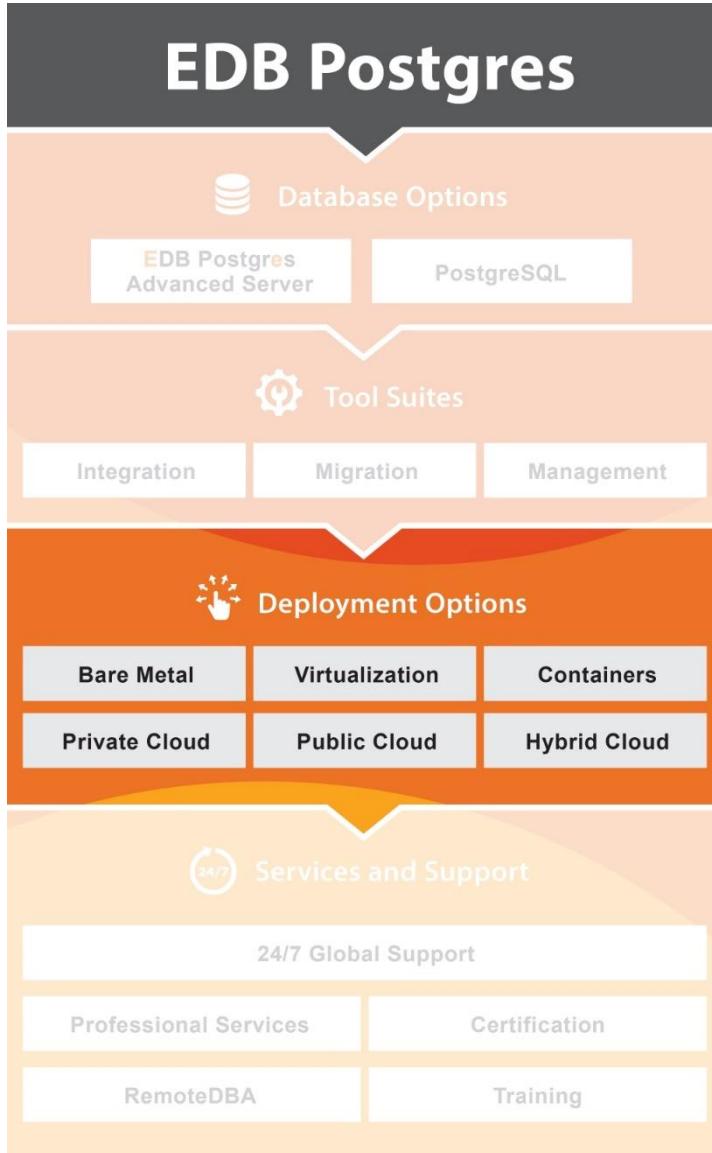
Migration

- Analyze and move your data from expensive legacy databases
- EDB Postgres Migration Assessment with services engagement
- EDB Postgres Migration Toolkit

Management

- Mission-critical tools for management, monitoring, tuning, high availability, as well as backup and disaster recovery
- EDB Postgres Enterprise Manager
- EDB Postgres Failover Manager
- EDB Postgres Backup and Recovery

EDB Postgres Platform – Deployment Options



Bare Metal

- The choice to maximize database simplicity, control, and power
- Tested and certified on the most popular operating systems.

Virtualization

- The choice for database consolidation, cost savings, and hardware efficiency
- Supports VMWare, KVM, Docker containers, and other virtualization platforms

Containers

- Provide small units of granularity to ensure greater utilization of system resources
- EDB is developing a set of containers that intelligently groups various components for ease of use
- Supports OpenShift and other Docker-based container platforms

Private Cloud

- The choice for on-premises control of elastic database: storage, scale-out, scale-up, and self-service provisioning.

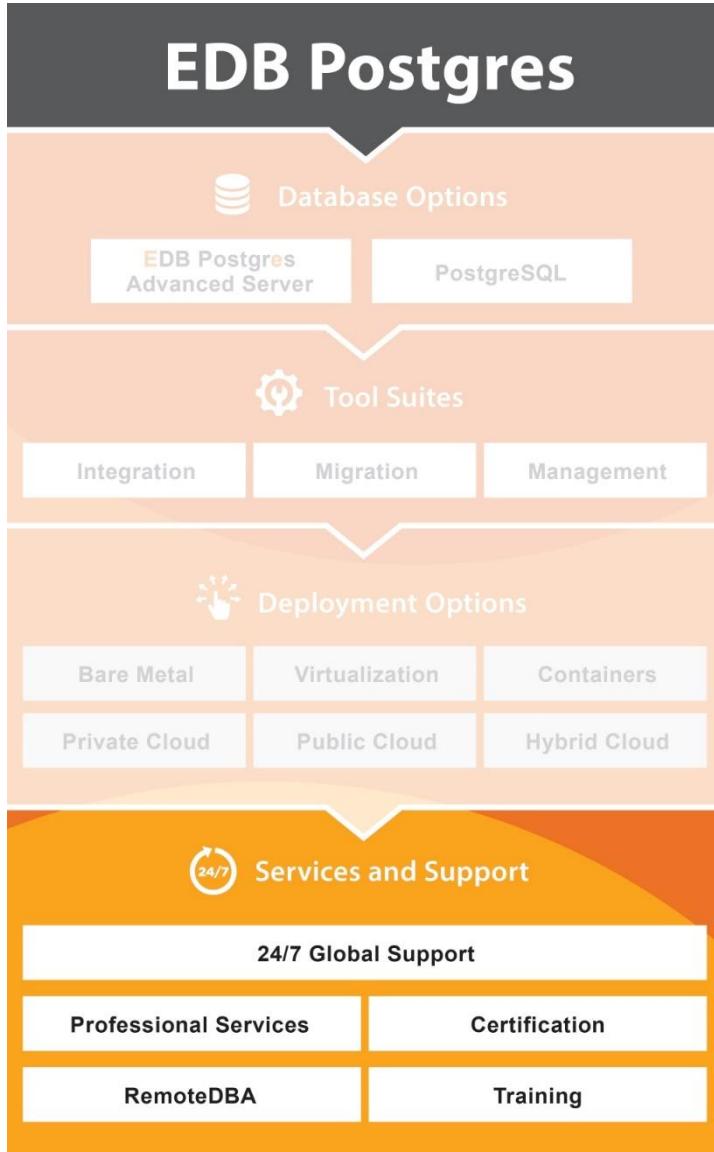
Public Cloud

- The choice for outsourced convenience, flexibility, and cost control of database resources in the cloud.
- Amazon AWS, Google and more to come

Hybrid Cloud

- Applications that span cloud types
- Data exchange and replication

EDB Postgres Platform – Services & Support



24/7 Global Support

- Production Support – 24 x 7 & Developer Support – 10 x 5

Professional Services

- Broad range of consulting services
- Getting Started, Migration, Deployment
- Enterprise Architecture Services, DBA Services, and Technical Account Management

Certification

- Industry-recognized global standard
- PostgreSQL & EDB Postgres programs: Associate-level, Professional-level
- Convenient online exam format

RemoteDBA

- Apply certified DBA personnel to EDB Postgres deployments
- Cost-effective vs. full-time DBA

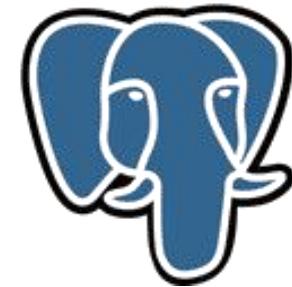
Training

- World-class training to fit your need: classroom & on-site, online, eSubscriptions with DBA & Developer tracks available
- Courses cover administration, migration, performance tuning and more

Facts about PostgreSQL

- The world's most advanced open source database
- Designed for extensibility and customization
- ANSI/ISO compliant SQL support
- Actively developed for more than 20 years
 - University Postgres (1986-1993)
 - Postgres95 (1994-1995)
 - PostgreSQL (1996-current)

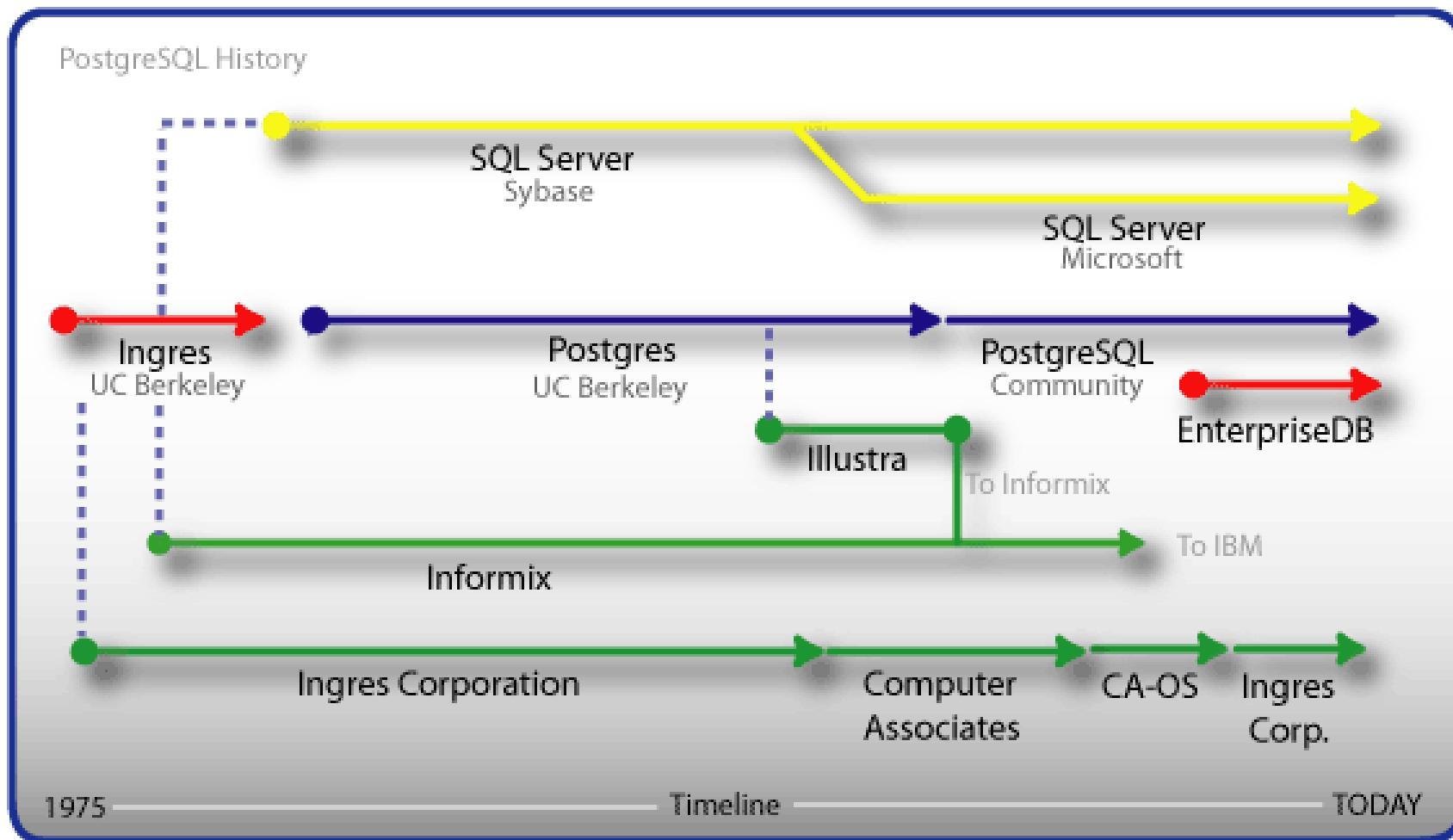
PostgreSQL



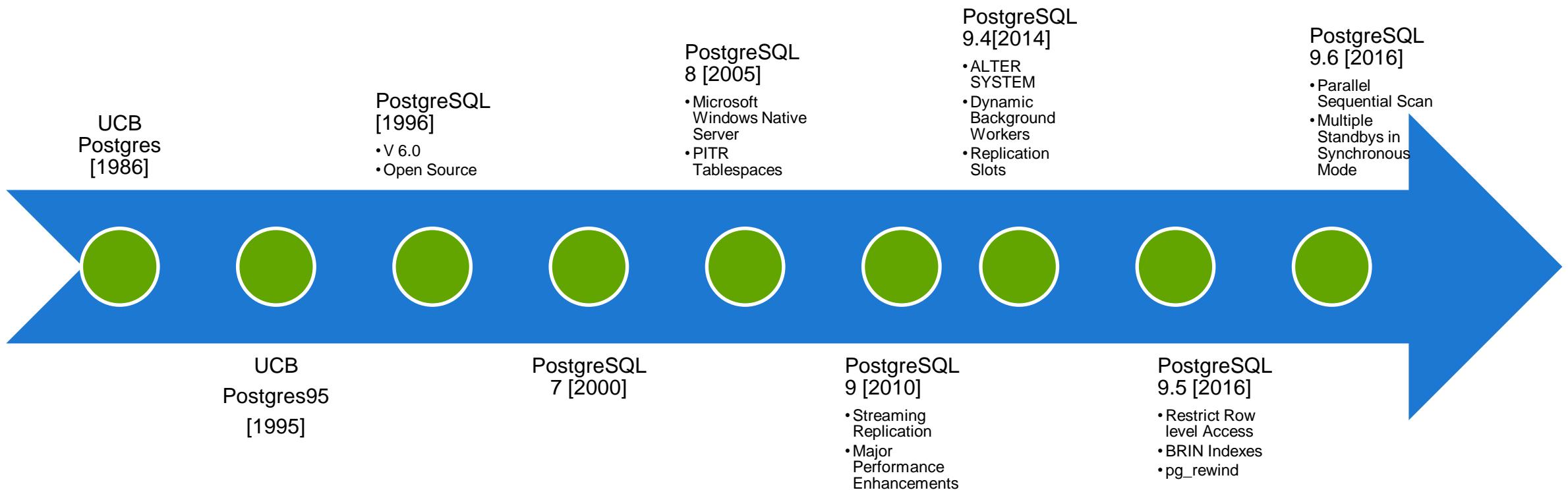
PostgreSQL Community

- Community mailing lists
www.Postgresql.org/community/lists
- Support Forums
<http://forums.enterprisedb.com/forums/list.page>
- Commercial SLAs
www.enterprisedb.com/products-services-training/subscriptions

PostgreSQL Lineage



PostgreSQL Version History



Major Features of PostgreSQL

- Portable:
 - Written in ANSI C
 - Supports Windows, Linux, Mac OS/X and major UNIX platforms
- Reliable:
 - ACID Compliant
 - Supports Transactions and Savepoints
 - Uses Write Ahead Logging (WAL)
- Scalable:
 - Uses Multi-version Concurrency Control
 - Supports Table Partitioning
 - Supports Tablespace
 - Supports Parallel Sequential Scans

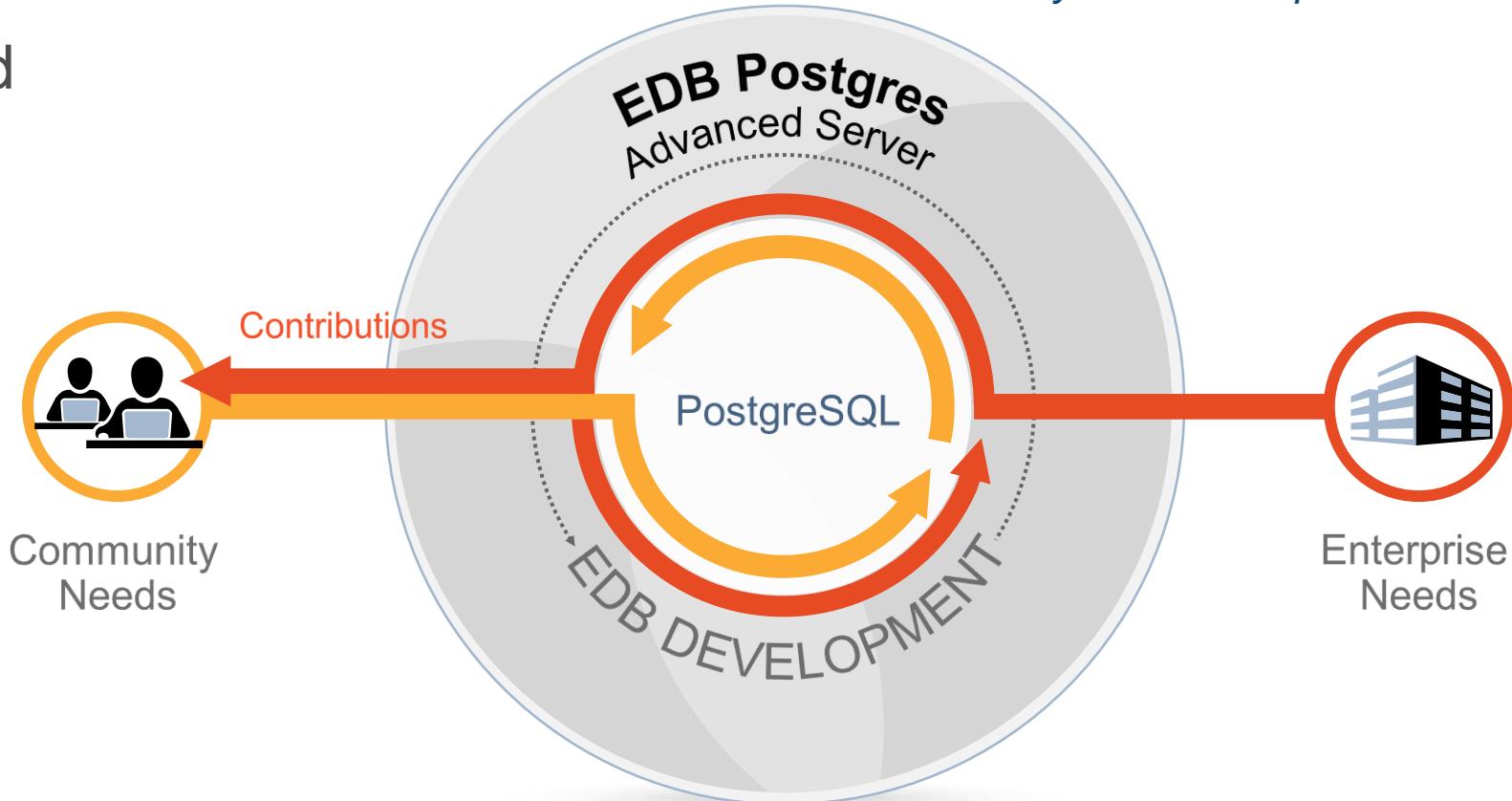
Major Features of PostgreSQL (continued)

- Secure:
 - Employs Host-Based Access Control
 - Provides Object-Level Permissions and Row Level Security
 - Supports Logging
 - SSL
- Recovery and Availability:
 - Streaming Replication
 - Replication Slots, Sync or Async Options
 - Supports Hot-Backup, **pg_basebackup**
 - Point-in-Time Recovery
- Advanced:
 - Supports Triggers and Functions
 - Supports Custom Procedural Languages
 - Upgrade using **pg_upgrade**
 - Unlogged Tables and Materialized Views

Facts about EDB Postgres Advanced Server

- EDB Postgres Advanced Server is built on rock-solid PostgreSQL foundation
- Enhanced version of PostgreSQL
 - Enhanced Performance
 - Database Compatibility
 - Enhanced Security
- Capable low cost alternative to costly proprietary databases

Continuously synchronized with PostgreSQL for a super-set of community PLUS enterprise features



EDB Postgres Advanced Server blends PostgreSQL with Enterprise level requirements

EDB POSTGRES

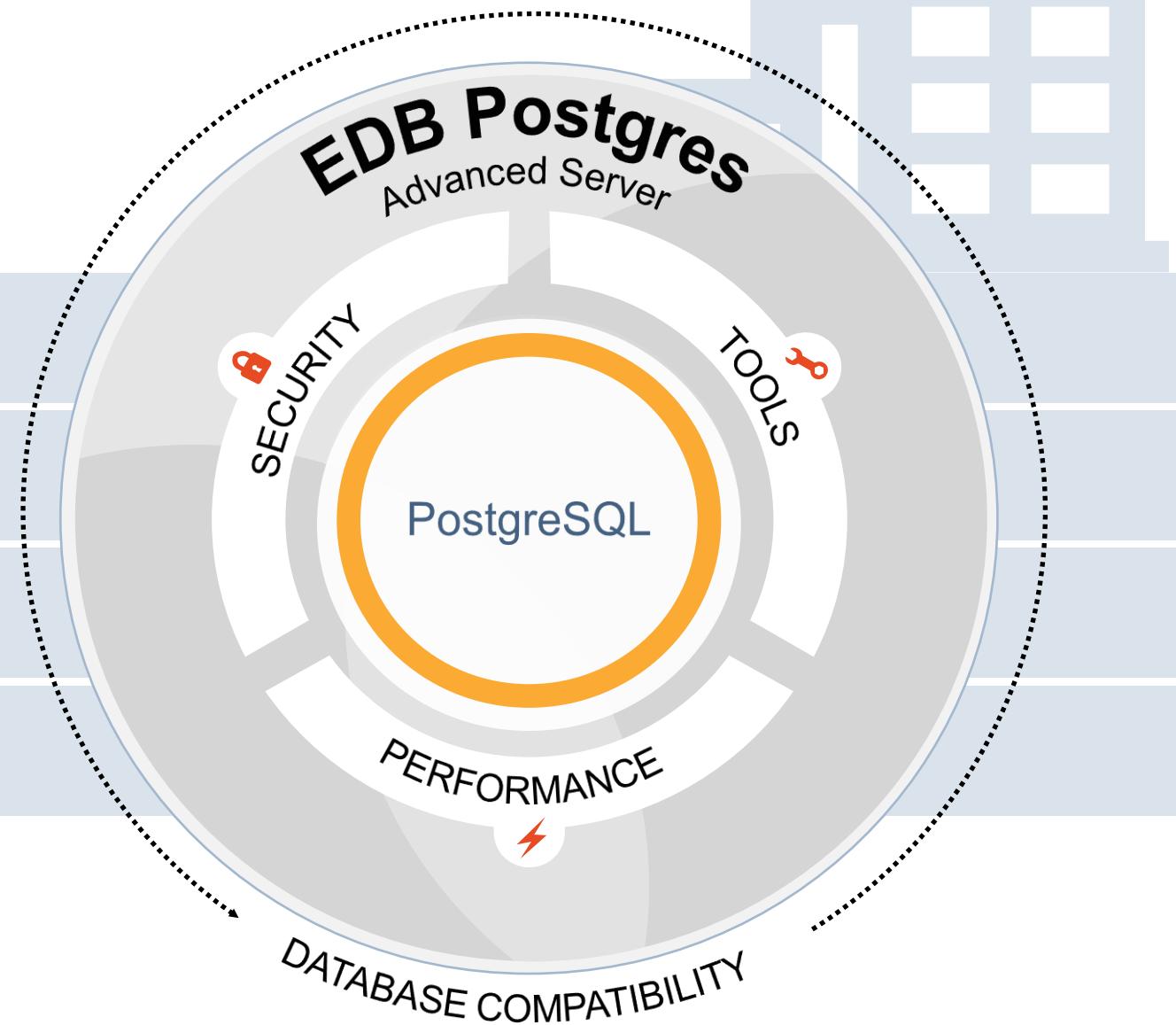
ADVANCED SERVER

Security

Tools

Performance

Compatibility

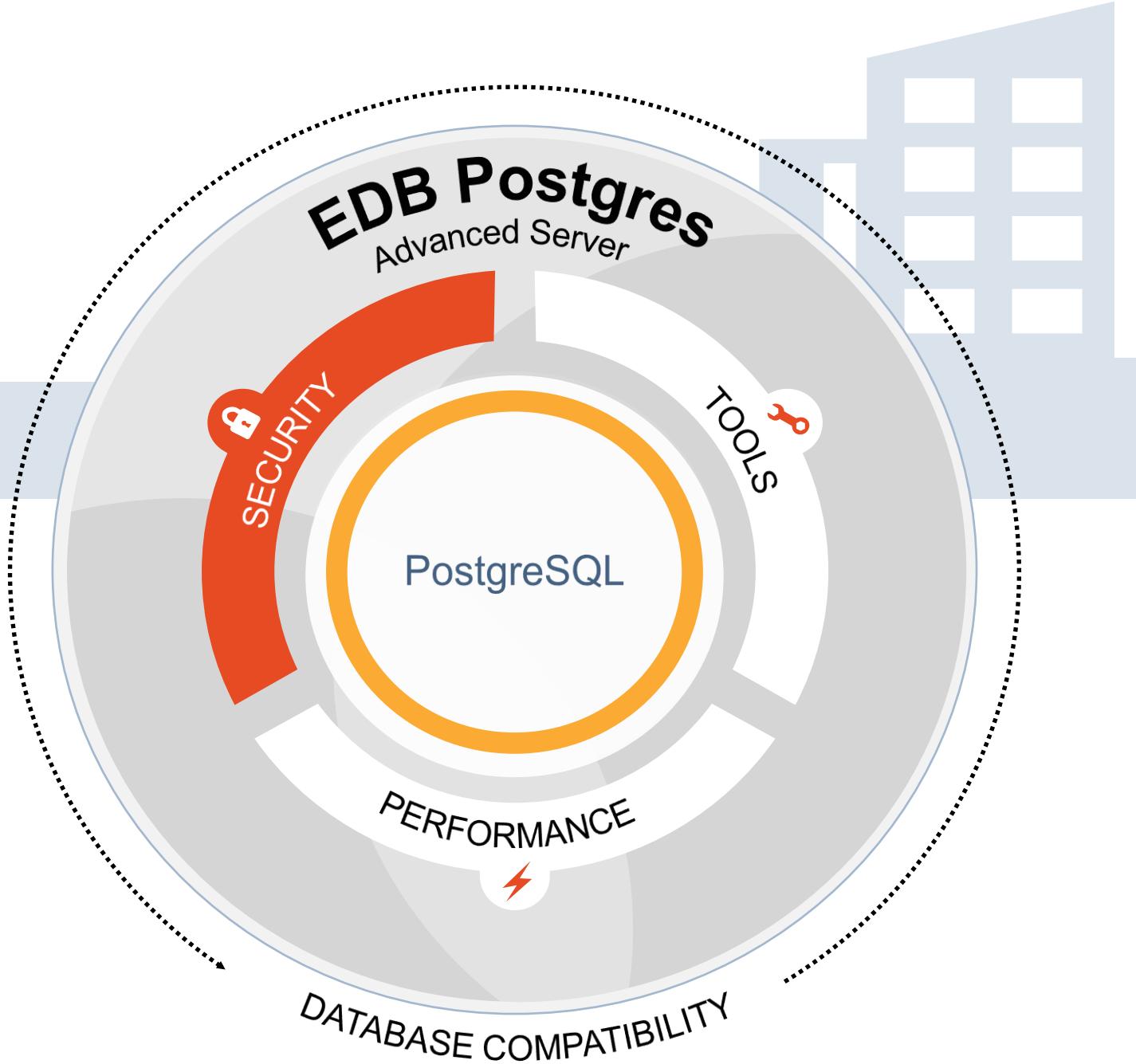


EDB POSTGRES

ADVANCED SERVER

Security

- User account / password policy management
- Enhanced Auditing
- Row Level Security (VPD)
- SQL Injection attack guard
- Server-side code protection
- Multiple US Gov't certifications including EAL2

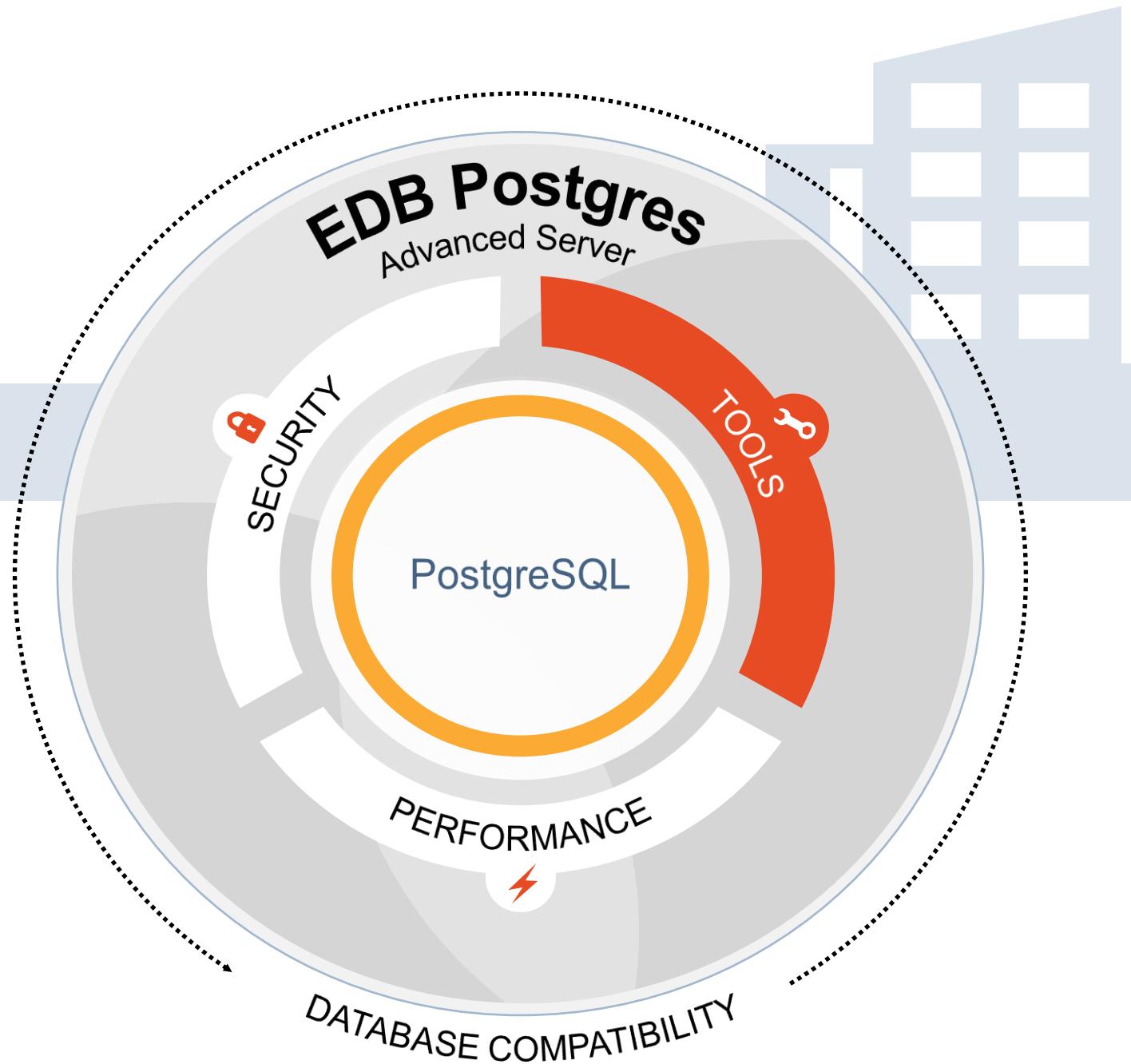


EDB POSTGRES

ADVANCED SERVER

Bundled Tools

- Enterprise management, monitoring, and tuning
- HA failover management
- Oracle, SQL Server, and PostgreSQL to EDB Postgres Advanced Server replication
- Multi-master replication
- Oracle, SQL Server, and MySQL to EDB Postgres Advanced Server migration
- Update monitoring

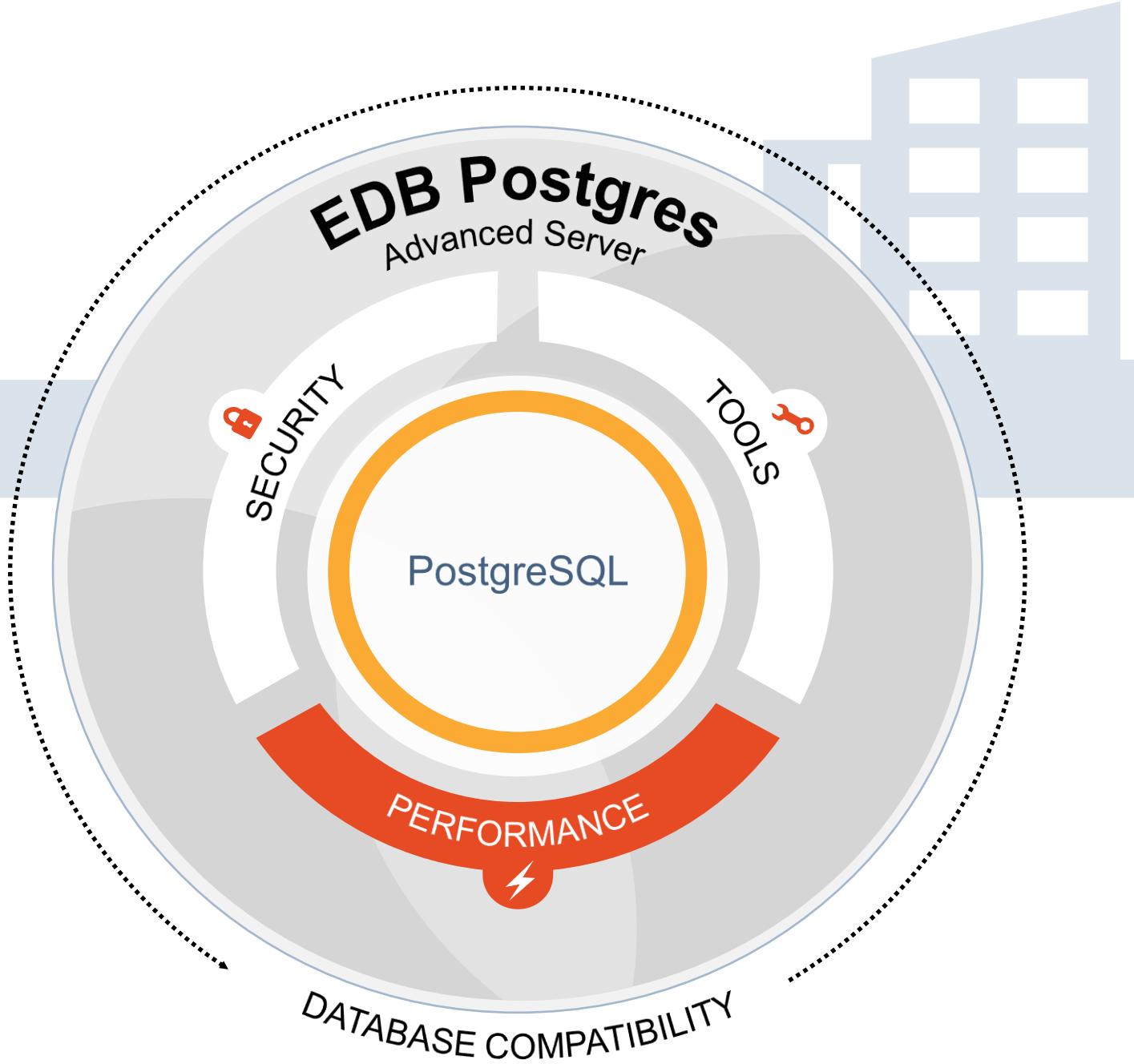


EDB POSTGRES

ADVANCED SERVER

Performance

- Resource Manager – *adjust CPU & I/O resources on mixed workloads*
- Faster Partitioning - *400x faster writes & 76x faster selects*
- SQL Profiler – *fix slow workloads*
- Bulk Data Loader - *2x faster*
- Index Advisor - *speeds up inquiries*
- Query Hints - *optimizer control*
- DynaTune - *memory upgrades*
- Bulk Collect/Fetch/Binding of arrays
- Dynamic runtime statistics - *reveals SQL wait bottlenecks*

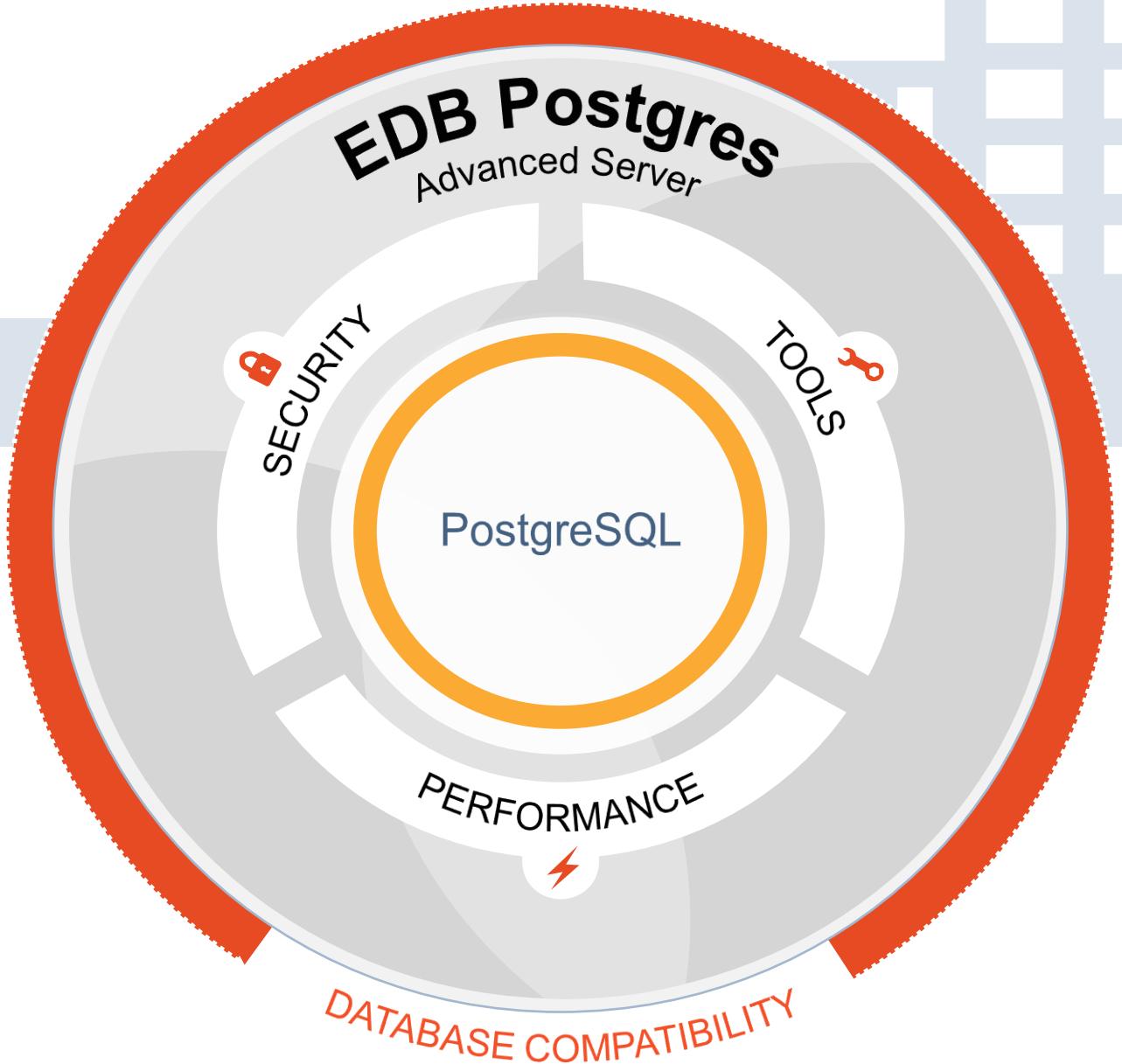


EDB POSTGRES

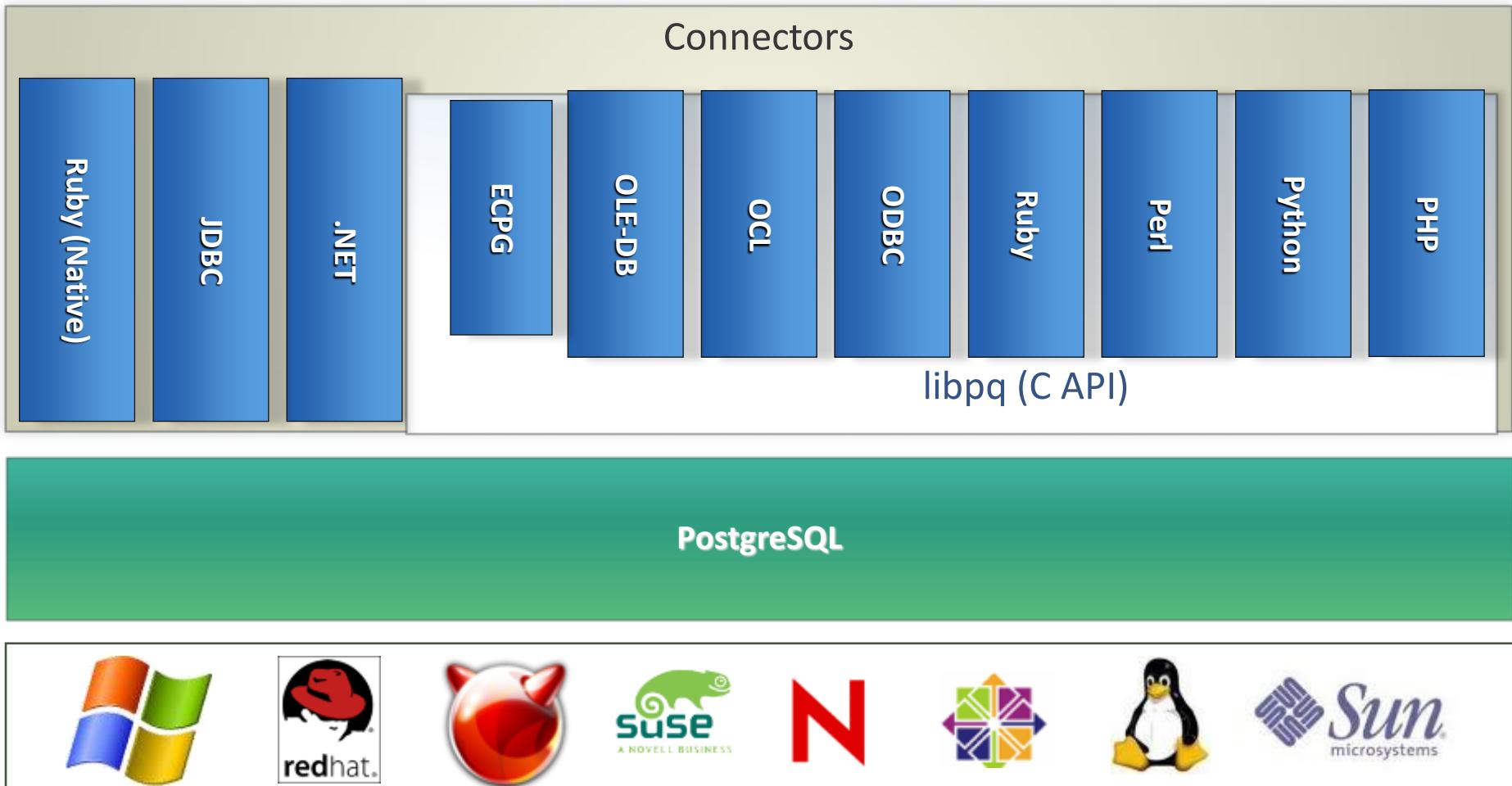
ADVANCED SERVER

Database Compatibility for Oracle®

- Faster, easier migrations
- PL/SQL, OCI support
- Oracle SQL extensions
- User defined objects
- Function packages
- Database links
- Oracle-like tools:
*EDB*Loader, EDB*Plus,
EDB*Wrap*



Architectural Overview



General Database Limits

Limit	Value
Maximum Database Size	Unlimited
Maximum Table Size	32 TB
Maximum Row Size	1.6 TB
Maximum Field Size	1 GB
Maximum Rows per Table	Unlimited
Maximum Columns per Table	250-1600 (Depending on Column types)
Maximum Indexes per Table	Unlimited

Common Database Object Names

Industry Term	PostgreSQL Term
Table or Index	Relation
Row	Tuple
Column	Attribute
Data Block	Page (when block is on disk)
Page	Buffer (when block is in memory)

Module Summary

- EDB Postgres Platform
- History of PostgreSQL
- Major Features
- EDB Postgres Advanced Server Features
- Architectural Overview
- General Database Limits
- Common Database Object Names

Module 2

System Architecture

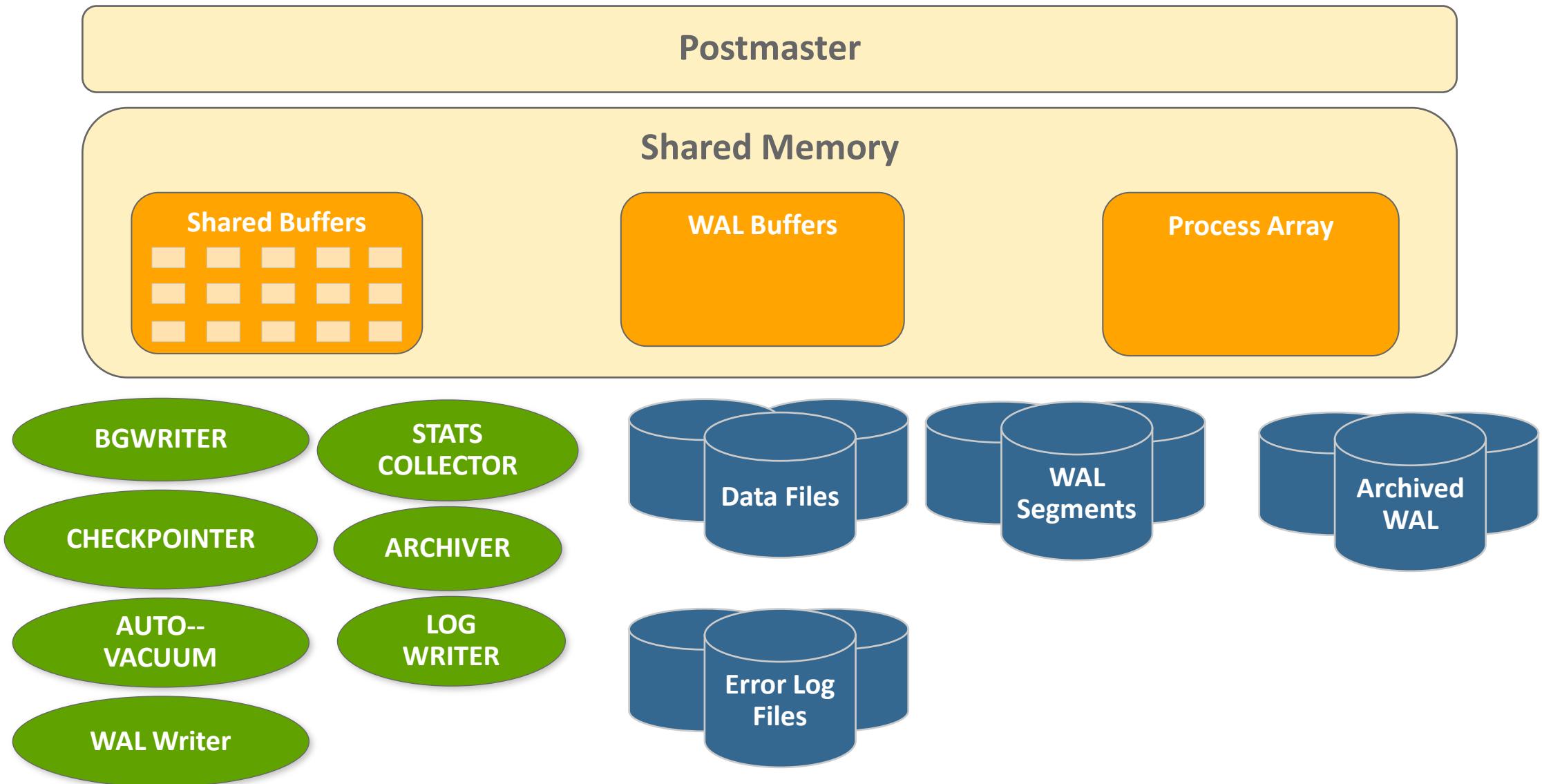
Module Objectives

- Architectural Summary
- Process and Memory Architecture
- Utility Processes
- Connection Request-Response
- Disk Read Buffering
- Disk Write Buffering
- Background Writer Cleaning Scan
- Commit and Checkpoint
- Statement Processing
- Physical Database Architecture
- Data Directory Layout
- Installation Directory Layout
- Page Layout

Architectural Summary

- PostgreSQL uses processes, not threads
- The postmaster process acts as a supervisor
- Several utility processes perform background work
 - postmaster starts them, restarts them if they die
- One back end process per user session
 - postmaster listens for new connections

Process and Memory Architecture



Utility Processes

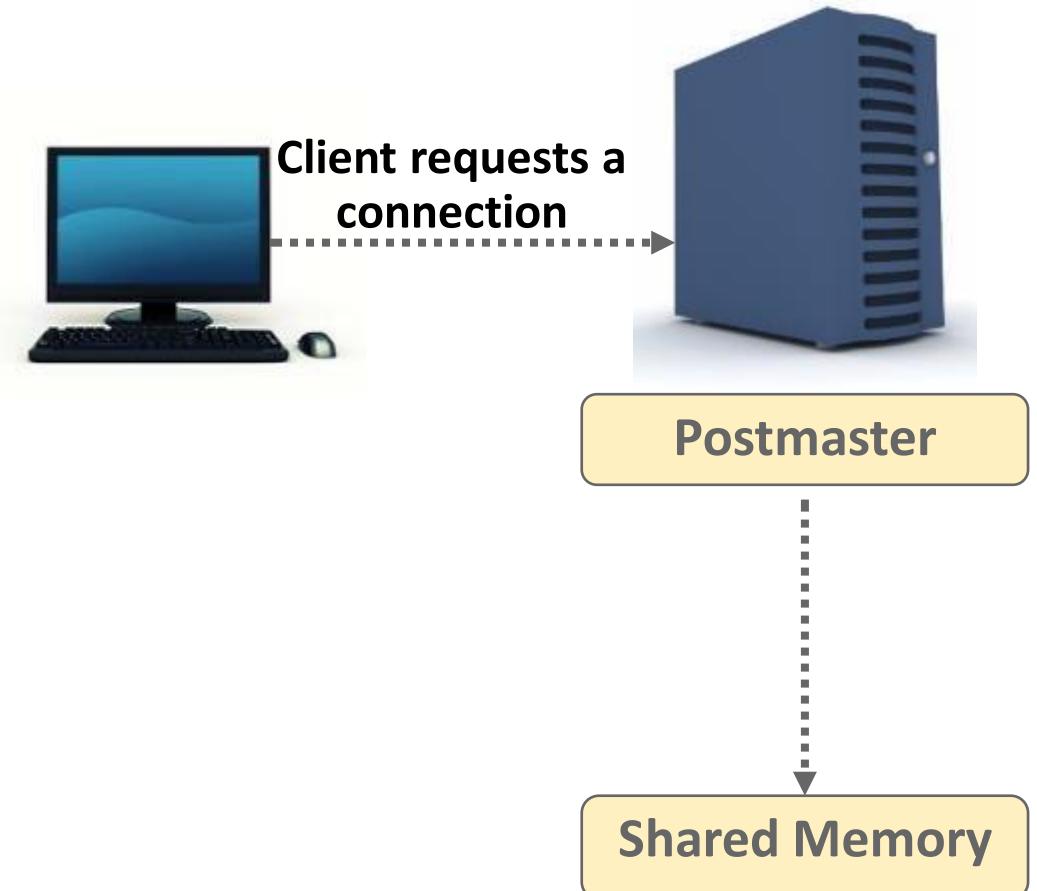
- Background writer
 - Writes dirty data blocks to disk
- WAL writer
 - Flushes write-ahead log to disk
- Checkpointer process
 - Automatically performs a checkpoint based on config parameters
- Autovacuum launcher
 - Starts Autovacuum workers as needed
- Autovacuum workers
 - Recover free space for reuse

More Utility Process

- Logging collector
 - Routes log messages to syslog, eventlog, or log files
- Stats collector
 - Collects usage statistics by relation and block
- Archiver
 - Archives write-ahead log files

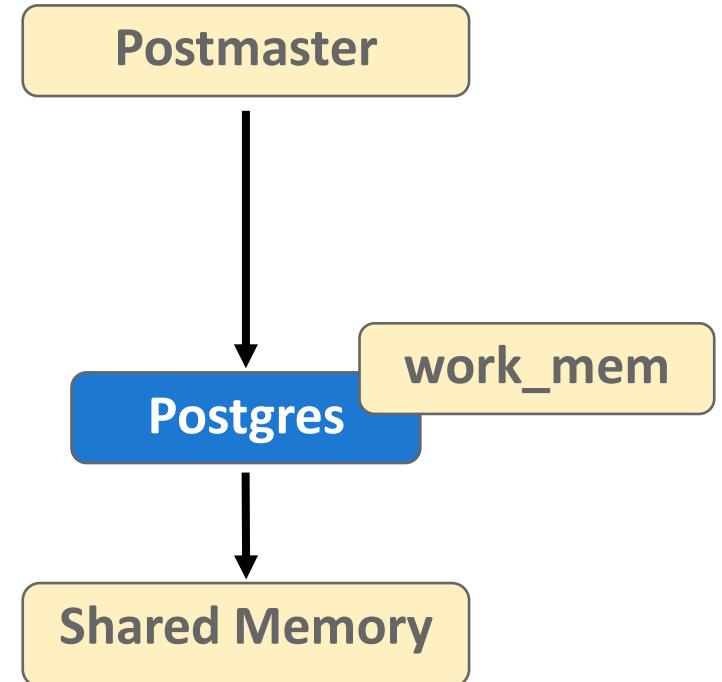
Postmaster as Listener

- Postmaster is master process called postgres
- Listens on 1, and only 1, tcp port
- Receives client connection requests



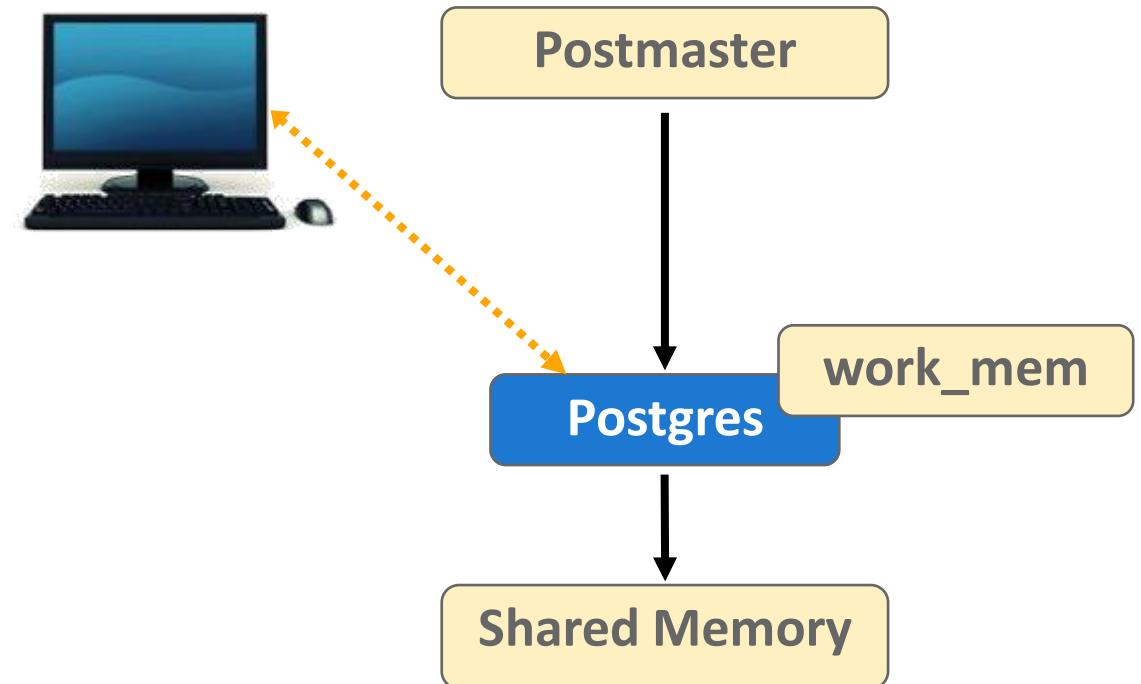
User Back End Process

- Master process postgres spawns a new server process for each connection request detected
- Communication is done using semaphores and shared memory
- Authentication - IP, user and password
- Authorization - Verify Permissions



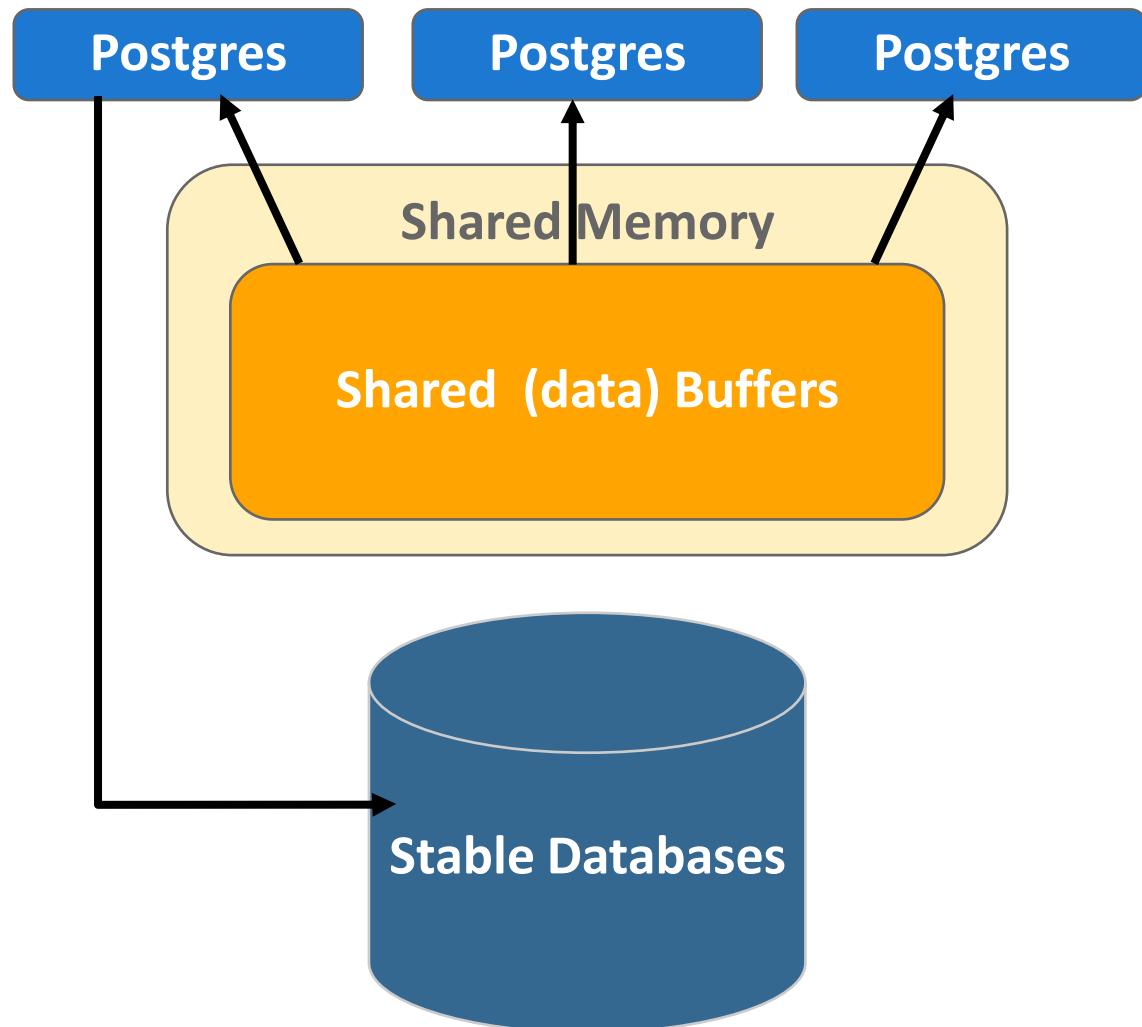
Respond to Client

- User back end process called postgres
- Callback to client
- Waits for SQL
- Query is transmitted using plain text



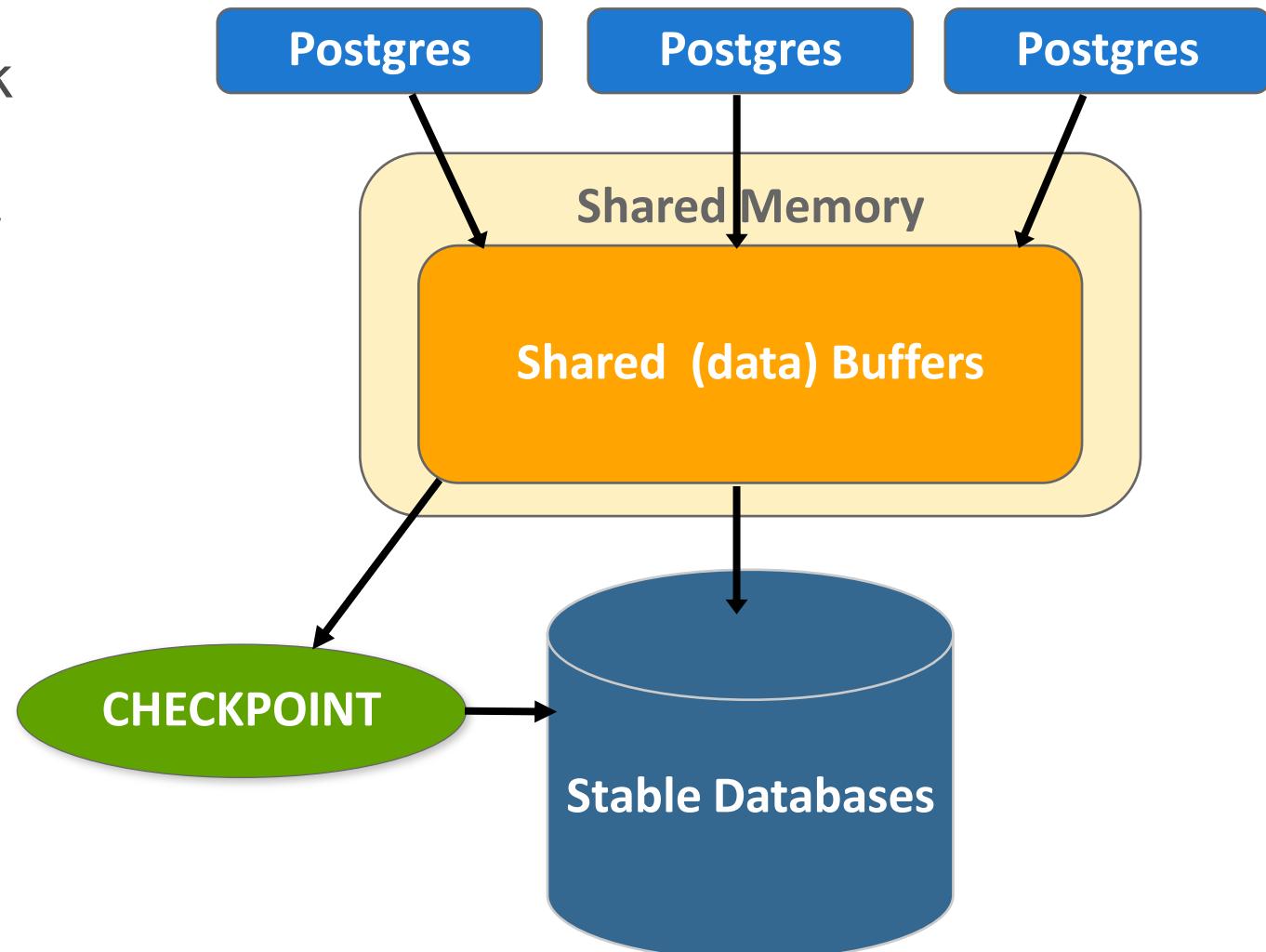
Disk Read Buffering

- PostgreSQL buffer cache (shared_buffers) reduces OS reads.
- Read the block once, then examine it many times in cache.



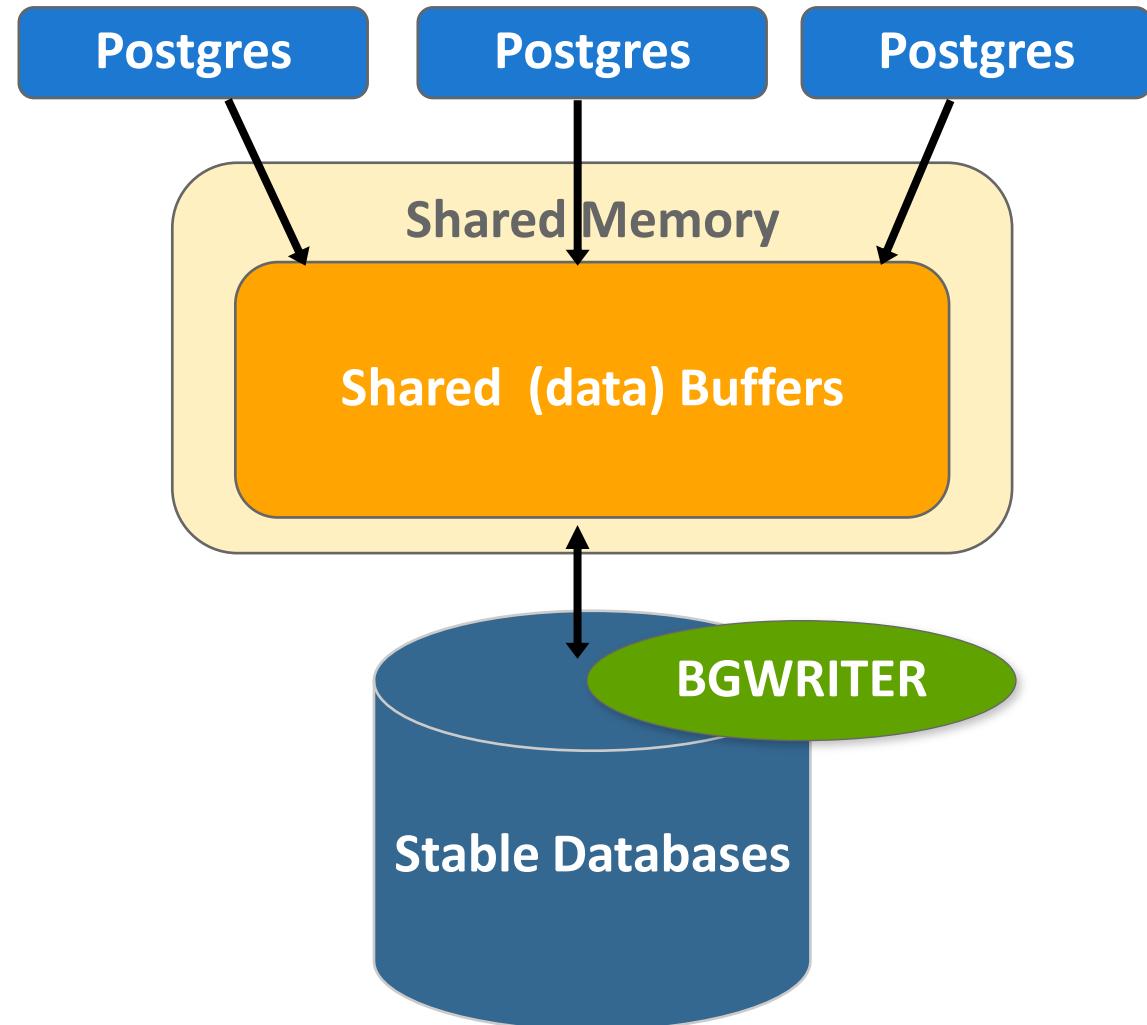
Disk Write Buffering

- Blocks are written to disk only when needed:
 - To make room for new blocks
 - At checkpoint time



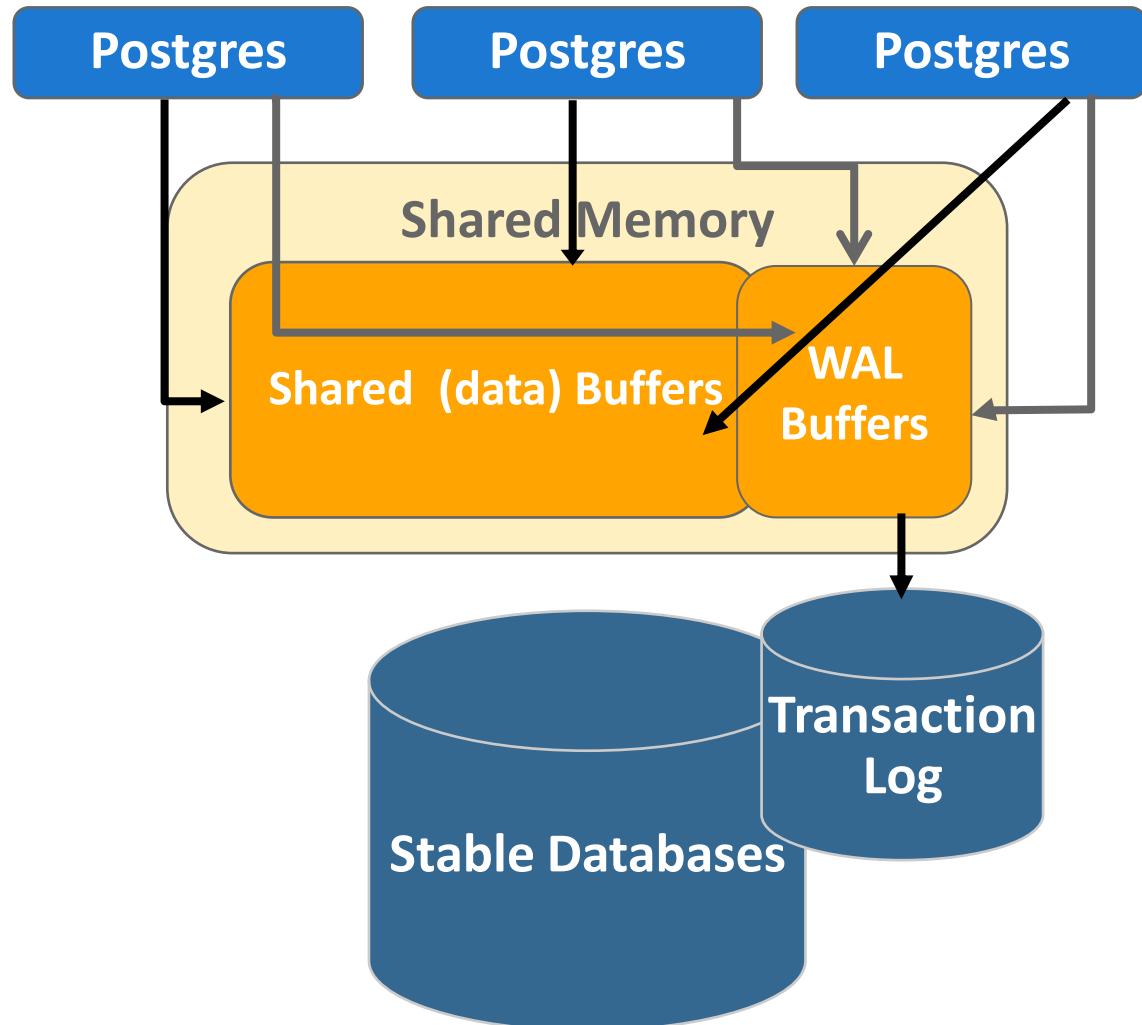
Background Writer Cleaning Scan

- Background writer scan attempts to ensure an adequate supply of clean buffers
- Back end write dirty buffers at need



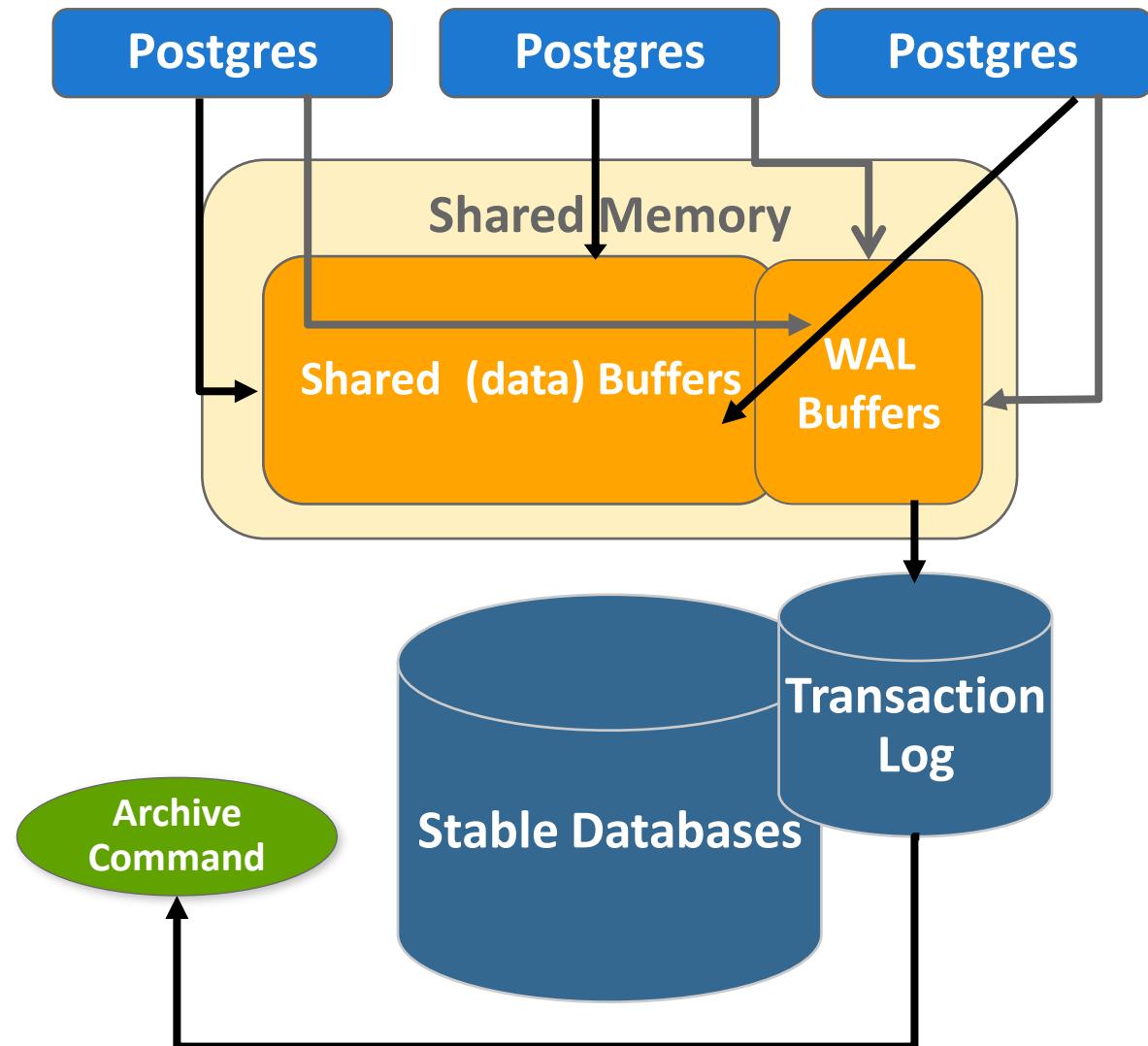
Write Ahead Logging (WAL)

- Back end write data to WAL buffers
- Flush WAL buffers periodically (WAL writer), on commit, or when buffers are full
- Group commit



Transaction Log Archiving

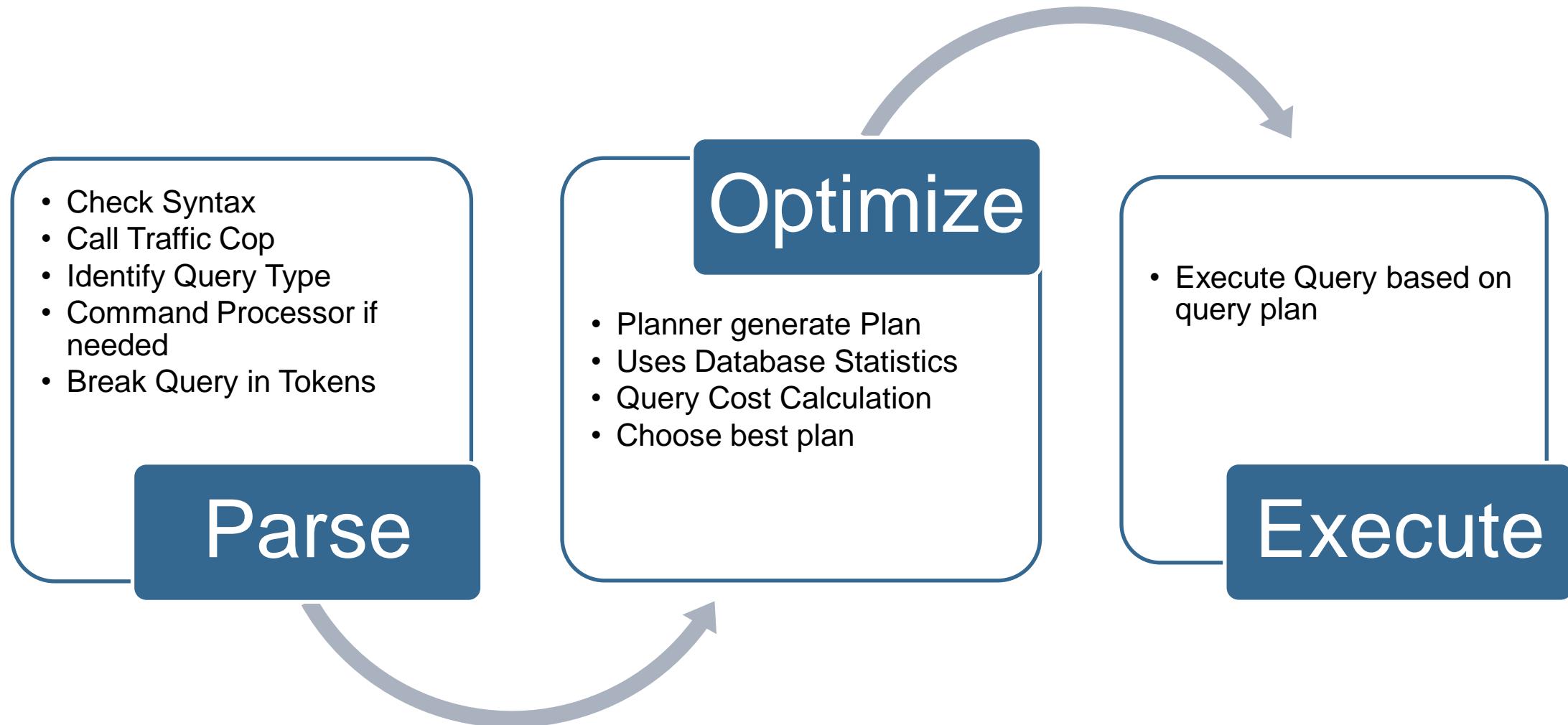
- Archiver spawns a task to copy away **pg_xlog** log files when full



Commit and Checkpoint

- Before commit
 - Uncommitted updates are in memory
- After commit
 - Committed updates written from shared memory to disk (write-ahead log file)
- After checkpoint
 - Modified data pages are written from shared memory to the data files

Statement Processing



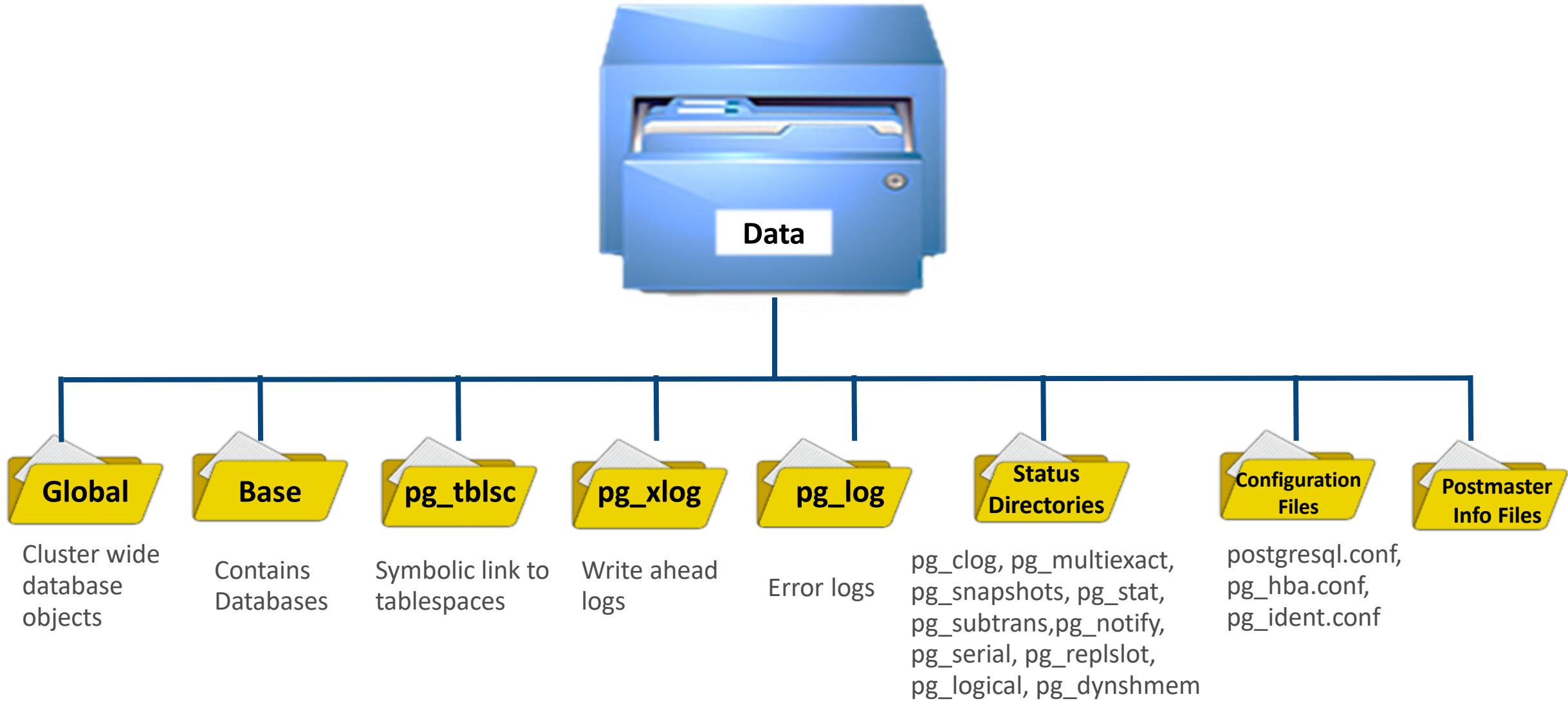
Physical Database Architecture

- A cluster is a collection of databases managed by a one server instance
- Each cluster has a separate
 - Data directory
 - TCP port
 - Set of processes
- A cluster can contain multiple databases

Installation Directory Layout

- **Default Installation Directory Location:**
 - **PostgreSQL** - /opt/PostgreSQL/9.6
 - **EDB Postgres Advanced Server** - /opt/PostgresPlus/9.5AS
- **bin** – Programs
- **data** – Data directory
- **doc** – Documentation
- **include** – Header files
- **installer, scripts** – Installer files
- **lib** – Libraries
- **pgAdmin 4** – Graphical administration tool
- **stackbuilder** (PostgreSQL) – Use to install additional components
- **stackbuilderplus** (Advanced Server) – installation directory for StackBuilder Plus
- **pg_env.bat/sh** – Set up environment

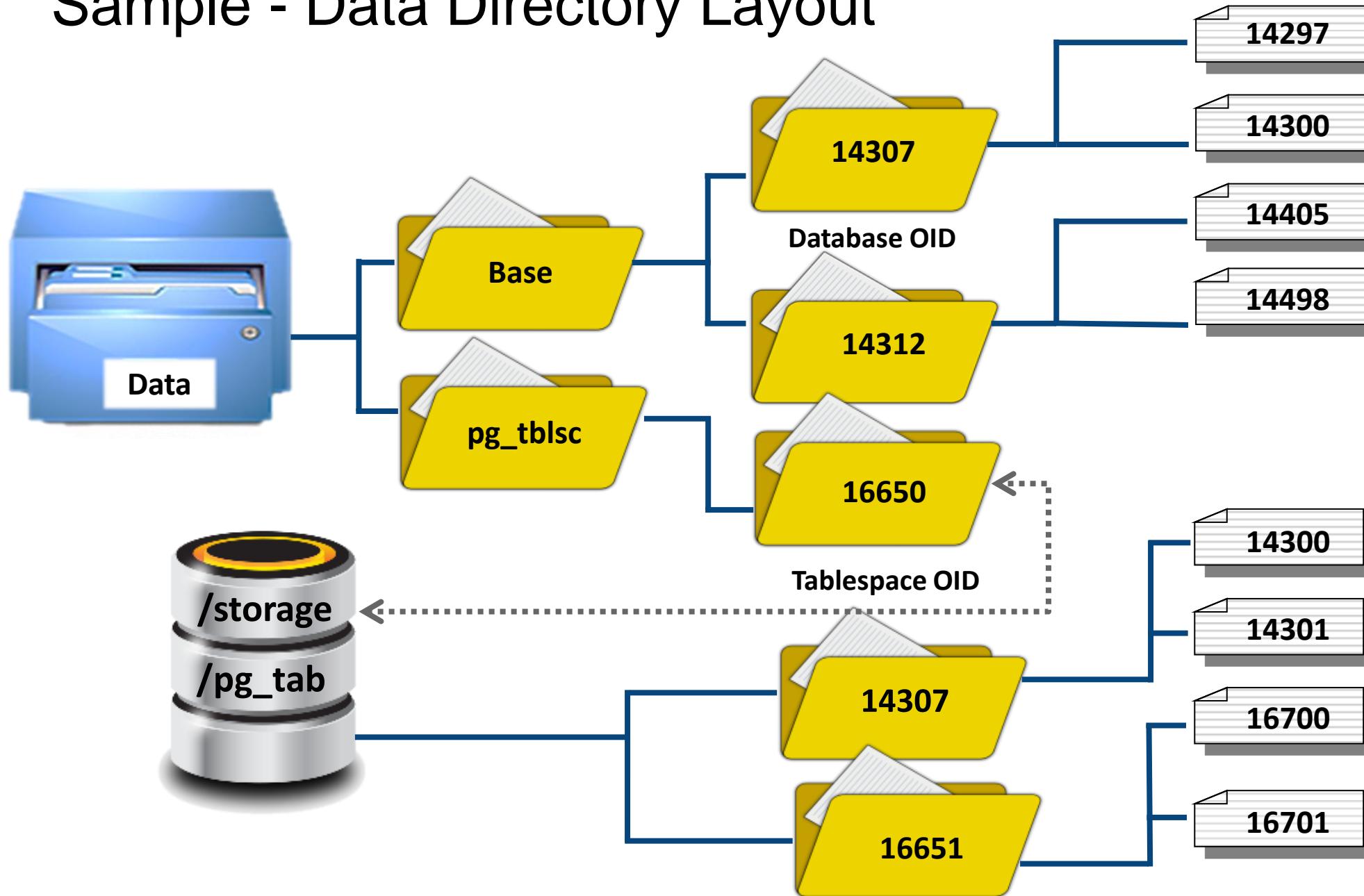
Database Cluster Data Directory Layout



Physical Database Architecture

- File-per-table, file-per-index
- A table-space is a directory
- Each database that uses that table-space gets a subdirectory
- Each relation using that table-space/database combination gets one or more files, in 1GB chunks
- Additional files used to hold auxiliary information (free space map, visibility map)
- Each file name is a number (see pg_class.relfilenode)

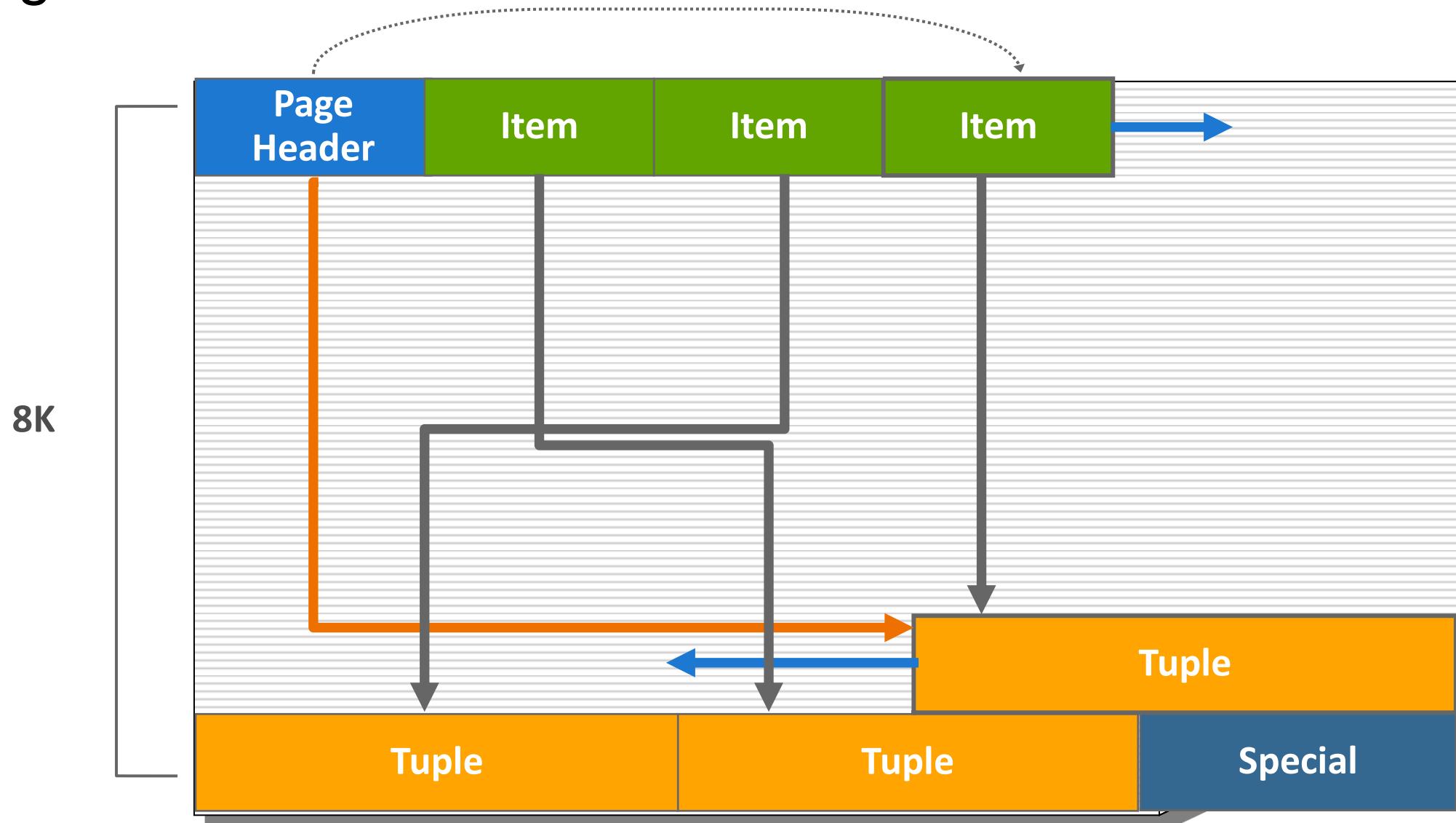
Sample - Data Directory Layout



Page Layout

- Page header
 - General information about the page
 - Pointers to free space
 - 24 bytes long
- Row/index pointers
 - Array of offset/length pairs pointing to the actual rows/index entries
 - 4 bytes per item
- Free space
 - Unallocated space
 - New pointers allocated from the front, new rows/index entries from the rear
- Row/index entry
 - The actual row or index entry data
- Special
 - Index access method specific data
 - Empty in ordinary tables

Page Structure



Module Summary

- Architectural Summary
- Shared Memory
- Inter-processes Communication
- Statement Processing
- Utility Processes
- Disk Read Buffering
- Disk Write Buffering
- Background Writer Cleaning Scan
- Commit and Checkpoint
- Physical Database Architecture
- Data Directory Layout
- Installation Directory Layout
- Page Layout

Module 3

PostgreSQL Installation

Module Objectives

- OS User and Permissions
- Installation Options
- Installation of PostgreSQL
- StackBuilder
- Setting Environmental Variables

OS User and Permissions

- PostgreSQL runs as a daemon (Unix / Linux) or service (Windows)
- The PostgreSQL GUI Installer needs superuser/admin access
- All processes and data files must be owned by a user in the OS
- During installation a `postgres` locked user will be created on Linux
- On windows a password is required

The postgres User Account

- It is advised to run PostgreSQL under a separate user account
- This user account should only own the data directory that is managed by the server
- The `useradd` or `adduser` Unix command can be used to add a user
- The user account named `postgres` is used throughout this training

Installation Options for PostgreSQL

- Wizard installer
- Operating system package
 - RPM/YUM
 - Debian/Ubuntu DEB
 - FreeBSD port
 - Solaris package
- Source code

Download the PostgreSQL Installer

- An interactive installation of PostgreSQL can be done using wizards
- Designed to make installation of PostgreSQL quick and easy
- pgAdmin 4 is also packaged in a one click installer
- Packaged and maintained by EnterpriseDB for the PostgreSQL community
- Download the installer from:
<http://www.enterprisedb.com/products-services-training/pgdownload>

Current Release		
PostgreSQL 9.6		
Recommended: Try EnterpriseDB tools by running Stackbuilder after installing PostgreSQL, expanding Trial Products and selecting Components under EnterpriseDB		
Windows-32		Download
Windows-64		Download
Linux x86-32		Download
Linux x86-64		Download
Mac		Download

PostgreSQL Installation Wizard - Welcome

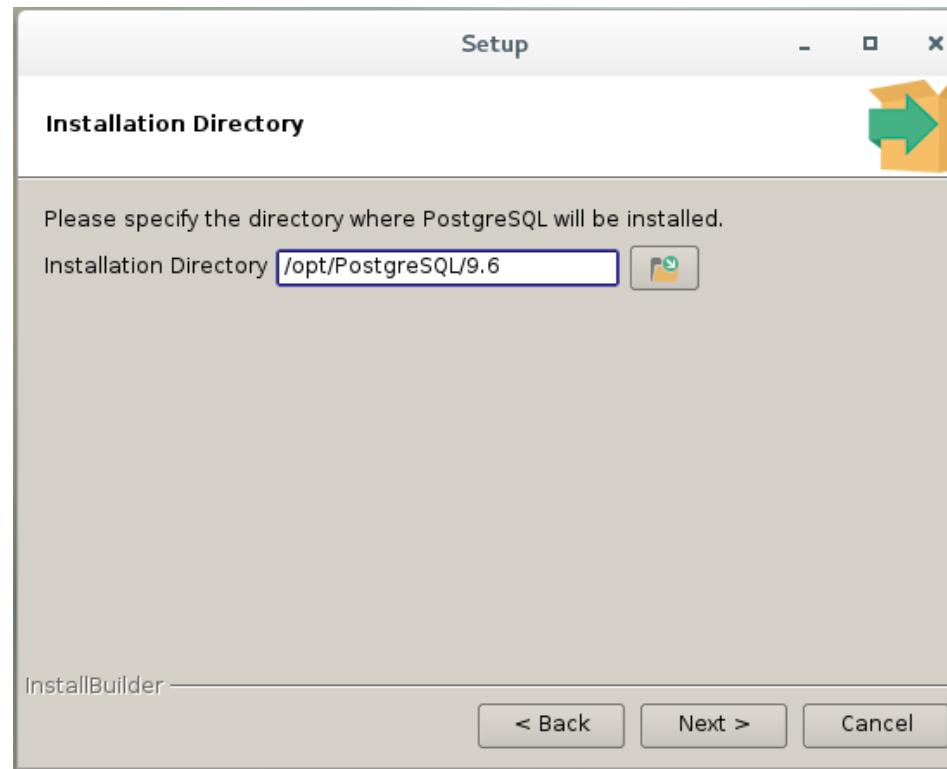
- Run the PostgreSQL binary using root or administrative user
- Linux/Unix systems require execute permission to run the downloaded binary
- Setup - Welcome message from the packaged installer

```
training@pgcentos:~/buildstore9.6
File Edit View Search Terminal Help
[training@pgcentos ~]$ cd buildstore9.6
[training@pgcentos buildstore9.6]$ chmod u+x postgresql-9.6.0-1-linux-x64.run
[training@pgcentos buildstore9.6]$ ls
postgresql-9.6.0-1-linux-x64.run
[training@pgcentos buildstore9.6]$ sudo useradd -p postgres postgres
[training@pgcentos buildstore9.6]$ sudo ./postgresql-9.6.0-1-linux-x64.run
```



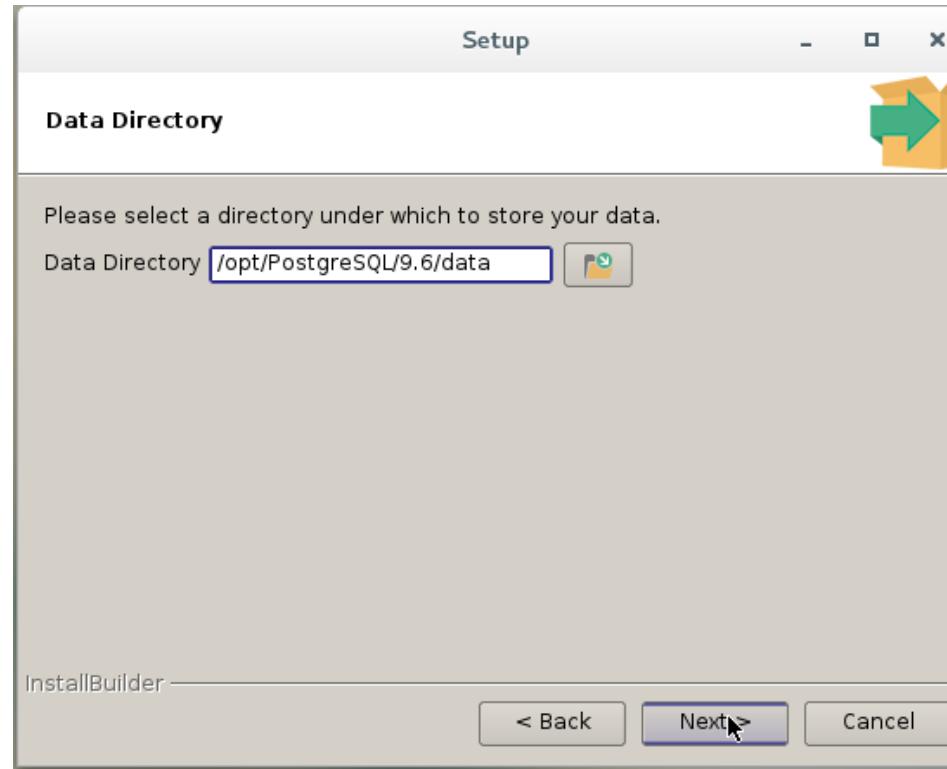
PostgreSQL Installation Wizard – Installation Directory

- Installation Directory - Choose the location where you want to install PostgreSQL



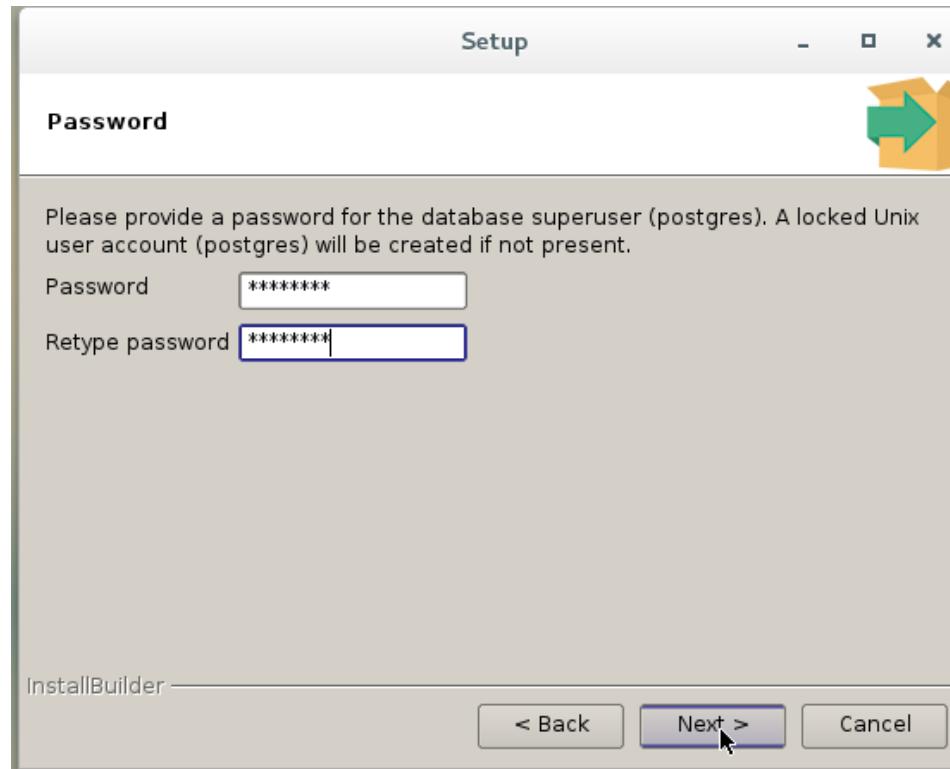
PostgreSQL Installation Wizard – Data Directory

- Data Directory - Choose the location where you want to install the default database cluster



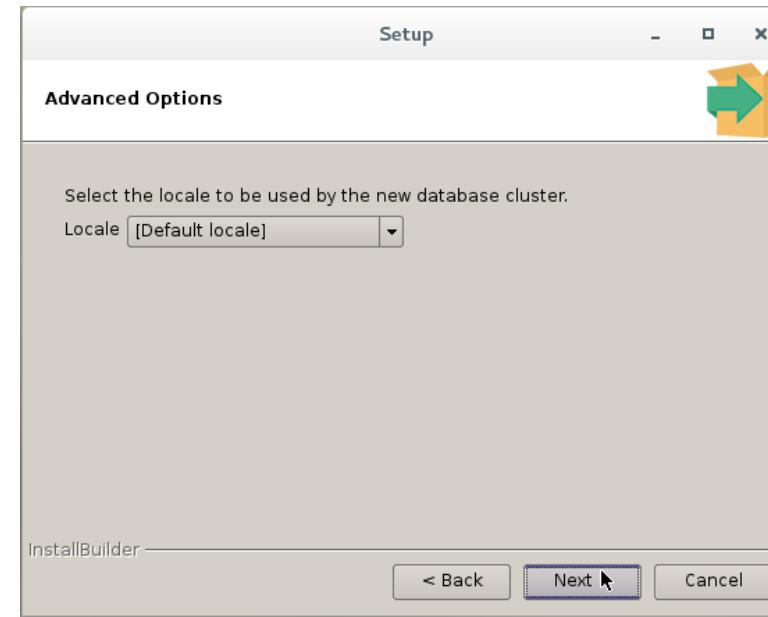
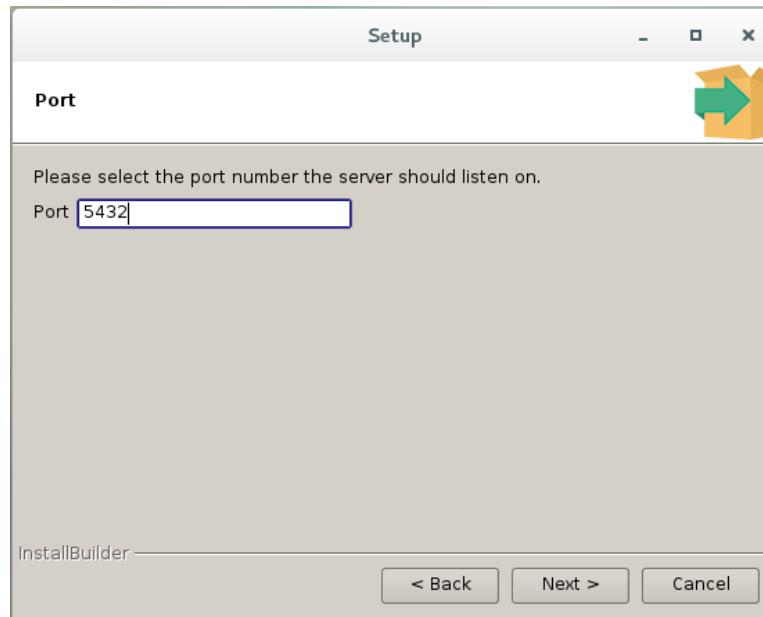
PostgreSQL Installation Wizard - Password

- Provide the database superuser and installation user password
- Read the instruction given in the snapshot

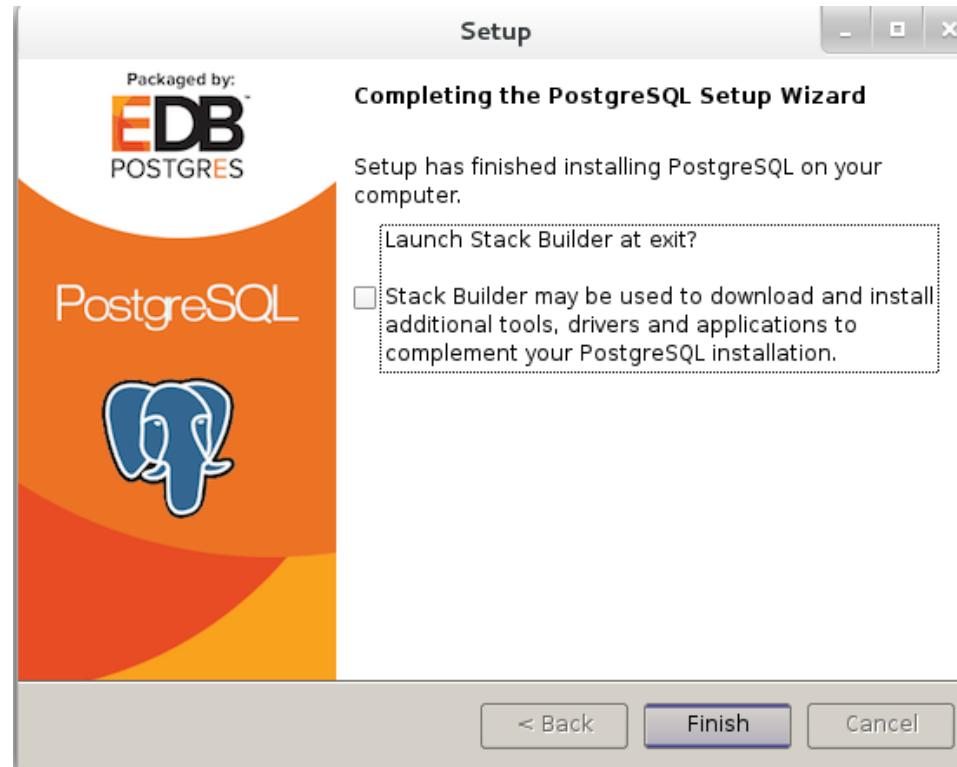


PostgreSQL Installation Wizard – Port and Locale

- Next specify the port number on which your default cluster will run
- The default port is 5432
- Then choose the Locale and click “Next” and install PostgreSQL

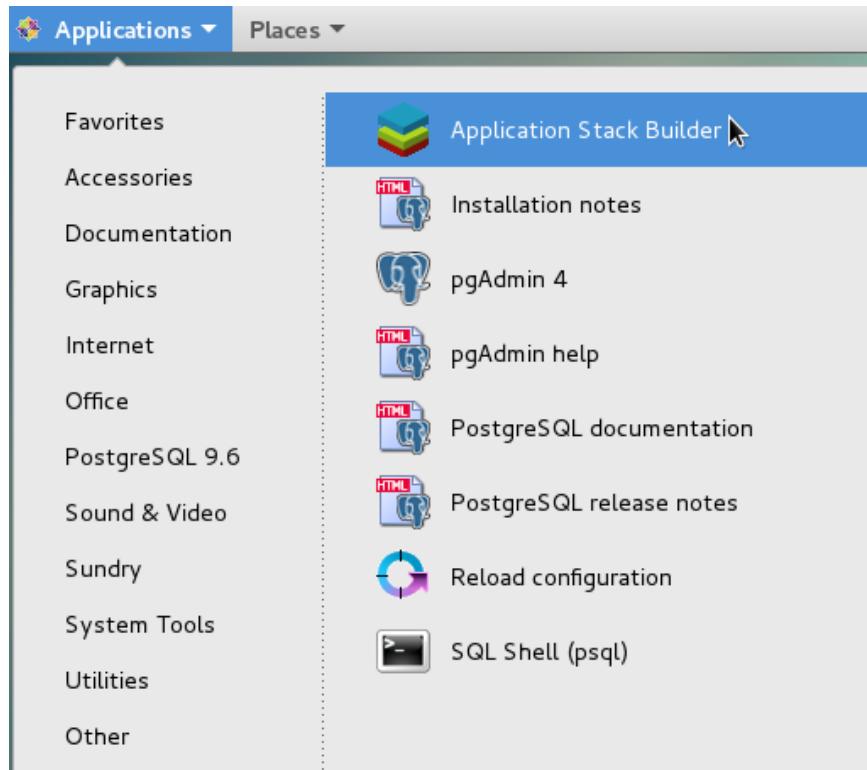


PostgreSQL Installation Wizard - Finish



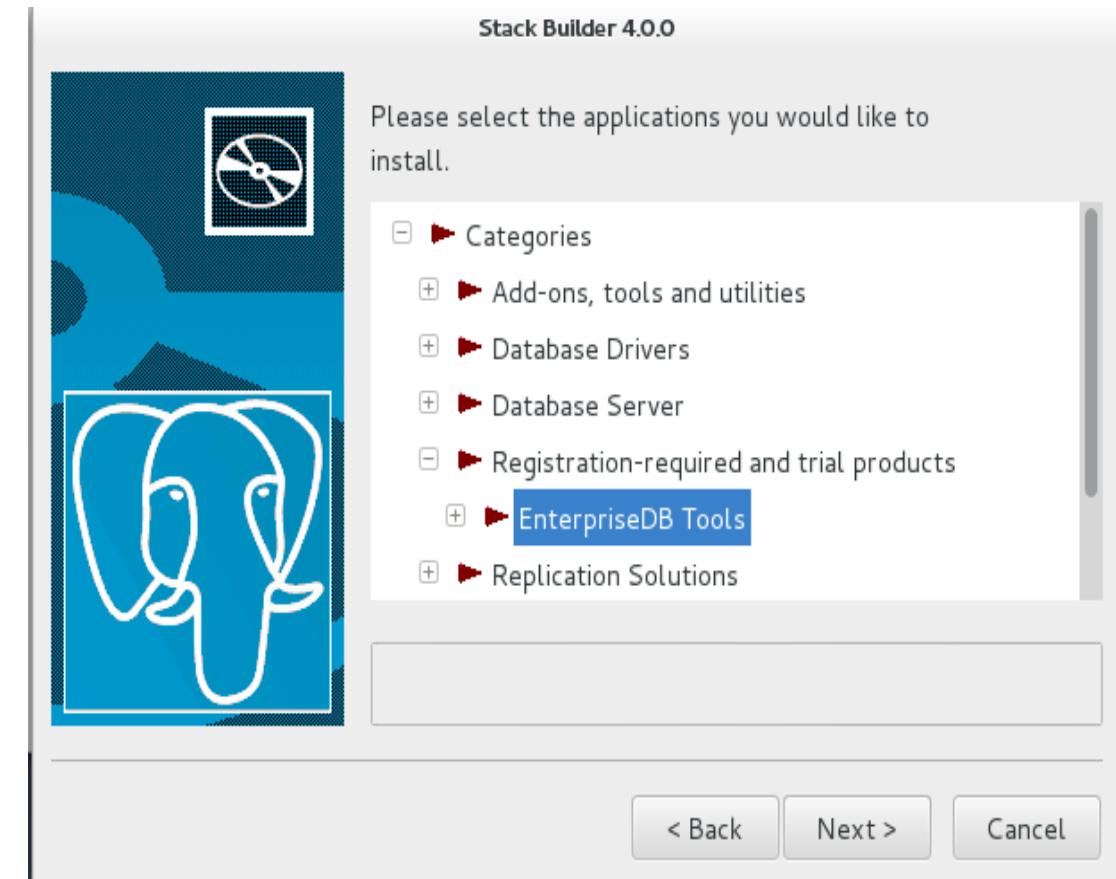
After PostgreSQL Installation

- Once installed you will get PostgreSQL running on the specified port
- You will also get a PostgreSQL menu to work with the database cluster



StackBuilder - EnterpriseDB Tools

- StackBuilder is an installation utility for add on modules and software that supports PostgreSQL databases
- Enhance PostgreSQL using powerful tools available from the EnterpriseDB Tools option of StackBuilder
- Tools from EnterpriseDB make your databases safer, faster and easier to manage



EnterpriseDB Tools

- **EDB Postgres Enterprise Manager (PEM)** - Intelligently manage, monitor, and tune multiple Postgres and EDB Postgres Advanced Server instances
- **EDB Postgres Replication Server** - Implement multi-master, single-master replication between different database
- **EDB Postgres Failover Manager** - Minimize downtime by automated failovers. Reduce false failovers with witness node
- **EDB Postgres Backup and Recovery** - Simple command line interface for backup, restore and managing backups
- **SQL/Protect** - Protects your database from SQL injections
- **EDB Postgres Migration Toolkit** - Fast, flexible and customized data migration tool
- **EDB Postgres Update Monitor** - Notifications for critical updates and patches

Setting Environmental Variables

- Setting environment variables is very important for trouble free startup/shutdown of the database server.
 - PATH - should point correct bin directory
 - PGDATA - should point to correct data cluster directory
 - PGPORT - should point correct port on which database cluster is running
 - PGUSER – specifies the default database user name
 - PGDATABASE – specify the default database
 - PGPASSWORD - specify default password
- Edit `.profile` or `.bash_profile` to set the variables.
- In Windows set these variables using my computer properties page.

Module Summary

- OS User and Permissions
- Installation Options
- Installation of PostgreSQL
- StackBuilder
- Setting Environmental Variables

Lab Exercise - 1

1. Choose the platform on which you want to install PostgreSQL
2. Download the PostgreSQL installer from the EnterpriseDB website for the chosen platform
3. Prepare the platform for installation
4. Install PostgreSQL
5. Connect to PostgreSQL using psql

Module 4

Database Clusters

Module Objectives

- Database Clusters
- Creating a Database Cluster
- Starting and Stopping the Server (pg_ctl)
- Connect to the Server Using psql

Database Clusters

- A Database Cluster is a collection of databases managed by a single server instance
- Database Clusters are comprised of:
 - Data directory
 - Port
- Default databases are created named:
 - template0
 - template1
 - postgres

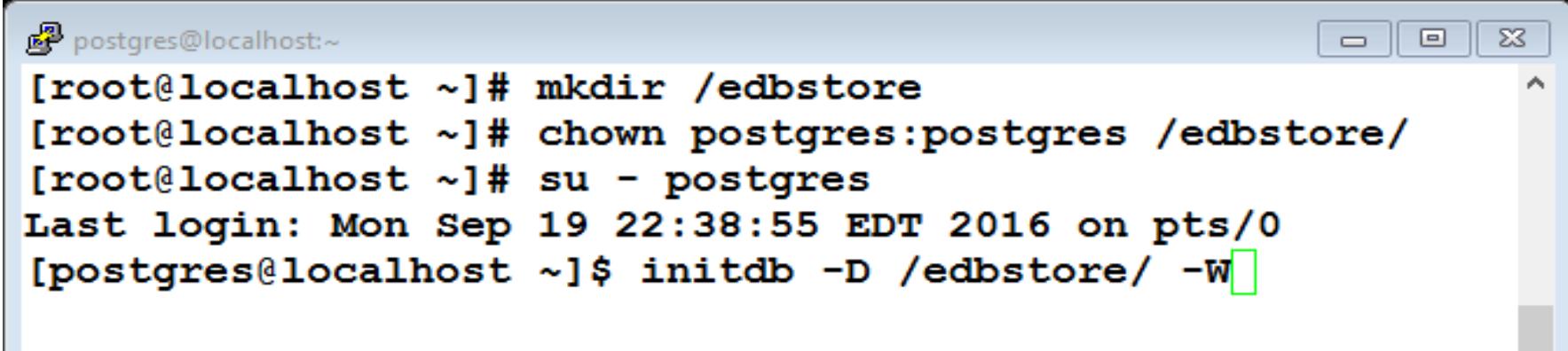
Creating a Database Cluster

- Choose the data directory location for new cluster
- Initialize the database cluster storage area (data directory) using `initdb` utility
- `initdb` will create the data directory if it doesn't exist
- You must have permissions on the parent directory so that `initdb` can create the data directory
- Data directory can be created manually using superuser access and ownership can be given to **postgres** user

initdb Utility

- `initdb [OPTION] ... [DATADIR]`
- Options
 - D, `--pgdata` location for this database cluster
 - E, `--encoding` set default encoding for new databases
 - U, `--username` database superuser name
 - W, `--pwprompt` prompt for a password for the new superuser
 - X, `--xlogdir` location for the transaction log directory
 - k, `--data-checksums` use data page checksums
 - V, `--version` output version information, then exit
 - A, `--auth` Default authentication method
 - ?, `--help` show this help, then exit
- If the data directory is not specified, the environment variable `PGDATA` is used

Example - initdb



A screenshot of a terminal window titled "postgres@localhost:~". The window contains the following command sequence:

```
[root@localhost ~]# mkdir /edbstore
[root@localhost ~]# chown postgres:postgres /edbstore/
[root@localhost ~]# su - postgres
Last login: Mon Sep 19 22:38:55 EDT 2016 on pts/0
[postgres@localhost ~]$ initdb -D /edbstore/ -W
```

The last command, "initdb -D /edbstore/ -W", is highlighted with a green rectangular selection.

- In the above example database system will be owned by user `postgres`
- `postgres` OS user must also own the server process
- `postgres` user is the database superuser
- The default server config file will be created in `/edbstore` named **`postgresql.conf`**
- `-W` is used to force `initdb` to prompt for the superuser password

Starting a Database Cluster

```
[postgres@localhost:~]$ vi /edbstore/postgresql.conf
                                # (change requires restart)
port = 5434                      # (change requires restart)
max_connections = 100              # (change requires restart)

[postgres@localhost:~]$ pg_ctl -D /edbstore/ -l startlog5432 start
server starting
[postgres@localhost:~]$
```

- Choose a unique port for postmaster
- Change the port in **postgresql.conf**
- pg_ctl utility can be used to start a cluster
 - Syntax
\$ pg_ctl start [options]
 - Options:
 - D location of the database storage area
 - l write (or append) server log to FILENAME
 - w wait until operation completes
 - t seconds to wait when using -w option

Connecting to a Database Cluster

- The `psql` and pgAdmin 4 clients can be used for connections
- The `edb-psql` and PEM clients can be used for EDB Postgres Advanced Server

```
postgres@localhost:~  
File Edit View Search Terminal Help  
[postgres@localhost ~]$ psql -p 5434 postgres postgres  
psql.bin (9.6.0)  
Type "help" for help.  
  
postgres=# show port;  
  port  
-----  
  5434  
(1 row)  
  
postgres=#
```

Reload a Database Cluster

- Some configuration parameter changes do not require a restart
- Changes can be reloaded using the `pg_ctl` utility
- Changes can also be reloaded using `pg_reload_conf()`
 - Syntax:
`$ pg_ctl reload [options]`
 - Options:
 - D location of the database storage area
 - s only print errors, no informational messages

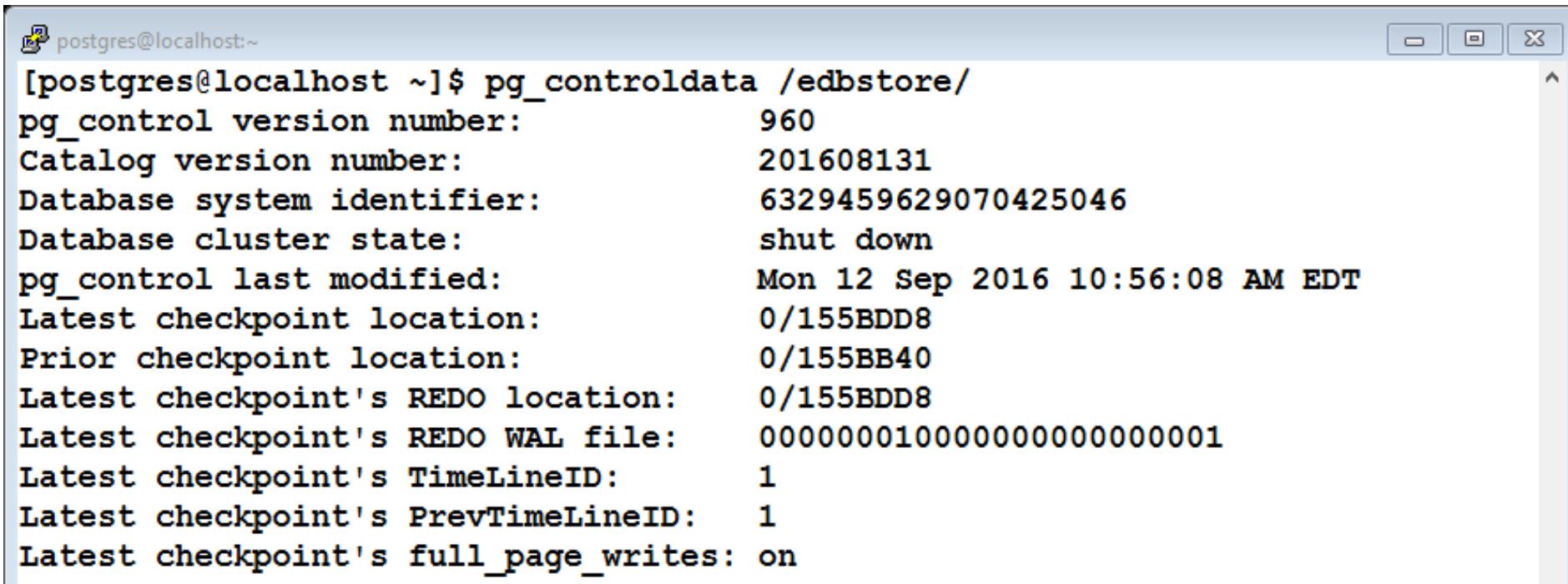
Stopping a Database Cluster

- pg_ctl can be used to stop a database cluster
- pg_ctl supports three modes of shutdown
 - smart quit after all clients have disconnected
 - fast (default) quit directly, with proper shutdown
 - immediate quit without complete shutdown; will lead to recovery
- Syntax:
 - pg_ctl stop [-W] [-t SECS] [-D DATADIR] [-s] [-m SHUTDOWN-MODE]

```
[postgres@edbtrain ~]$ pg_ctl -D /edbstore/ -mf stop
LOG:  received fast shutdown request
LOG:  aborting any active transactions
LOG:  autovacuum launcher shutting down
LOG:  shutting down
waiting for server to shut down....LOG:  database system is shut down
done
```

View Cluster Control Information

- `pg_controldata` can be used to view the control information for a database cluster
- Syntax:
 - `pg_controldata [DATADIR]`



A screenshot of a terminal window titled "postgres@localhost:~". The window displays the output of the command `pg_controldata /edbstore/`. The output is a series of key-value pairs:

Parameter	Value
pg_control version number:	960
Catalog version number:	201608131
Database system identifier:	6329459629070425046
Database cluster state:	shut down
pg_control last modified:	Mon 12 Sep 2016 10:56:08 AM EDT
Latest checkpoint location:	0/155BDD8
Prior checkpoint location:	0/155BB40
Latest checkpoint's REDO location:	0/155BDD8
Latest checkpoint's REDO WAL file:	000000010000000000000001
Latest checkpoint's TimeLineID:	1
Latest checkpoint's PrevTimeLineID:	1
Latest checkpoint's full_page_writes:	on

Module Summary

- Database Clusters
- Creating a Database Cluster
- Starting and Stopping the Server (pg_ctl)
- Connect to the Server Using psql

Lab Exercise - 1

1. A new website is to be developed for an online music store
 - Create a new cluster with data directory `/edbdata` and ownership of `postgres` user
 - Reload your cluster with `pg_ctl` utility and using `pg_reload_conf()` function
 - Stop your `edbdata` cluster with fast mode
 - Start your `edbdata` cluster

Module 5

Configuration

Module Objectives

- Setting PostgreSQL Parameters
- Access Control
- Connection Settings
- Security and Authentication Settings
- Memory Settings
- Query Planner Settings
- WAL Settings
- Log Management
- Background Writer Settings
- Statement Behavior
- Vacuum Cost Settings
- Autovacuum Settings
- Configuration File Includes

Setting PostgreSQL Parameters

- There are many configuration parameters that effect the behavior of the database system
- All parameter names are case-insensitive
- Every parameter takes a value of one of five types:
 - boolean
 - integer
 - floating point
 - string
 - enum
- One way to set these parameters is to edit the file **postgresql.conf**, which is normally kept in the data directory

The Server Parameter File - postgresql.conf

- Holds parameters used by a cluster
- Parameters are case-insensitive
- Normally stored in data directory
- initdb installs default copy
- Some parameters only take effect on server restart (`pg_ctl restart`)
- # used for comments
- One parameter per line
- Use include directive to read and process another file
- Can also be set using command-line option

Ways to Set PostgreSQL Parameters

- Some parameters can be changed per session using the `SET` command
- Some parameters can be changed at the user level using `ALTER USER`
- Some parameters can be changed at the database level using `ALTER DATABASE`
- The `SHOW` command can be used to see settings
- The `pg_settings` and `pg_file_settings` catalog table lists settings information

PostgreSQL ALTER SYSTEM

- ALTER SYSTEM command:

- Edits cluster settings without editing **postgresql.conf**
 - Writes the setting to a file called **postgresql.auto.conf**

- Example:

```
=# ALTER SYSTEM SET work_mem=20480;
```

- **postgresql.auto.conf** is always read last during server reload/restarts
- Reset a parameter change

```
=# ALTER SYSTEM SET work_mem = DEFAULT;
```

```
=# ALTER SYSTEM RESET ALL;
```

Connection Settings

- **listen_addresses** (default localhost) - Specifies the addresses on which the server is to listen for connections. Use * for all
- **port** (default 5432) - The port the server listens on
- **max_connections** (default 100) - Maximum number of concurrent connections the server can support
- **superuser_reserved_connections** (default 3) - Number of connection slots reserved for superusers
- **unix_socket_directory** (default /tmp) - Directory to be used for UNIX socket connections to the server
- **unix_socket_permissions** (default 0777) - access permissions of the Unix-domain socket

Security and Authentication Settings

- **authentication_timeout** (default 1 minute) - Maximum time to complete client authentication, in seconds
- **ssl** (default off) - Enables SSL connections
- **ssl_ca_file** - Specifies the name of the file containing the SSL server certificate authority (CA)
- **ssl_cert_file** - Specifies the name of the file containing the SSL server certificate
- **ssl_key_file** - Specifies the name of the file containing the SSL server private key
- **ssl_ciphers** - List of SSL ciphers that may be used for secure connections

Memory Settings

- **shared_buffers** (default 128MB) - Size of PostgreSQL shared buffer pool for a cluster
- **temp_buffers** (default 8MB) - Amount of memory used by each backend for caching temporary table data
- **work_mem** (default 4MB) - Amount of memory used for each sort or hash operation before switching to temporary disk files
- **maintenance_work_mem** (default 64MB) - Amount of memory used for each index build or VACUUM
- **temp_file_limit** (default 1) - amount of disk space that a session can use for temporary files. A transaction attempting to exceed this limit will be cancelled. Default is unlimited

Query Planner Settings

- **random_page_cost** (default 4.0) - Estimated cost of a random page fetch, in abstract cost units. May need to be reduced to account for caching effects
- **seq_page_cost** (default 1.0) - Estimated cost of a sequential page fetch, in abstract cost units. May need to be reduced to account for caching effects. Must always set $\text{random_page_cost} \geq \text{seq_page_cost}$
- **effective_cache_size** (default 4GB) - Used to estimate the cost of an index scan. Rule of thumb is 75% of system memory
- There are plenty of enable_* parameters which influence the planner in choosing an optimal plan

Write Ahead Log Settings

- **wal_level** (default: minimal) - Determines how much information is written to the WAL. Change this to enable replication. Other values are replica and logical
- **fsync** (default on) - Turn this off to make your database much faster – and silently cause arbitrary corruption in case of a system crash
- **wal_buffers** (default -1, autotune) - The amount of memory used in shared memory for WAL data. The default setting of -1 selects a size equal to 1/32nd (about 3%) of shared_buffers
- **min_wal_size** (default 80 MB) – The WAL size to start recycling the WAL files
- **max_wal_size** (default 1GB) – The WAL size to start checkpoint. Controls the number of WAL Segments(16MB each) after which checkpoint is forced
- **checkpoint_timeout** (default 5 minutes) - Maximum time between checkpoints
- **wal_compression** (default off) – The WAL of Full Page write will be compressed and written

Where To Log

- **log_destination** - Valid values are combinations of stderr, csvlog, syslog, and eventlog, depending on platform
- **logging_collector** - Enables advanced logging features. csvlog requires logging_collector
 - **log_directory** - Directory where log files are written. Requires logging_collector
 - **log_filename** - Format of log file name (e.g. postgresql-%Y-%M-%d.log). Allows regular log rotation. Requires logging_collector
 - **log_file_mode** (default 0600) - On Unix systems this parameter sets the permissions for log files when logging_collector is enabled
 - **log_rotation_age** - Automatically rotate logs after this much time. Requires logging_collector
 - **log_rotation_size** - Automatically rotate logs when they get this big. Requires logging_collector

When To Log

- **client_min_messages** (default NOTICE) - Messages of this severity level or above are sent to the client
- **log_min_messages** (default WARNING) - Messages of this severity level or above are sent to the server
- **log_min_error_statement** (default ERROR) - When a message of this severity or higher is written to the server log, the statement that caused it is logged along with it
- **log_min_duration_statement** (default -1, disabled) - When a statement runs for at least this long, it is written to the server log, with its duration

What To Log

- **log_connections** (default off) - Log successful connections to the server log
- **log_disconnections** (default off) - Log some information each time a session disconnects, including the duration of the session
- **log_error_verbosity** (default “default”) - Can also select “terse” or “verbose”
- **log_duration** (default off) - Log duration of each statement
- **log_line_prefix** - Additional details to log with each line
- **log_statement** (default none) - Legal values are none, ddl, mod (DDL and all other data-modifying statements), or all
- **log_temp_files** (default -1) - Log temporary files of this size or larger, in kilobytes
- **log_checkpoints** (default off) - Causes checkpoints and restartpoints to be logged in the server log

Background Writer Settings

- **bgwriter_delay** (default 200 ms) - Specifies time between activity rounds for the background writer
- **bgwriter_lru_maxpages** (default 100) - Maximum number of pages that the background writer may clean per activity round
- **bgwriter_lru_multiplier** (default 2.0) - Multiplier on buffers scanned per round. By default, if system thinks 10 pages will be needed, it cleans $10 * \text{bgwriter_lru_multiplier}$ of 2.0 = 20
- Primary tuning technique is to lower **bgwriter_delay**

Statement Behavior

- **search_path** - This parameter specifies the order in which schemas are searched. The default value for this parameter is "\$user", public
- **default_tablespace** - Name of the tablespace in which objects are created by default
- **temp_tablespaces** - Tablespaces name(s) in which temporary objects are created
- **statement_timeout** - Postgres will abort any statement that takes over the specified number of milliseconds A value of zero (the default) turns this off
- **idle_in_transaction_session_timeout** - Determines how long a session can be “idle in transaction” state

Parallel Query Scan Settings

- PostgreSQL supports parallel execution of read-only queries
- Can be enable and configures using configuration parameters
 - **max_parallel_workers_per_gather(default 0)** - Enables parallel query scan
 - **parallel_tuple_cost(default 0.1)** - Estimated cost of transferring one tuple from a parallel worker process to another process
 - **parallel_setup_cost(default 1000.0)** - Estimates cost of launching parallel worker processes
 - **min_parallel_relation_size(default 8MB)** - Relation larger than this parameter is considered for parallel scan
 - **force_parallel_mode(default off)** - Useful when testing parallel query scan even when there is no performance benefit

Vacuum Cost Settings

- **vacuum_cost_delay** (default 0 ms) - The length of time, in milliseconds, that the process will wait when the cost limit is exceeded
- **vacuum_cost_page_hit** (default 1) - The estimated cost of vacuuming a buffer found in the PostgreSQL buffer pool
- **vacuum_cost_page_miss** (default 10) - The estimated cost of vacuuming a buffer that must be read into the buffer pool
- **vacuum_cost_page_dirty** (default 20) - The estimated cost charged when vacuum modifies a buffer that was previously clean
- **vacuum_cost_limit** (default 200) - The accumulated cost that will cause the vacuuming process to sleep

Autovacuum Settings

- **autovacuum** (default on) - Controls whether the autovacuum launcher runs, and starts worker processes to vacuum and analyze tables
- **log_autovacuum_min_duration** (default -1) - Autovacuum tasks running longer than this duration are logged. Can now be specified per table
- **autovacuum_max_workers** (default 3) - Maximum number of autovacuum worker processes which may be running at one time
- **autovacuum_work_mem** (default -1, to use maintenance_work_mem) - Maximum amount of memory used by each autovacuum worker

Configuration File Includes

- The **postgresql.conf** file can now contain include directives
- Allows configuration file to be divided in separate files
- Usage in **postgresql.conf** file:
 - include ‘filename’
 - include_dir ‘directory name’

Module Summary

- Setting PostgreSQL Parameters
- Access Control
- Connection Settings
- Security and Authentication Settings
- Memory Settings
- Query Planner Settings
- WAL Settings
- Log Management
- Background Writer Settings
- Statement Behavior
- Vacuum Cost Settings
- Autovacuum Settings
- Configuration File Includes

Lab Exercise - 1

1. You are working as a DBA. It is recommended to keep a backup copy of the **postgresql.conf** file before making any changes. Make the necessary changes in the server parameter file for following settings:
 - So that the server allows up to 200 connected users
 - So the server should reserve 10 connection slots for DBA
 - Maximum time to complete client authentication will be 10 seconds

Lab Exercise - 2

1. Working as a DBA is a challenging job and to track down certain activities on the database server logging has to be implemented. Go through the server parameters that control logging and implement the following:
 - Save all the error message in a file inside the `pg_log` folder in your cluster data directory (for example `/edbdata`)
 - Log all queries which are taking more than 5 seconds to execute, and their time
 - Log the users who are connecting to the database cluster
 - Make the above changes and verify them

Lab Exercise - 3

1. Perform the following changes recommended by a senior DBA and verify them. Set:
 - Shared buffer to 256MB
 - Effective cache for indexes to 512MB
 - Maintenance memory to 64MB
 - Temporary memory to 8MB

Lab Exercise - 4

1. Vacuuming is an important maintenance activity and needs to be properly configured. Change the following **autovacuum** parameters on the production server. Set:
 - Autovacuum workers to 6
 - Autovacuum threshold to 100
 - Autovacuum scale factor to 0.3
 - Auto analyze threshold to 100
 - Autovacuum cost limit to 100

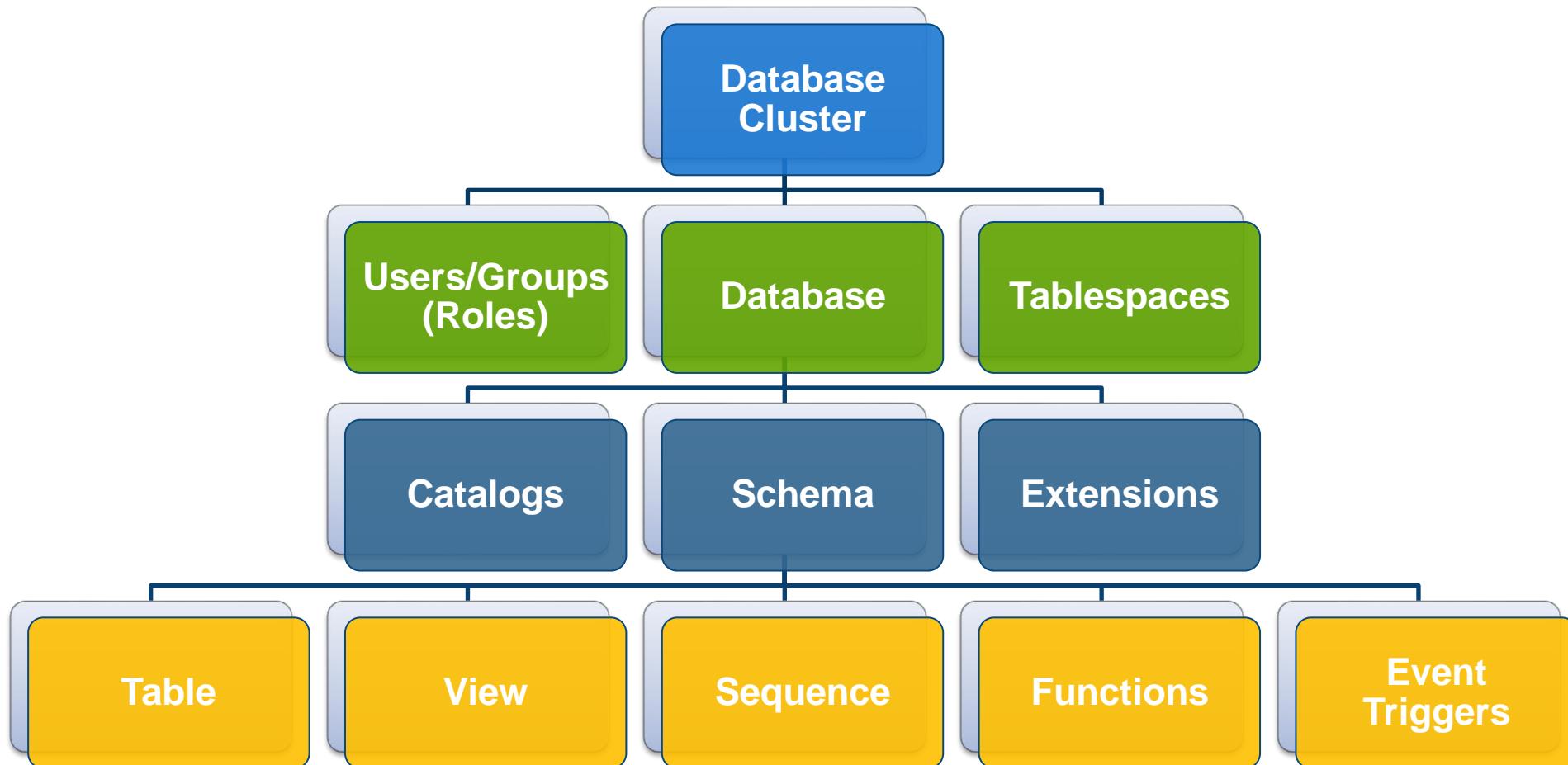
Module 6

Creating and Managing Databases

Module Objectives

- Object Hierarchy
- Creating Databases
- Users
- Access Control
- Creating Schemas
- Schema Search Path

Object Hierarchy



What is a Database

- A running PostgreSQL server can manage many databases
- A database is a named collection of SQL objects. It is a collection of schemas and the schemas contain the tables, functions, etc
- Databases are created with the `CREATE DATABASE` command
- Databases are destroyed with the `DROP DATABASE` command
- To determine the set of existing databases:
 - SQL - `SELECT datname FROM pg_database;`
 - psql META COMMAND - `\l` (backslash lowercase L)

Creating Databases

- Createdb is a program that executes from the shell to create new databases

```
$ createdb edbstore
```

- Create Database command can be used to create a database in a cluster.
 - Syntax:

```
=# CREATE DATABASE name  
[ [ WITH ] [ OWNER [=] user_name ]  
[ TEMPLATE [=] template ]  
[ ENCODING [=] encoding ]  
[ TABLESPACE [=] tablespace_name ]  
[ ALLOW_CONNECTIONS [=] allowconn ]  
[ CONNECTION LIMIT [=] connlimit ]
```

Example – Creating Databases

```
postgres=# CREATE DATABASE prod OWNER postgres;
CREATE DATABASE
postgres=# REVOKE CONNECT ON DATABASE prod FROM public;
REVOKE
postgres=# \connect prod
You are now connected to database "prod" as user "postgres".
prod=# SELECT datname FROM pg_database;
      datname
-----
template1
template0
postgres
prod
(4 rows)

prod=#
```

Accessing a Database

- The **psql** tool allows you to interactively enter, edit, and execute SQL commands
- The **pgAdmin 4** GUI tool can also be used to access a database
- To use **psql** to access a database:
 - Open the command prompt or terminal
 - If PATH is not set you can execute next command from the bin directory location of postgres installation:

```
$ psql -U postgres edbstore
```

```
edbstore=#
```

Users

- Database users are completely separate from operating system user
- Database users are global across a database cluster
- Username must be unique and must not start with `pg_`
- Every connection made to a database is made using a user
- `postgres` (PostgreSQL) or `enterprisedb` (Advanced Server) is a predefined superuser in default data cluster
- Pre-defined superuser name can be specified during initialization of the database cluster
- This pre-defined superuser have all the privileges with `GRANT OPTION`

Creating Users using psql or edb-psql

- Users can be added using CREATE USER sql command
- Syntax:

```
=# CREATE USER name [ [ WITH ] option [ ... ]
```

where option can be:

```
SUPERUSER | NOSUPERUSER | CREATEDB | NOCREATEDB |
CREATEROLE | NOCREATEROLE | INHERIT | NOINHERIT | LOGIN |
NOLOGIN | REPLICATION | NOREPLICATION | BYPASSRLS |
NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
```

- Example:

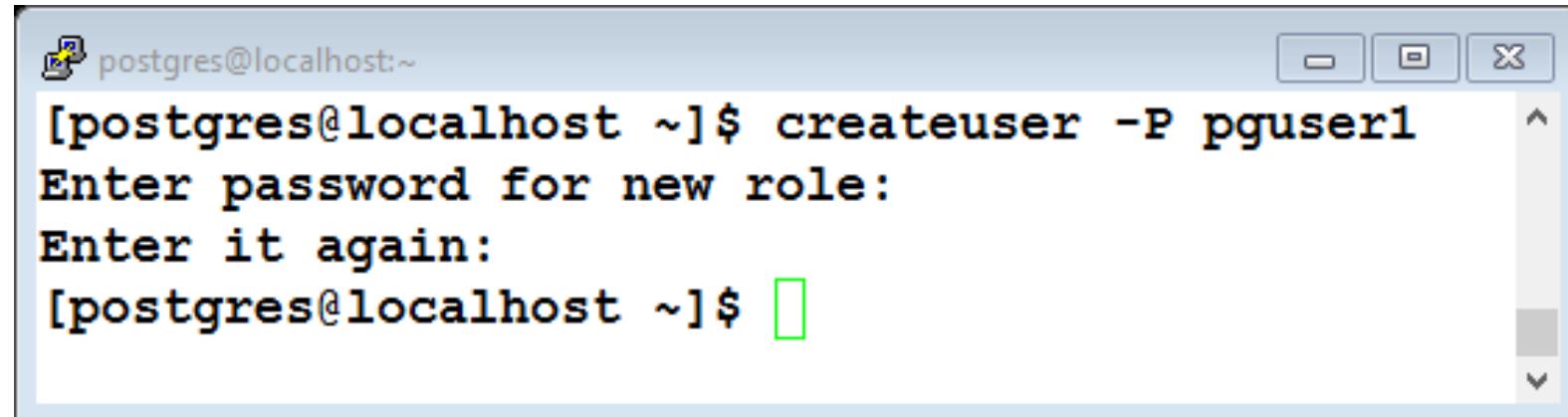
```
postgres=# CREATE USER rupi
postgres-# PASSWORD 'rupi123' SUPERUSER;
CREATE ROLE
postgres=#
```

Creating Users using EDB*Plus in EDB Postgres Advanced Server

```
enterprisedb@localhost:~  
File Edit View Search Terminal Help  
[enterprisedb@localhost ~]$ /opt/PostgresPlus/9.5AS/edbplus/edbplus.sh  
    User: enterprisedb  
Enter Password:  
Connected to EnterpriseDB 9.5.0.5 (localhost:5444/edb) AS enterprisedb  
  
EDB*Plus: Release 9.5 (Build 34.0.1)  
Copyright (c) 2008-2016, EnterpriseDB Corporation. All rights reserved.  
  
SQL> CREATE USER jack IDENTIFIED by jone;  
  
Role created.  
  
SQL> █
```

Creating Users Using createuser

- The `createuser` utility can also be used to create a user
- Syntax:
`$ createuser [OPTION]... [ROLENAME]`
- Use `--help` option to view the full list of options available
- Example:



The screenshot shows a terminal window titled "postgres@localhost:~". The command entered is `[postgres@localhost ~]$ createuser -P pguser1`. The terminal prompts for a password with "Enter password for new role:" and asks to "Enter it again:". A green cursor is visible at the end of the second password entry line.

```
postgres@localhost:~ [postgres@localhost ~]$ createuser -P pguser1
Enter password for new role:
Enter it again:
[postgres@localhost ~]$
```

Privileges

- Cluster level
 - Granted to a user during CREATE or later using ALTER USER
 - These privileges are granted by superuser
- Object Level
 - Granted to user using GRANT command
 - These privileges allow a user to perform particular action on an database object, such as table, view, sequence etc.
 - Can be granted by Owner, superuser or someone who have been given permission to grant privilege (WITH GRANT OPTION)

GRANT Statement

- GRANT can be used for granting object level privileges to database users, groups or roles
- Privileges can be granted on a tablespace, database, schema, table, sequence, domain and function
- GRANT is also used to grant a role to a user
- Syntax:
 - Type `\h GRANT` in psql terminal to view the entire syntax and available privileges that can be granted on different objects

Example – GRANT Statement

```
postgres=# GRANT CONNECT ON DATABASE postgres to pguser1;
GRANT
postgres=# GRANT USAGE ON SCHEMA public to pguser1;
GRANT
postgres=# GRANT SELECT,INSERT ON EMP TO pguser1;
GRANT
postgres=# \c postgres pguser1;
Password for user pguser1:
You are now connected to database "postgres" as user "pguser1".
postgres=> SELECT * FROM emp;
 id
-----
(0 rows)
```

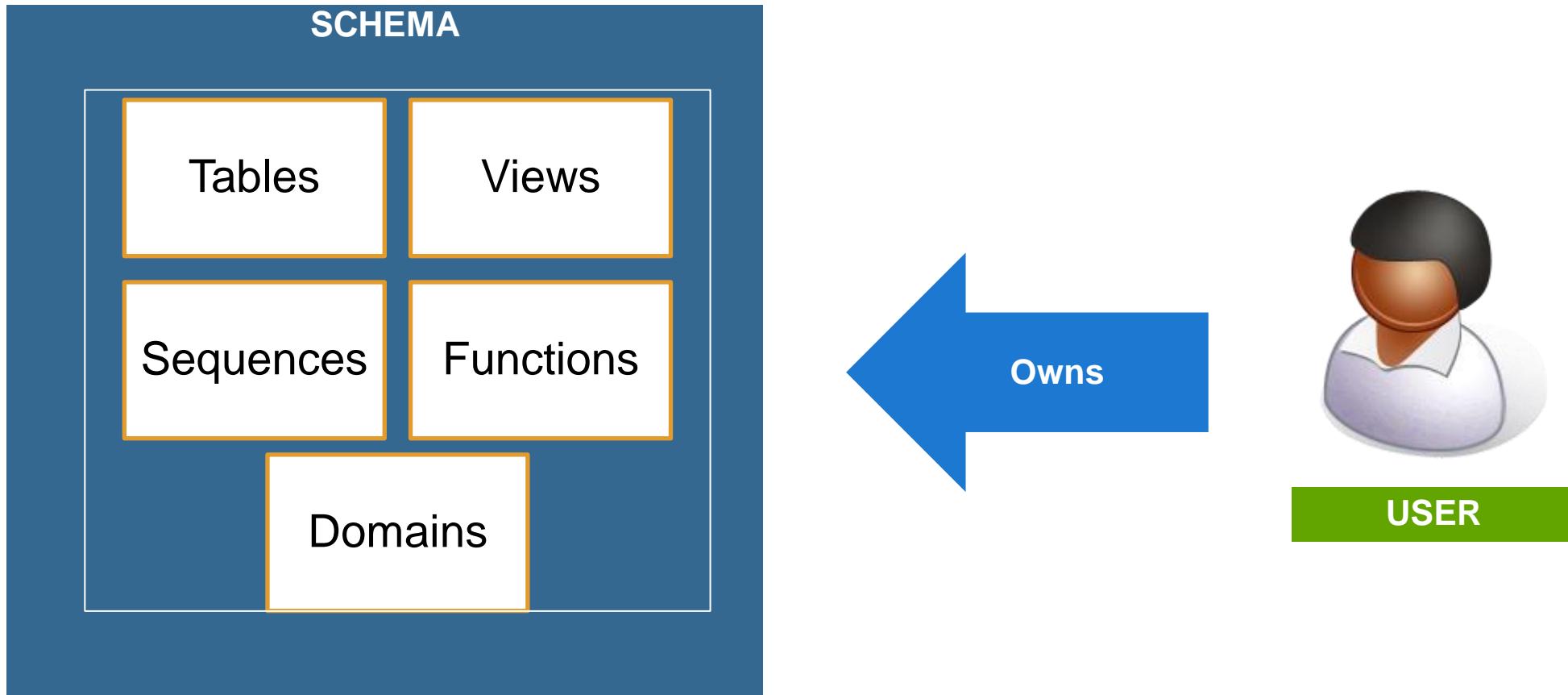
REVOKE Statement

- REVOKE can be used for revoking object level privileges to database users, groups or roles
- Privileges can be revoked on a tablespace, database, schema, table, sequence, domain and function
- REVOKE [GRANT OPTION FOR] can be used to revoke only the grant option without revoking the actual privilege
- Syntax:
 - Type \h REVOKE in **edb-psql** terminal to view the entire syntax and available privileges that can be revoked on different objects

Example - REVOKE Statement

```
postgres=# REVOKE SELECT,INSERT ON emp FROM pguser1;
REVOKE
postgres=# \c postgres pguser1;
Password for user pguser1:
You are now connected to database "postgres" as user "pguser1".
postgres=> SELECT * FROM emp;
ERROR: permission denied for relation emp
postgres=> \c postgres postgres
Password for user postgres:
You are now connected to database "postgres" as user "postgres".
postgres=# REVOKE CONNECT ON DATABASE postgres FROM pguser1;
REVOKE
postgres=# REVOKE CONNECT ON DATABASE postgres FROM public;
REVOKE
postgres=# \c postgres pguser1;
Password for user pguser1:
FATAL: permission denied for database "postgres"
DETAIL: User does not have CONNECT privilege.
Previous connection kept
postgres=# █
```

What is a Schema



Benefits of Schemas

- A database can contains one or more named schemas
- By default, all databases contain a public schema
- There are several reasons why one might want to use schemas:
 - To allow many users to use one database without interfering with each other
 - To organize database objects into logical groups to make them more manageable
 - Third-party applications can be put into separate schemas so they cannot collide with the names of other objects

Creating Schemas

- Schemas can be added using CREATE SCHEMA sql command
- Syntax:

```
=# CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION  
role_specification ]
```

- Example:

```
prod=# CREATE SCHEMA rupi AUTHORIZATION rupi;  
CREATE SCHEMA  
prod=# GRANT CONNECT ON DATABASE prod to rupi;  
GRANT  
prod=# \c prod rupi  
Password for user rupi:  
You are now connected to database "prod" as user "rupi".  
prod=> CREATE TABLE city(cityname VARCHAR2);  
CREATE TABLE  
prod=> \dt  
      List of relations  
 Schema | Name | Type  | Owner  
-----+-----+-----+-----  
    rupi | city | table | rupi  
(1 row)  
  
prod=> █
```

What is a Schema Search Path

- Qualified names are tedious to write, so we use table names directly in queries
- If no schema name is given, the schema search path determines which schemas are searched for matching table names
 - Example:

```
=> SELECT * FROM employee;
```

 - This statement will find the first employee table from the schemas listed in the search path

Determine the Schema Search Path

- The first schema named in the search path is called the current schema if that named schema exists
- Aside from being the first schema searched, it is also the schema in which new tables will be created if the `CREATE TABLE` command does not specify a schema name
- To show the current search path, use the following command:
 => `SHOW search_path;`
- To put our new schema in the path, we use:
 => `SET search_path TO myschema, public;`

Schema Search Path and Users

- The first element specifies that a schema with the same name as the current user is to be searched. If no such schema exists, the entry is ignored.
- The second element refers to the public schema that we have seen already. To put our new schema in the path, we use:

```
=> SET search_path TO  
myschema, public;
```

In Default Setup

```
postgres=# show search_path;  
search_path  
-----  
"$user",public  
(1 row)  
postgres=#
```

Set the Schema Search Path

- To put our new schema in the path, we use:

```
=> SET search_path TO myschema, public;
```

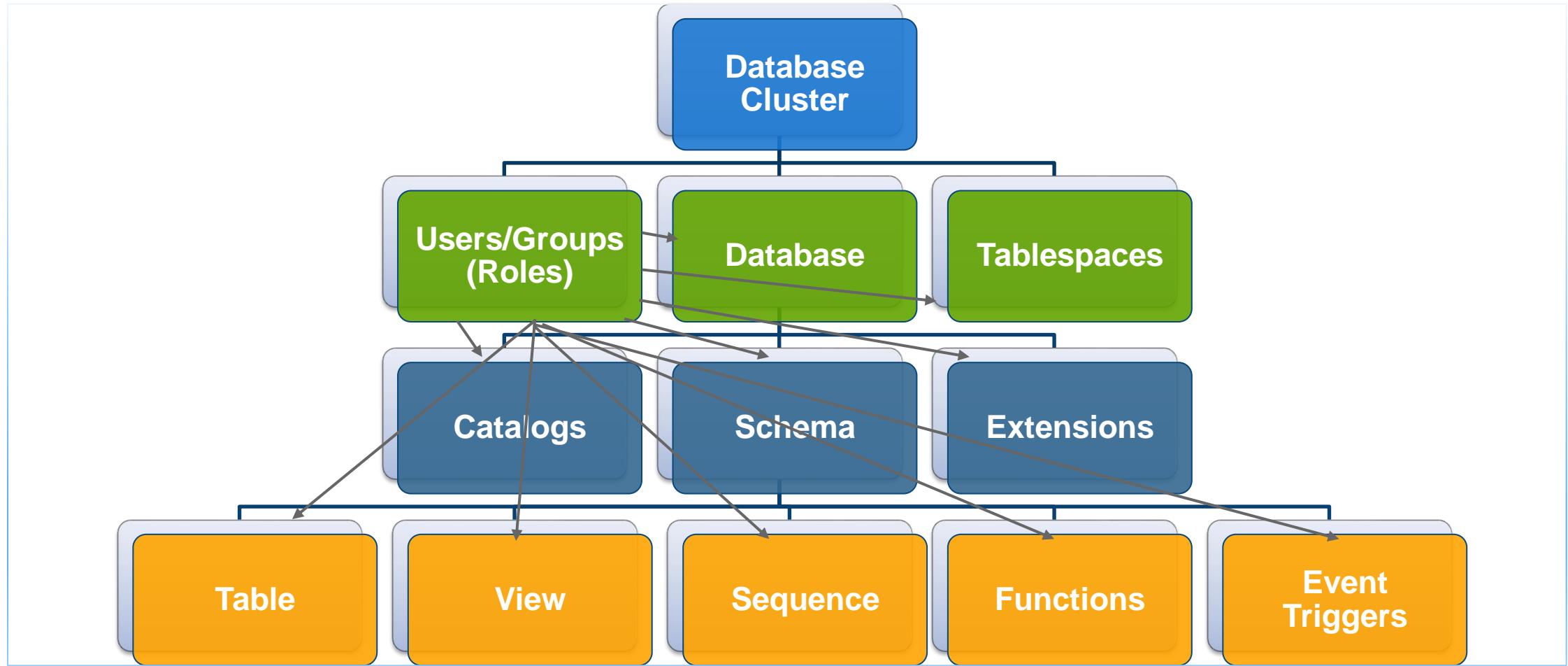
**cust table created in
myschema**

**Reset search_path so
myschema
No longer visible**

```
postgres=# set search_path TO myschema,public;
SET
postgres=# create table cust(id integer, name varchar(10));
CREATE TABLE
postgres=# \d
          List of relations
 Schema |      Name       | Type | Owner
-----+-----+-----+-----+
 myschema | cust         | table | postgres
 public   | consultant    | table | postgres
 public   | consultant_id_sequence | sequence | postgres
(3 rows)

postgres=# set search_path to "$user",public;
SET
postgres=# \d
          List of relations
 Schema |      Name       | Type | Owner
-----+-----+-----+-----+
 public | consultant    | table | postgres
 public | consultant_id_sequence | sequence | postgres
(2 rows)
```

Object Ownership



Module Summary

- Object Hierarchy
- Creating Databases
- Users
- Access Control
- Creating Schemas
- Schema Search Path

Lab Exercise - 1

1. A new website is to be developed for an online music store.
 - Create a database user `edbuser` in your existing cluster
 - Create an `edbstore` database with ownership of `edbuser` user
 - Login to the `edbstore` database using the `edbuser` user and create the `edbuser` schema
 - Logoff from `psql`

Lab Exercise - 2

1. An e-music online store website application developer wants to add an online buy/sell facility and has asked you to separate all tables used in online transactions. Here you have suggested to use schemas.
Implement the following suggested options:
 - Create an ebuy user with password 'lion'
 - Create an ebuy schema which can be used by user ebuy
 - Login as the ebuy user, create a table sample1 and check whether that table belongs to the ebuy schema or not

Lab Exercise - 3

1. EnterpriseDB provided an **edbstore.sql** file with the training material and this script file can be installed in the newly created `edbstore` database
 - Download **edbstore.sql** and place it in a directory which is accessible to the `postgres` user. Make sure the file is also owned by the `postgres` user
 - Run the `psql` command with the `-f` option to execute the **edbstore.sql** file and install all the sample objects required for this training

Lab Exercise - 4

1. Retrieve a list of databases using a SQL query
2. Retrieve a list of databases using the psql meta command
3. Retrieve a list of tables in the `edbstore` database and check which schema and owner they have

Module 7

User Tools - Command Line Interfaces

Module Objectives

- Introduction to psql and edb-psql
- Conventions
- Connecting to Database
- psql Command Line Parameters
- Entering psql Commands
- psql Meta-Commands
- psql SET Parameters
- Information Commands

Introduction to psql

- **psql** is a command line interface (CLI) to PostgreSQL
- Can be used to execute SQL queries and **psql** meta commands

```
postgres@localhost:~  
File Edit View Search Terminal Help  
[postgres@localhost ~]$ which psql  
/opt/PostgreSQL/9.6/bin/psql  
[postgres@localhost ~]$ psql  
Password:  
psql.bin (9.6.0)  
Type "help" for help.  
  
postgres=#
```

Connecting to a Database

- In order to connect to a database you need to know the name of your target database, the host name and port number of the server and the user name
- Command line options, namely `-d`, `-h`, `-p`, and `-U` respectively.
- Environment variables `PGDATABASE`, `PGHOST`, `PGPORT` and `PGUSER` to appropriate values

Conventions

- **psql** has its own set of commands, all of which start with a backslash (\). These are in no way related to SQL commands, which operate in the server. **psql** commands only affect psql
- Some commands accept a pattern. This pattern is a modified regex. Key points:
 - * and ? are wildcards
 - Double-quotes are used to specify an exact name, ignoring all special characters and preserving case

On Startup...

- **psql** will execute commands from `$HOME/.psqlrc`, unless option `-x` is specified
- `-f FILENAME` will execute the commands in `FILENAME`, then exit
- `-c COMMAND` will execute `COMMAND` (SQL or internal) and then exit
- `--help` will display all the startup options, then exit
- `--version` will display version info and then exit

Entering Commands

- **psql** uses the command line editing capabilities that are available in the native OS. Generally, this means
 - Up and Down arrows cycle through command history
 - On UNIX, there is tab completion for various things, such as SQL commands
 - `\s` will show the command history
 - `\s FILENAME` will save the command history
 - `\e` will edit the query buffer and then execute it
 - `\e FILENAME` will edit FILENAME and then execute it
 - `\w FILENAME` will save the query buffer to FILENAME

Controlling Output

- There are numerous ways to control output
- The most important are:
 - `-o FILENAME` or `\o FILENAME` will send query output (excluding STDERR) to `FILENAME` (which may be a pipe)
 - `\g FILENAME` executes the query buffer sending output to `FILENAME` (may be a pipe)
 - `\watch` can be used to run previous query repeatedly
 - `-q` runs quietly

Advanced Features - Variables

- **psql** provides variable substitution
- Variables are simply name/value pairs
- Use **\set** meta command to set a variable

```
=# \set city Edmonton
```

```
=# \echo :city
```

```
Edmonton
```

- Use **\unset** to delete a variable

```
=# \unset city
```

Advanced Features - Special Variables

- Settings can be changed at runtime by altering special variables
- Some important special variables:
 - AUTOCOMMIT
 - ENCODING
 - HISTFILE
 - ON_ERROR_ROLLBACK
 - ON_ERROR_STOP
 - PROMPT1
 - VERBOSITY
- Example:

```
=# \set AUTOCOMMIT off
```

 - Once AUTOCOMMIT is set to off use COMMIT/ROLLBACK to complete the running transaction

Information Commands

- `\d(i, s, t, v, b, S) [+]` [pattern]
 - List information about indexes, sequences, tables, views ,tablespaces or System objects. Any combination of letters may be used in any order, for example - `\dvs`
 - + displays comments
- `\d[+]` [pattern]
 - For each relation describe/display the relation structure details
 - + displays any comments associated with the columns of the table, and if the table has an OID column
 - Without a pattern, `\d[+]` is equivalent to `\dtvs [+]`

Information Commands (continued)

- `\l [ist] [+]`
 - Lists the names, owners, and character set encodings of all the databases in the server
 - If + is appended to the command name, database descriptions are also displayed
- `\dn+ [pattern]`
 - Lists schemas (namespaces)
 - + adds permissions and description to output
- `\df [+] [pattern]`
 - Lists functions
 - + adds owner, language, source code and description to output

Other Common psql Meta Commands

- \conninfo
 - Current connection information
- \q or ^d
 - Quits the psql program
- \cd [directory]
 - Change current working directory
 - Tip - To print your current working directory, use \! pwd
- \! [command]
 - Executes the specified command
 - If no command is specified, escapes to a separate Unix shell (CMD.EXE in Windows)

Help

- `\?`
 - Shows help information about psql commands
- `\h [command]`
 - Shows information about SQL commands
 - If command isn't specified, lists all SQL commands
- `psql --help`
 - Lists command line options for psql

Introduction to edb-psql

- **edb-psql** is the name of EnterpriseDB psql's executable
- EnterpriseDB **edb-psql** provides a terminal-based front-end to Advanced Server
- It enables you to type in queries interactively, issue them to EnterpriseDB, and see the query results

```
$ edb-psql [option...] [dbname [username]]
```

EDB*Plus

- EDB*Plus is a command line user interface to the EDB Postgres Advanced Server
- EDB*Plus accepts SQL commands, SPL anonymous blocks, and EDB*Plus commands
- EDB*Plus commands are compatible with Oracle SQL*Plus commands

```
[enterprisedb@localhost ~]$ cd /opt/PostgresPlus/9.5AS/edbplus/  
[enterprisedb@localhost edbplus]$ ./edbplus.sh  
enterprisedb/edb@localhost:5db  
Connected to EnterpriseDB 9.5.0.1 (localhost:5444/edb) AS enterprisedb
```

EDB*Plus: Release 9.5 (Build 33.0.0)
Copyright (c) 2008-2014, EnterpriseDB Corporation. All rights reserved.

SQL>

Module Summary

- Introduction to PSQL and EDB-PSQL
- Conventions
- Connecting to Database
- PSQL Command Line Parameters
- Entering PSQL Commands
- PSQL Meta-Commands
- PSQL SET Parameters
- Information Commands

Lab Exercise – 1

1. Connect to a database using **psql**
2. Switch databases
3. Describe the `customers` table
4. Describe the `customers` table including description
5. List all databases
6. List all schemas
7. List all tablespaces
8. Execute a sql statement, saving the output to a file
9. Do the same thing, just saving data, not the column headers
10. Create a script via another method, and execute from **psql**
11. Turn on the expanded table formatting mode
12. Lists tables, views and sequences with their associated access privileges.
Which meta command displays the SQL text for a function?
13. View the current working directory

Module 8

GUI Tools

Module Objectives

- Introduction to pgAdmin 4
- Registering a server
- Viewing and Editing Data
- Query Tool
- Databases
- Languages
- Schemas
- Database Objects
- Maintenance
- Tablespaces
- Roles
- Introduction to PEM Client

Introduction to pgAdmin 4

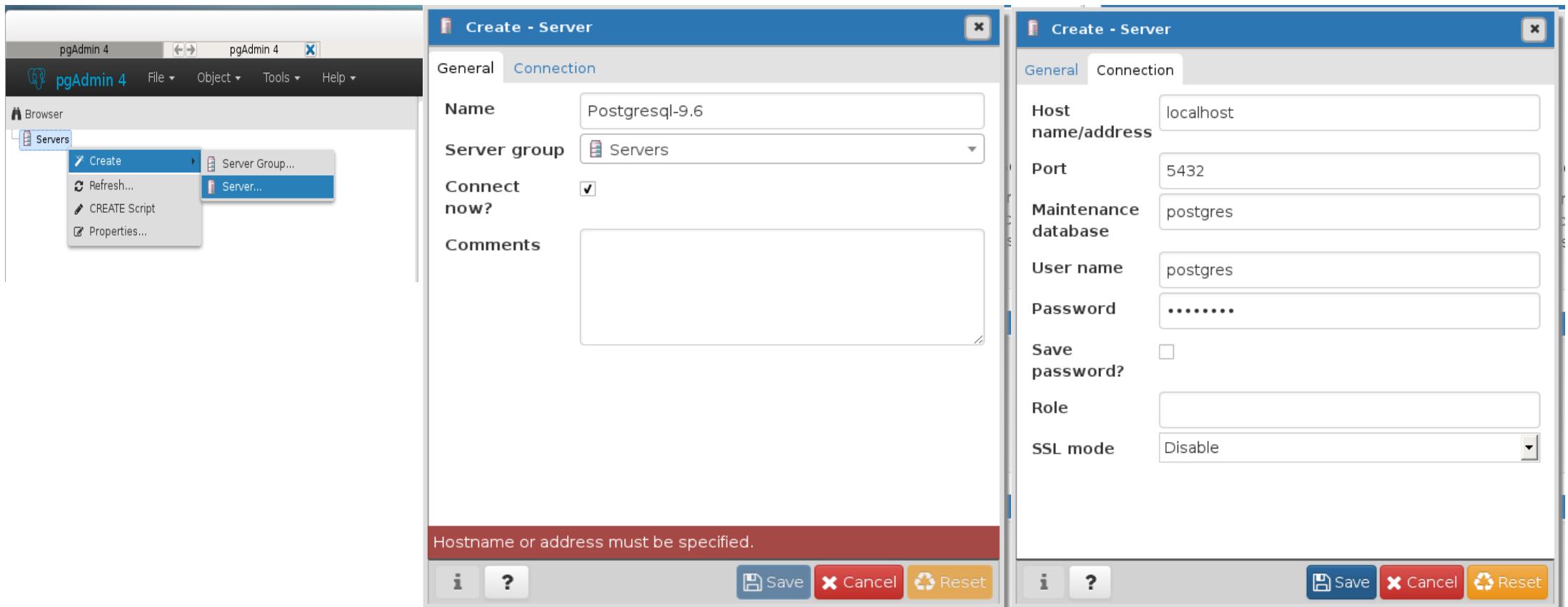
The screenshot shows the pgAdmin 4 interface with the following components:

- Toolbar:** pgAdmin 4, File, Object, Tools, Help.
- Sidebar (Browser):** Servers (1) - Postgresql-9.6, Databases, Login/Group Roles, Tablespaces.
- Dashboard:** A collection of monitoring charts:
 - Server sessions:** Shows Active, Total, and Idle session counts. Active is at 1.00.
 - Transactions per second:** Shows Commits, Transactions, and Rollbacks. Transactions fluctuate between 4.0 and 10.0.
 - Tuples in:** Shows Inserts, Deletes, and Updates. Deletes are near 1.00.
 - Tuples out:** Shows Fetched and Returned tuples. Returns spike to over 1000.
 - Block I/O:** Shows Reads and Hits. Both spike to over 70.
- Server activity:** A table showing session details.

Process ID	Database	User	Application name	Client address	Backend start	State
30617	postgres	postgres	pgAdmin 4 - DB:postgres	::1	2016-09-22 17:17:09.189258+05:30	active

Registering a Server

- Right Click on the server to add a server



Common Connection Problems

- There are 2 common error messages that you encounter while connecting to a PostgreSQL database:

Could not connect to Server - Connection refused

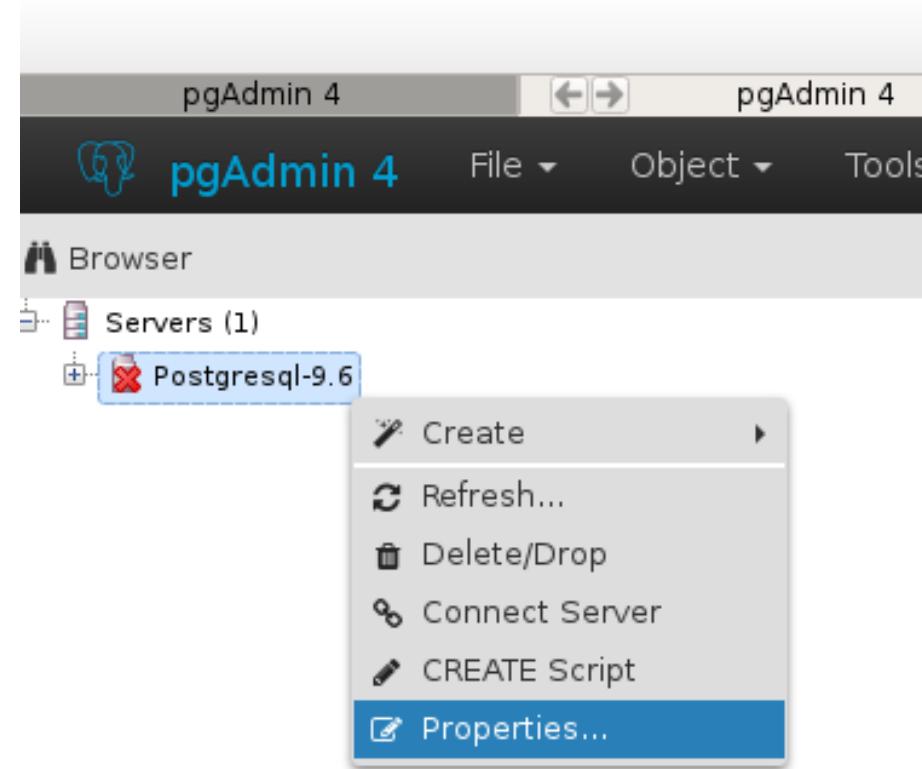
- This error occurs when either the database server isn't running OR the port 5432 may not be open on database server to accept external TCP/IP connections.

FATAL: no pg_hba.conf entry

- This means your server can be contacted over the network, but is not configured to accept the connection. Your client is not detected as a legal user for the database. You will have to add an entry for each of your clients to the **pg_hba.conf** file.

Changing a Server's Registration

- Right-click on a server entry and disconnect first to modify its properties
- Click on the properties to do changes to server's entry



Viewing Data

- Right-click on a table and select View Data

The screenshot shows the pgAdmin 4 interface. On the left, the Browser pane displays a tree view of database objects, including Domains, FTS Configurations, FTS Dictionaries, FTS Parsers, FTS Templates, Foreign Tables, Functions, Materialized Views, Sequences, Tables (13), and Views. The 'customer' table is selected under the 'Tables (13)' category. A context menu is open over the 'customer' table, with the 'View Data' option highlighted.

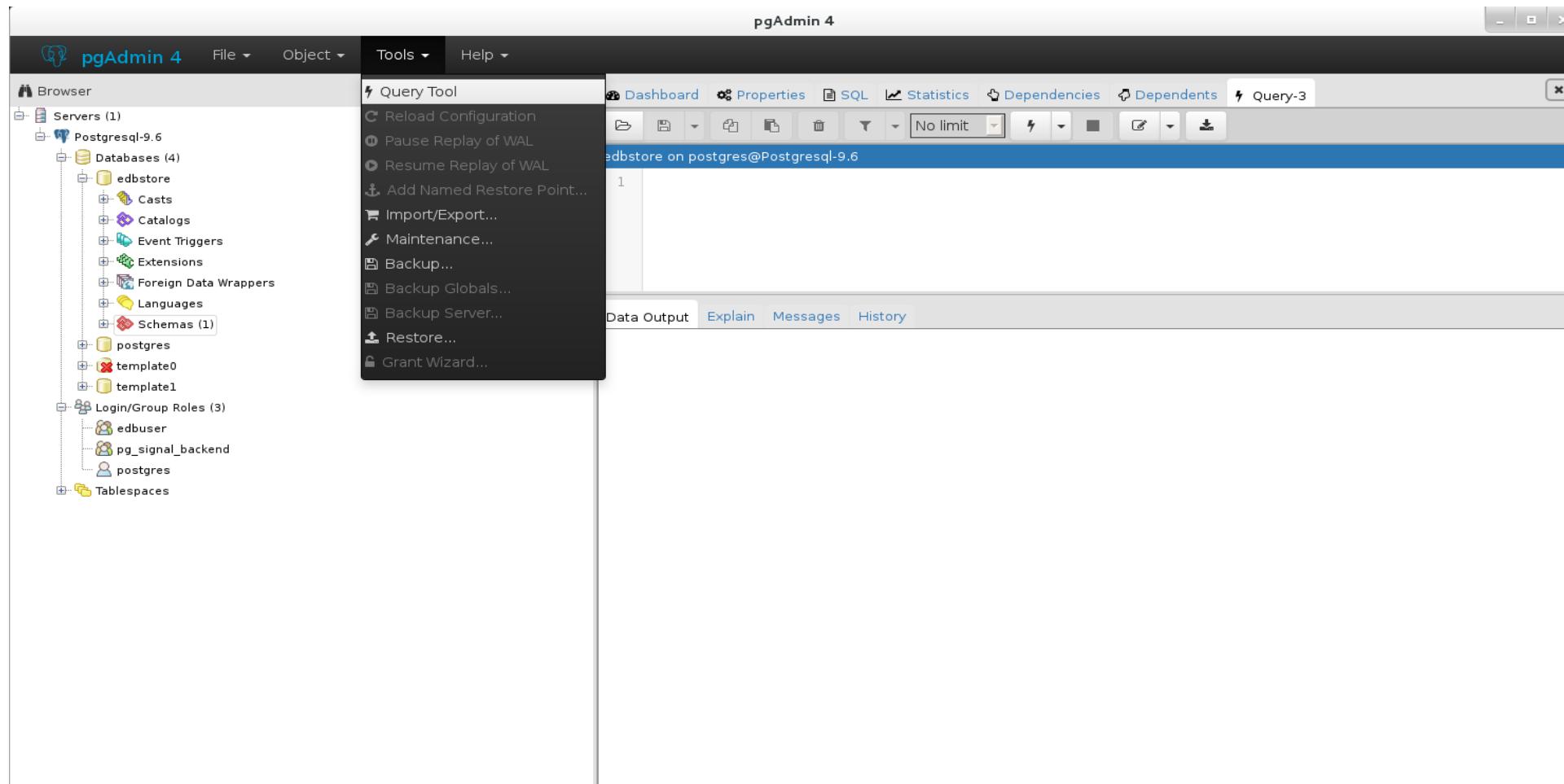
The main window shows the 'customer' table data. At the top, there is a SQL query editor with the following code:

```
1 SELECT * FROM edbuser.customers
2 ORDER BY customerid
3 ASC
```

Below the query is a 'Data Output' grid displaying 14 rows of customer data. The columns are labeled as follows:

	customerid [PK] integer	firstname character varying	lastname character varying	address1 character varying	address2 character varying	city character varying	state character varying	zip integer	country character varying	region smallint	email character varying
1	VKUUXF	ITHOMQJ...	4608499...		QSDPAGD	SD	24101	US		1	ITHOMQJ...
2	HQNMZH	UNUKXHJ...	5119315...		YNCERXJ	AZ	11802	US		1	UNUKXHJ...
3	JTNRNB	LYYSHTQJRE	6297761...		LWVIFXJ	OH	96082	US		1	LYYSHTQJ...
4	XMFYXD	WQLQHUU...	9862764...		HOKECD	MS	78442	US		1	WQLQHUU...
5	PGDTDU	ETBYBNE...	2841895...		RZQTCDN	AZ	16291	US		1	ETBYBNE...
6	FXDZBW	BAXPEEK...	6192740...		OPLRCNT	IN	99300	US		1	BAXPEEK...
7	WVZTXZ	RMEVXCQ...	9743191...		SIIGBQF	NV	55961	US		1	RMEVXCQ...
8	LIWLAI	PVGRMM...	7576564...		BKSRRQIE	NJ	66444	US		1	PVGRMM...
9	NCGWRC	CJOPRHU...	7291678...		ZAVIELY	VT	78838	US		1	CJOPRHU...
10	FUOHHX	WMOEH...	2603867...		HERSDPM	TN	75182	US		1	WMOEH...
11	XQVVMI	KRPGDBC...	2415449...		ICLYPGR	PA	53868	US		1	KRPGDBC...
12	KGISQZ	IXDKAUU...	1896033...		SBLZSFM	UT	18452	US		1	IXDKAUU...
13	LURLDP	PNPJHXM...	3029418...		QPGVBCY	DE	53356	US		1	PNPJHXM...
14	AGUQVI	FFPCRUS...	3748672...		LGKNEOA	MA	44395	US		1	FFPCRUS...

Query Tool



Query Tool - Data Output

pgAdmin 4

edbstore on postgres@Postgresql-9.6

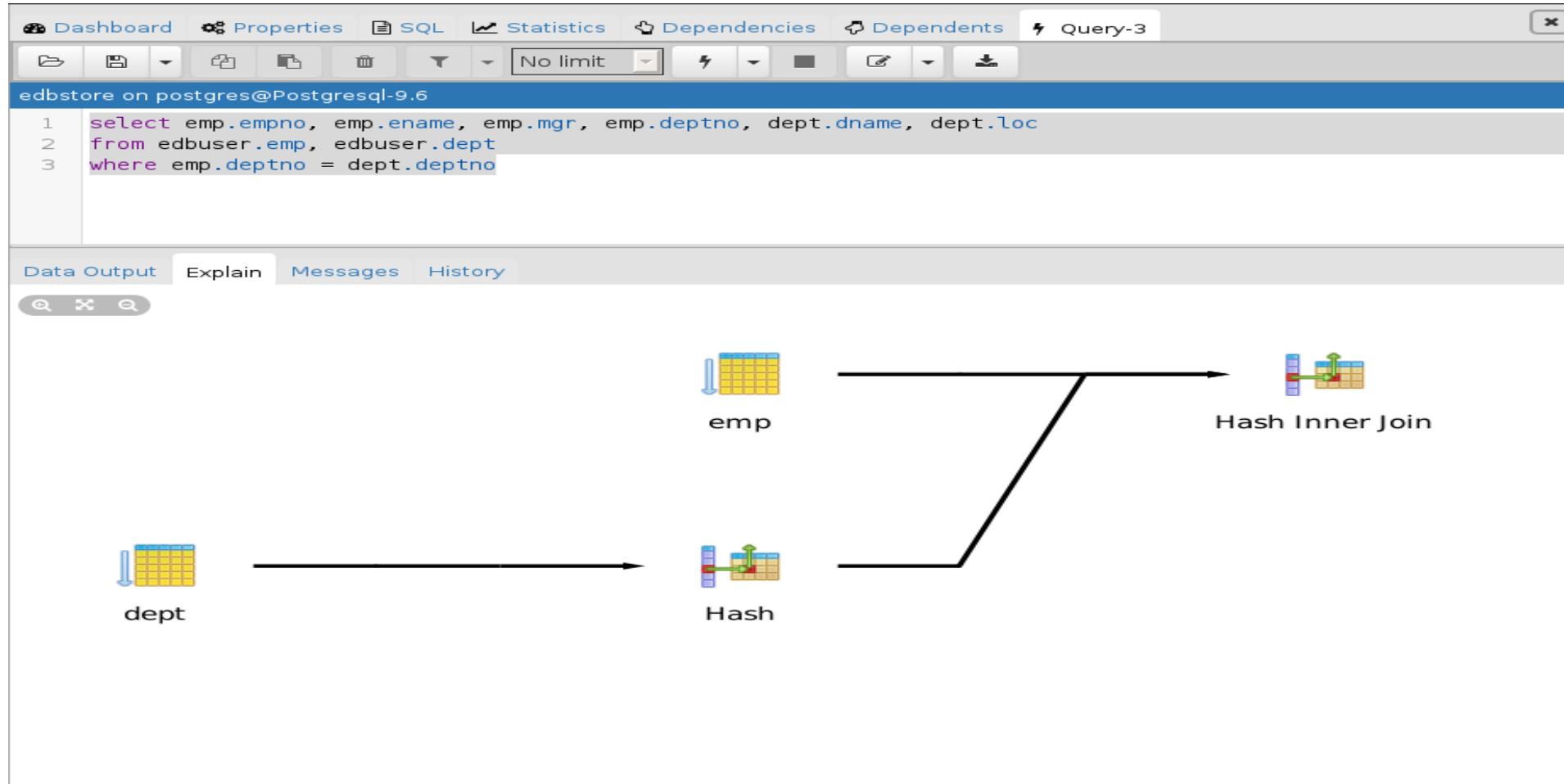
1 select * from edbuser.customers

Execute/Refresh (F5)
Explain (F7)
Explain Analyze (Shift+F7)
Explain Options
Auto commit?
Auto rollback?

Data Output Explain Messages History

customer_id	firstname	lastname	address1	address2	city	state	zip	country	region	email	phone	creditcard	creditcard_type	creditcard_expiry	username	password
1	VKUUXF	ITHOMQJ...	4608499...		QSDPAGD	SD	24101	US	1	ITHOMQJ...	4608499...	1	1979279...	2012/03	user1	pass1
2	HQNMZH	UNUKXHJ...	5119315...		YNCERXJ	AZ	11802	US	1	UNUKXHJ...	5119315...	1	3144519...	2012/11	user2	pass2
3	JTNRNB	LYYSHTQJRE	6297761...		LWVIFXJ	OH	96082	US	1	LYYSHTQJ...	6297761...	4	8728086...	2010/12	user3	pass3
4	XMFYXD	WQLQHU...	9862764...		HOKEXCD	MS	78442	US	1	WQLQHU...	9862764...	5	7160005...	2009/09	user4	pass4
5	PGDTDU	ETBYBNE...	2841895...		RZQTCDN	AZ	16291	US	1	ETBYBNE...	2841895...	3	8377095...	2010/10	user5	pass5
6	FXDZBW	BAXPEEK...	6192740...		OPLRCNT	IN	99300	US	1	BAXPEEK...	6192740...	5	7730283...	2011/01	user6	pass6
7	WVZTXZ	RMEVXCQ...	9743191...		SIIGBQF	NV	55961	US	1	RMEVXCQ...	9743191...	2	5914961...	2011/02	user7	pass7
8	LIWLAI	PVGRMM...	7576564...		BKSRQJE	NJ	66444	US	1	PVGRMM...	7576564...	2	7663945...	2009/12	user8	pass8
9	NCGWRC	CJOPRHU...	7291678...		ZAVIELY	VT	78838	US	1	CJOPRHU...	7291678...	1	7172072...	2009/10	user9	pass9
10	FUOHXX	WMOEH...	2603867...		HERSDPM	TN	75182	US	1	WMOEH...	2603867...	2	5486729...	2008/02	user10	pass10
11	XQVVMI	KRPGDBC...	2415449...		ICLYPGR	PA	53868	US	1	KRPGDBC...	2415449...	5	6630987...	2010/03	user11	pass11
12	KGISQZ	IXDKAUU...	1896033...		SBLZSFM	UT	18452	US	1	IXDKAUU...	1896033...	2	3715867...	2011/10	user12	pass12
13	LURLDP	PNPJHXM...	3029418...		QPGVBCY	DE	53356	US	1	PNPJHXM...	3029418...	5	3617457...	2009/11	user13	pass13
14	AGUQVI	FFPCRUS...	3748672...		LGKNEOA	MA	44395	US	1	FFPCRUS...	3748672...	4	3344003...	2011/07	user14	pass14
15	SIQANV	QQNKJSU...	3354132...		BREQSOA	AK	37471	US	1	QQNKJSU...	3354132...	4	8717996...	2008/05	user15	pass15
16	IXEENV	RXEKSW...	6914808...		YOYMOYOU	SD	63504	US	1	RXEKSW...	6914808...	5	9422665...	2009/11	user16	pass16
17	UUGPME	UWWRKP...	1436497...		PASNVNC	MO	55931	US	1	UWWRKP...	1436497...	5	4461925...	2011/06	user17	pass17
18	KASOVP	LMZBBQ...	4002123...		YTMLWYY	VA	79031	US	1	LMZBBQ...	4002123...	3	2972224...	2010/12	user18	pass18
19	FLUTYC	TRIMNO...	6004007...		QYEDWYI	CA	60100	US	1	TRIMNO...	6004007...	4	6004007...	6004007...	user19	pass19

Query Tool - Explain



Query Tool - Messages

The screenshot shows a database query tool window with the following details:

- Toolbar:** Includes icons for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a search bar labeled "Query-3".
- Connection:** Connected to "edbstore on postgres@Postgresql-9.6".
- SQL Editor:** Contains the following SQL query:

```
1 select emp.empno, emp.ename, emp.mgr, emp.deptno, dept.dname, dept.loc
2 from edbuser.emp, edbuser.dept
3 where emp.deptno = dept.deptno
```
- Execution Results:** Shows the total query runtime as 232 msec and 14 rows retrieved.
- Bottom Navigation:** Includes tabs for Data Output, Explain, Messages (which is selected), and History.

Query Tool - History

The screenshot shows a database management interface with a toolbar at the top and a main content area below. The toolbar includes tabs for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and Query-5. Below the toolbar is a toolbar with various icons for file operations like Open, Save, Print, and a search bar set to 'No limit'. The main content area has a blue header bar labeled 'edbstore on postgres@Postgresql-9.6'. Underneath is a code editor window containing the following SQL query:

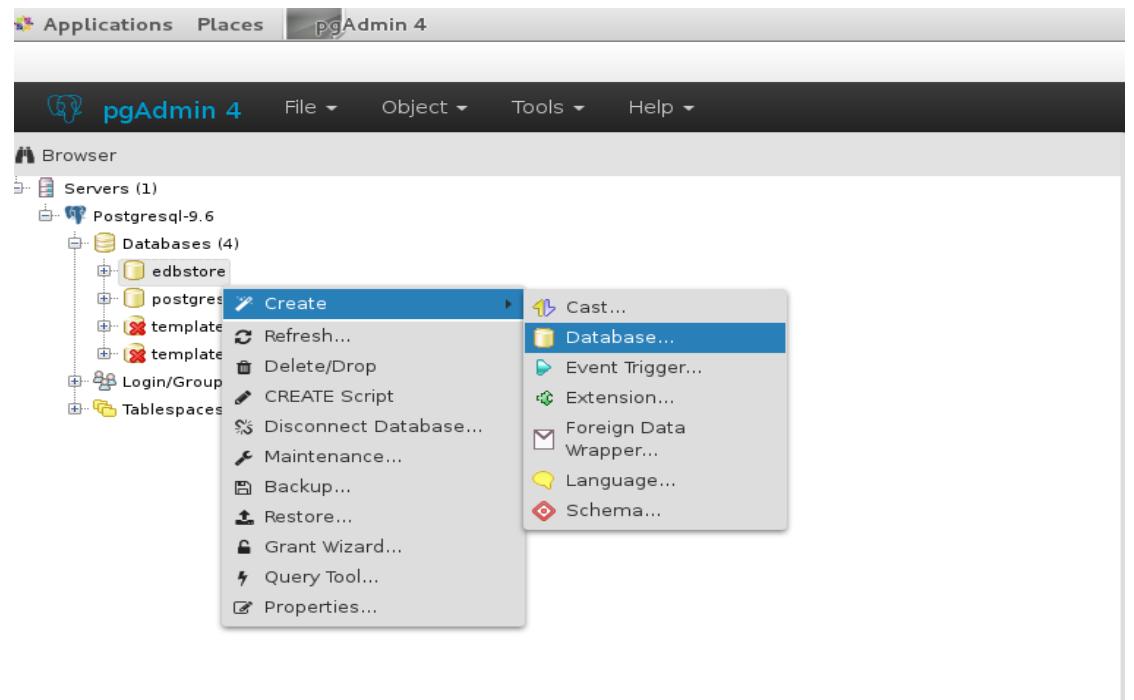
```
1 select emp.empno, emp.ename, emp.mgr, emp.deptno, dept.dname, dept.loc
2 from edbuser.emp, edbuser.dept
3 where emp.deptno = dept.deptno
```

Below the code editor is a table titled 'History' with columns: Date, Query, Rows affected, Total Time, and Message. It contains two rows of data:

Date	Query	Rows affected	Total Time	Message
Mon Oct 03 2016 22:48:30 GMT+05...	select emp.empno, emp.ename, emp...	14	336 msec	
Mon Oct 03 2016 22:47:09 GMT+05...	select * from edbuser.customers	20,000	7 secs	

Databases

- The databases menu allows you to create a new database
- The menu for an individual database allows you to perform operations on that database
 - Create a new object in the database
 - Drop the database
 - Open the Query Tool with a script to re-create the database
 - Perform maintenance
 - Backup or Restore
 - Modify the databases properties



Creating a Database

Create - Database

General **Definition** Security Parameters SQL

Database

Owner

Comment

Save **Cancel** **Reset**

Create - Database

General **Definition** Security Parameters SQL

Encoding

Template

Tablespace

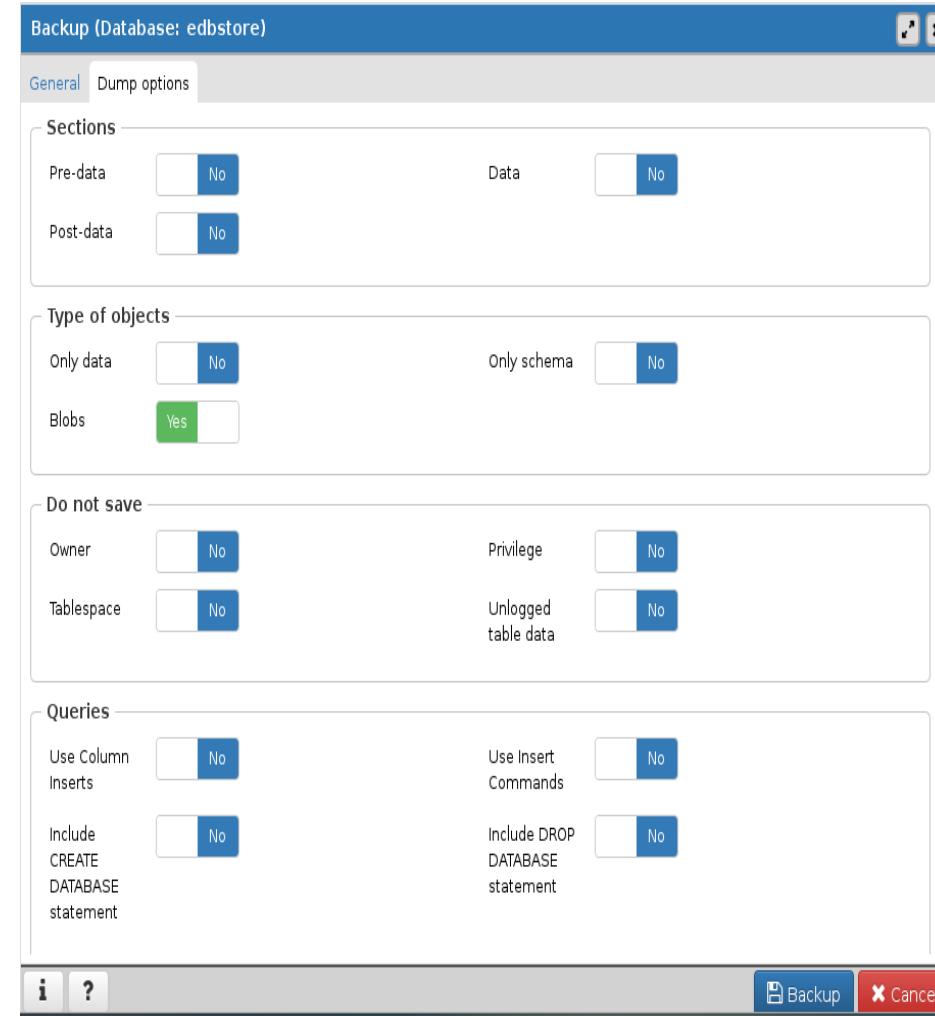
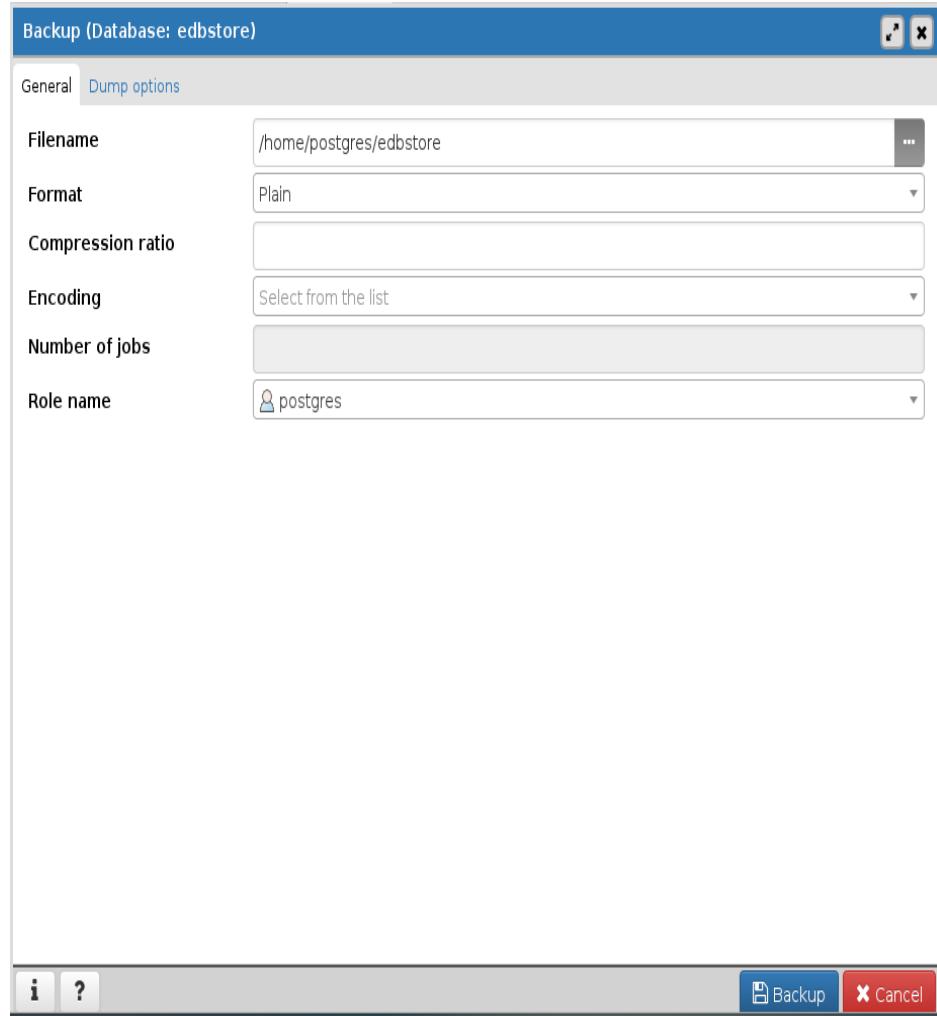
Collation

Character type

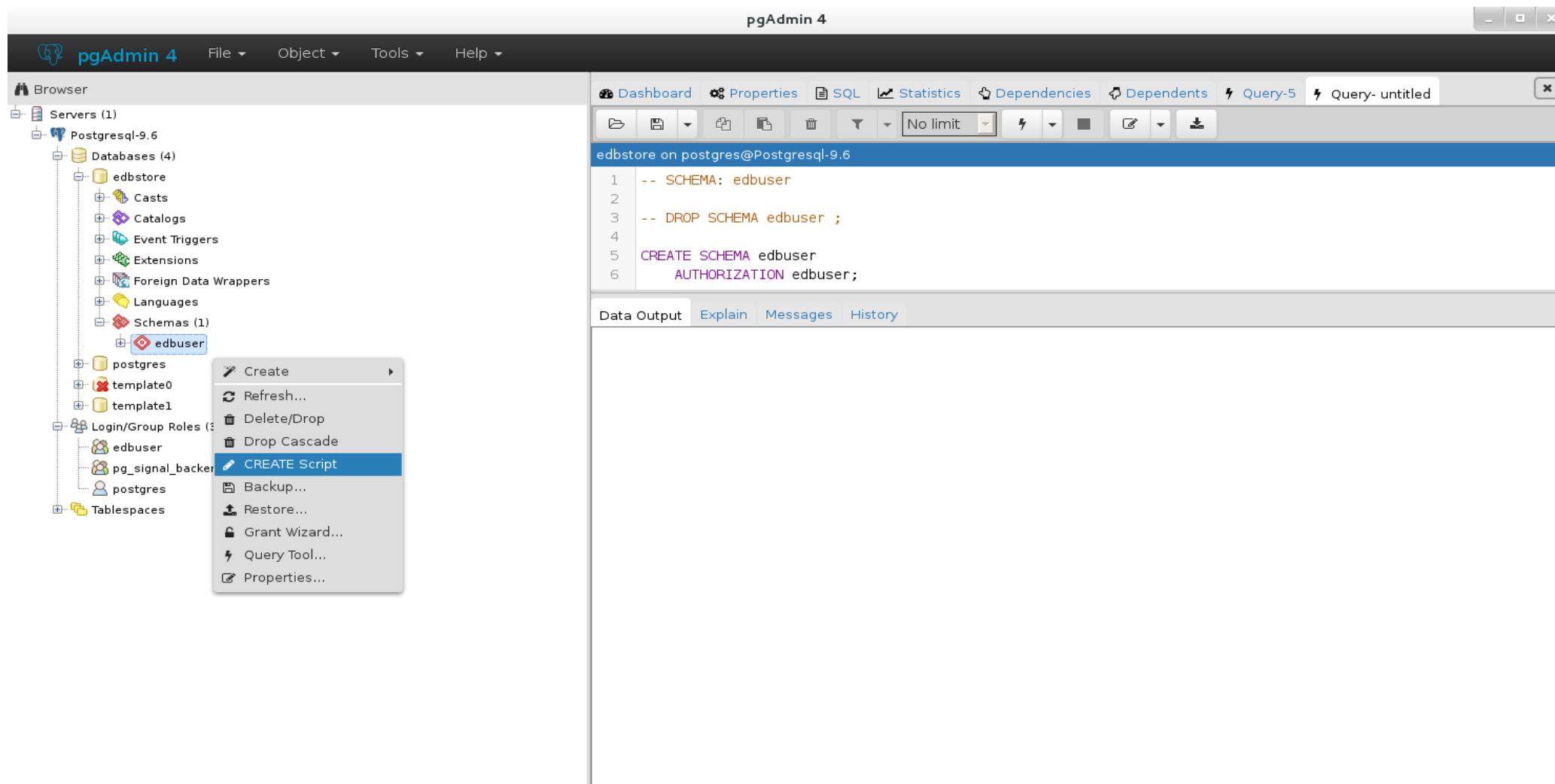
Connection limit

Save **Cancel** **Reset**

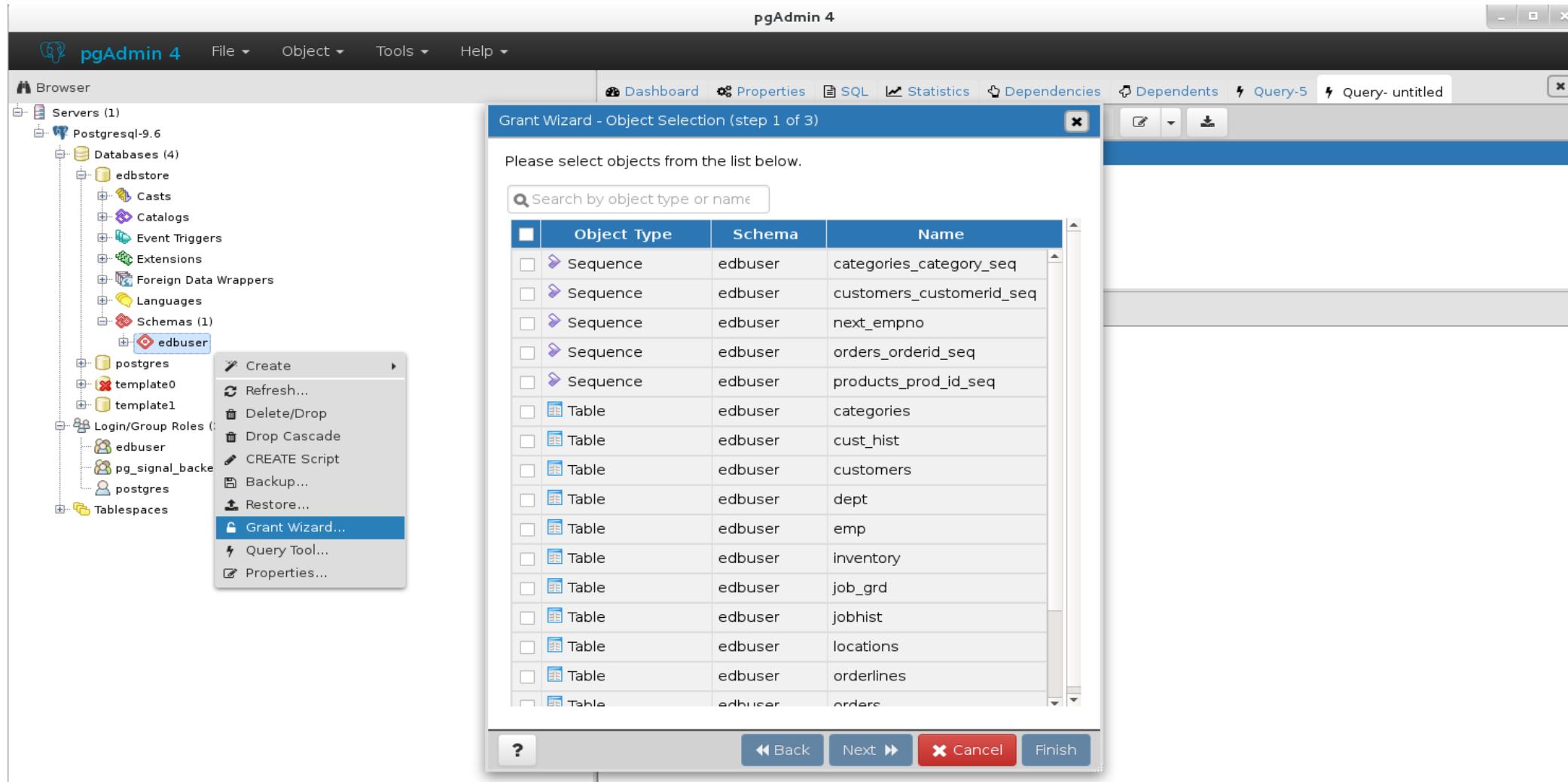
Backup and Restore



Schemas



Schemas - Grant Wizard



Domains

Create - Domain

General Definition Constraints Security SQL

Name

Owner postgres

Schema edbuser

Comment

Save **Cancel** **Reset**

Create - Domain

General Definition Constraints Security SQL

Base type Select from the list

Length

Precision

Default Enter an expression or a value.

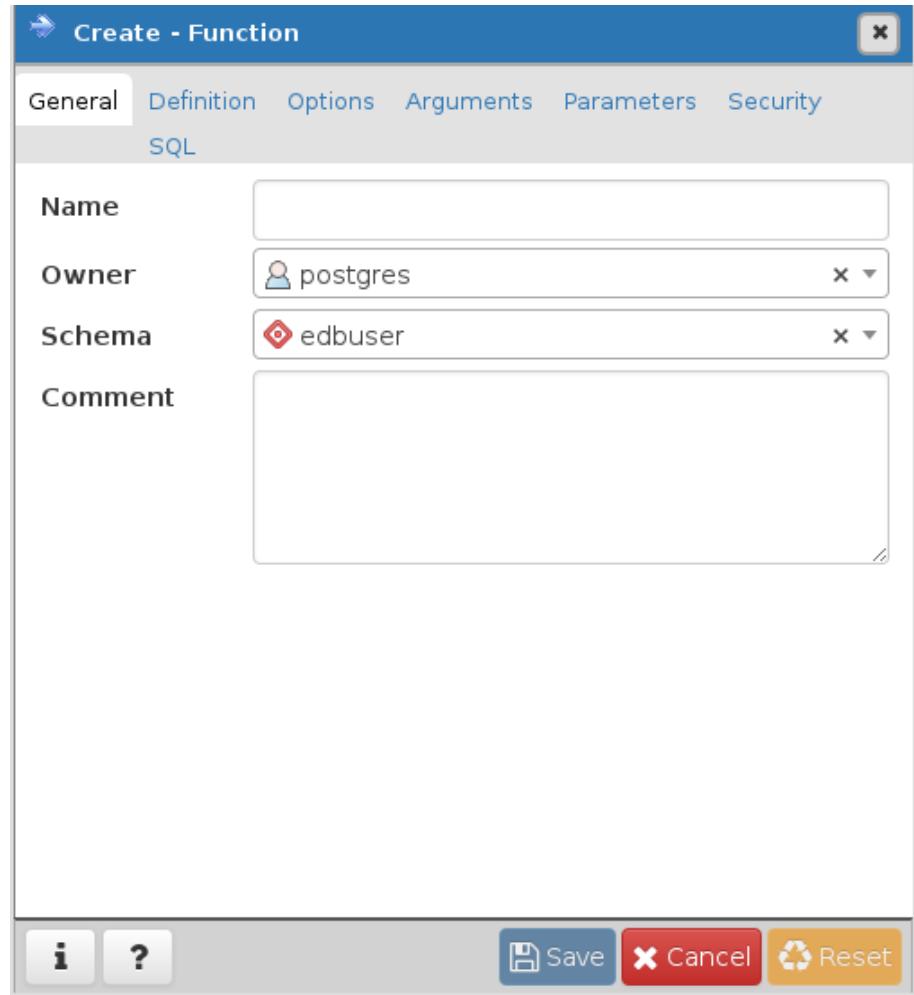
Not Null?

Collation Select from the list

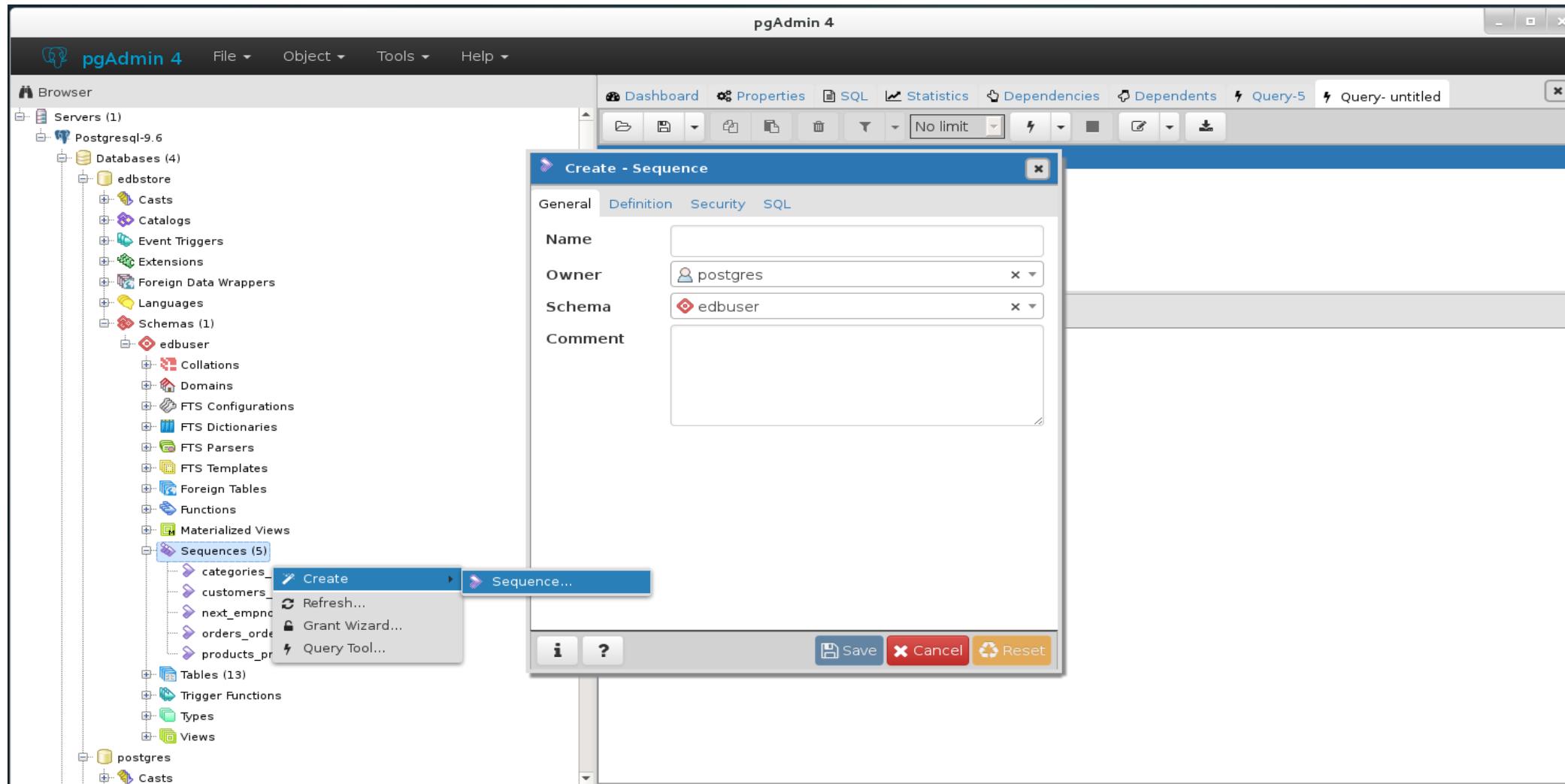
Save **Cancel** **Reset**

Functions

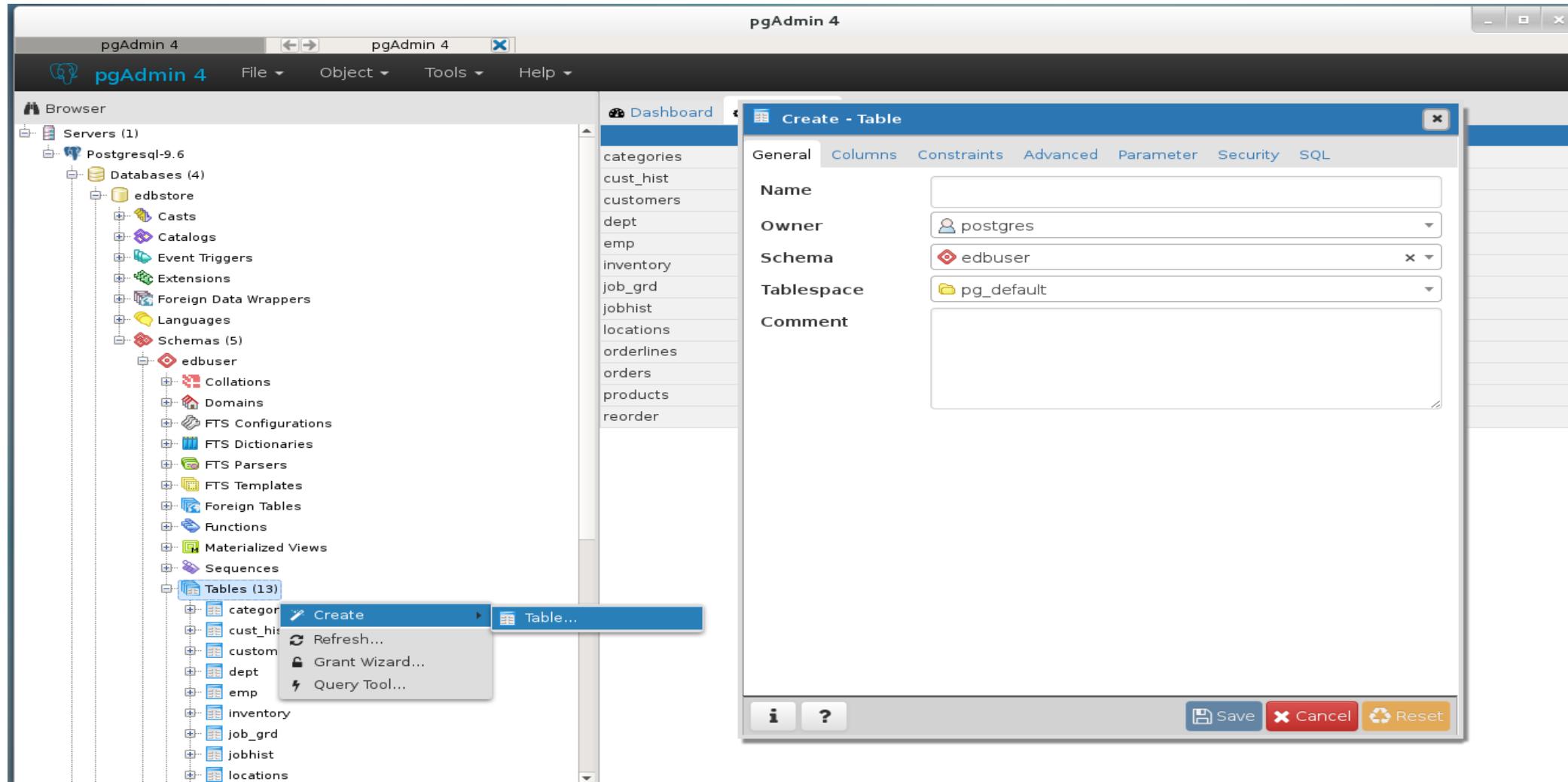
- Functions, Trigger Functions and Procedures are all identical except for their Return Type
 - Trigger Functions have a fixed return type of `trigger`
 - Procedures do not return anything; their return type is `void`



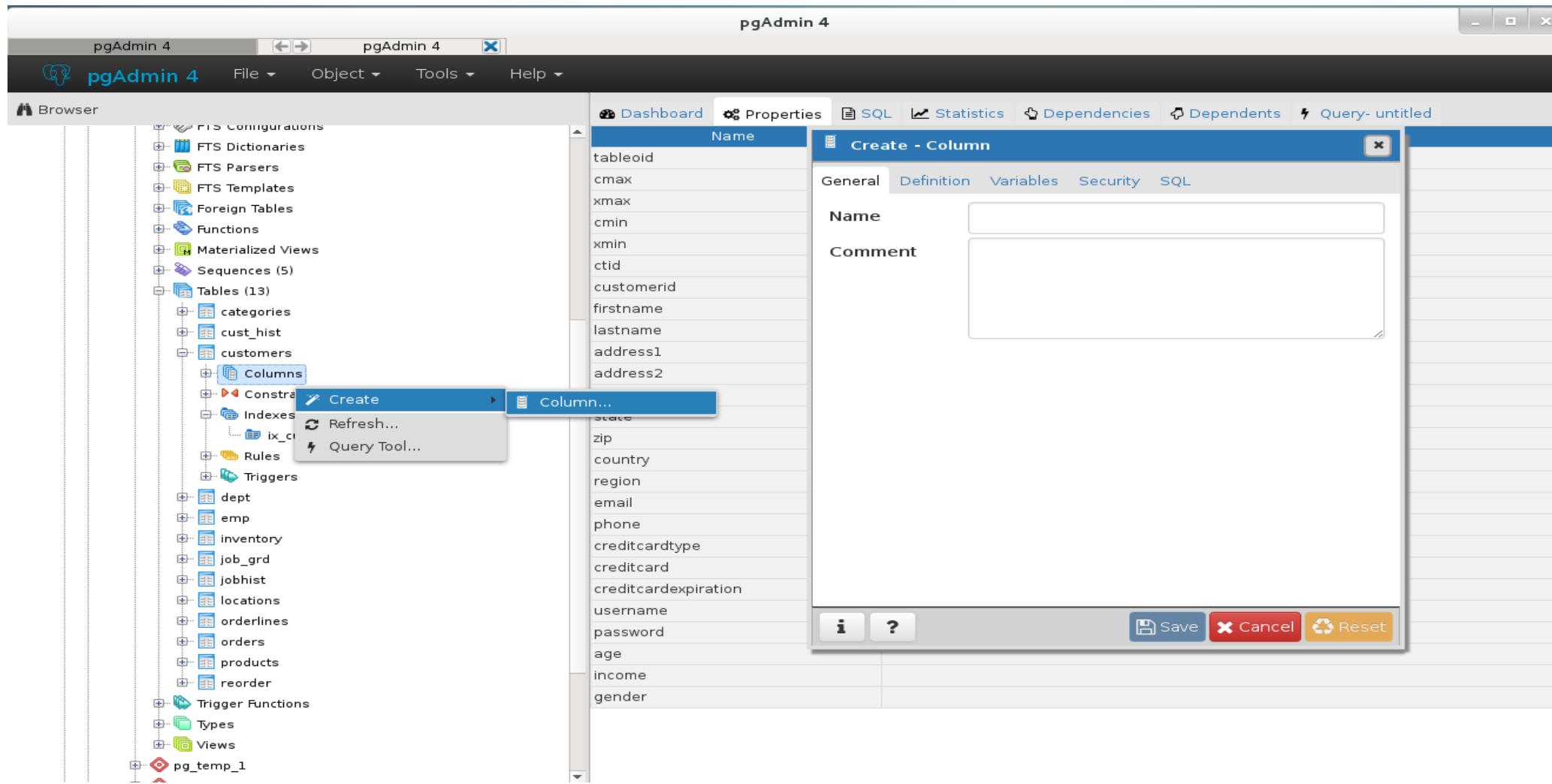
Sequences



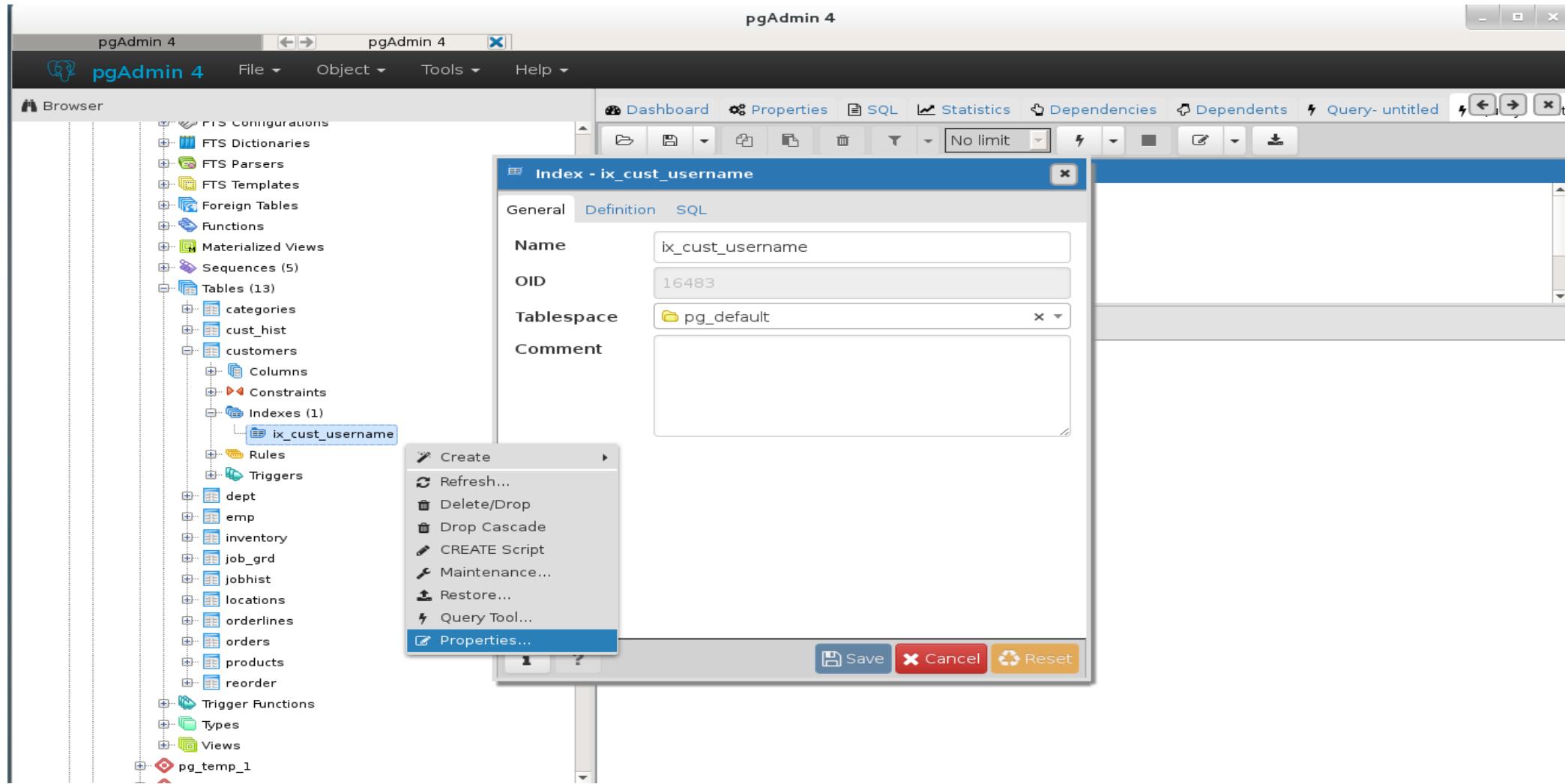
Tables



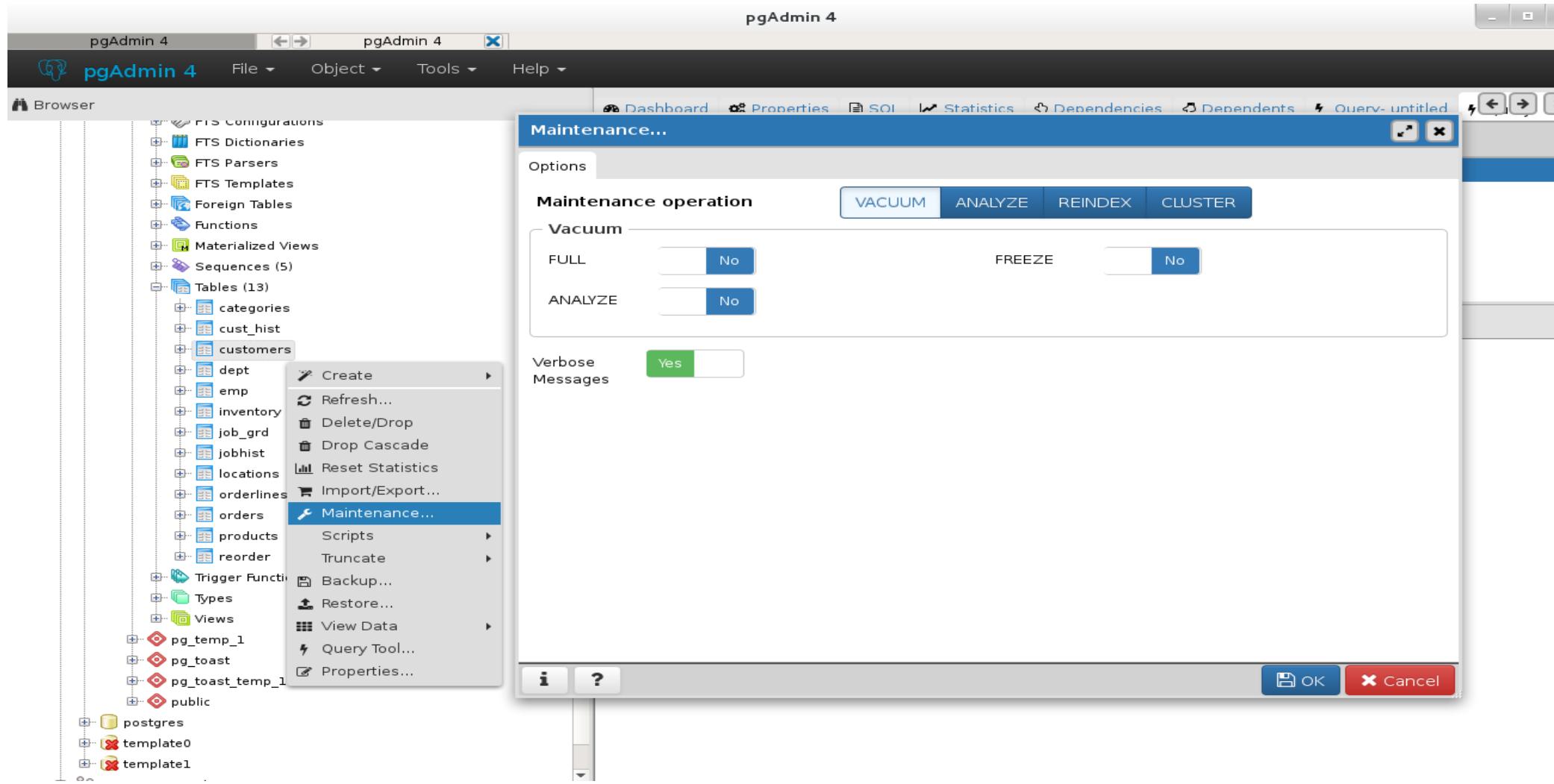
Tables - Columns



Tables - Indexes

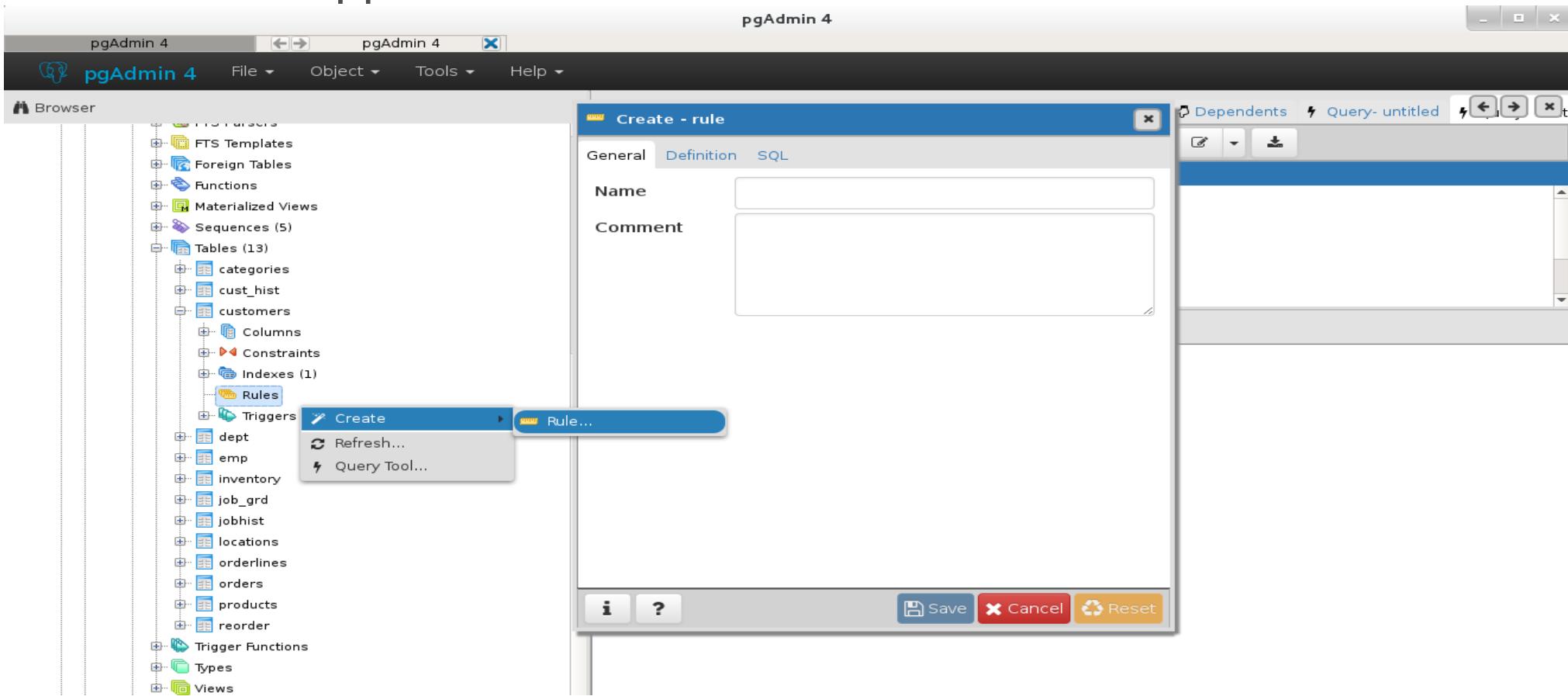


Tables - Maintenance



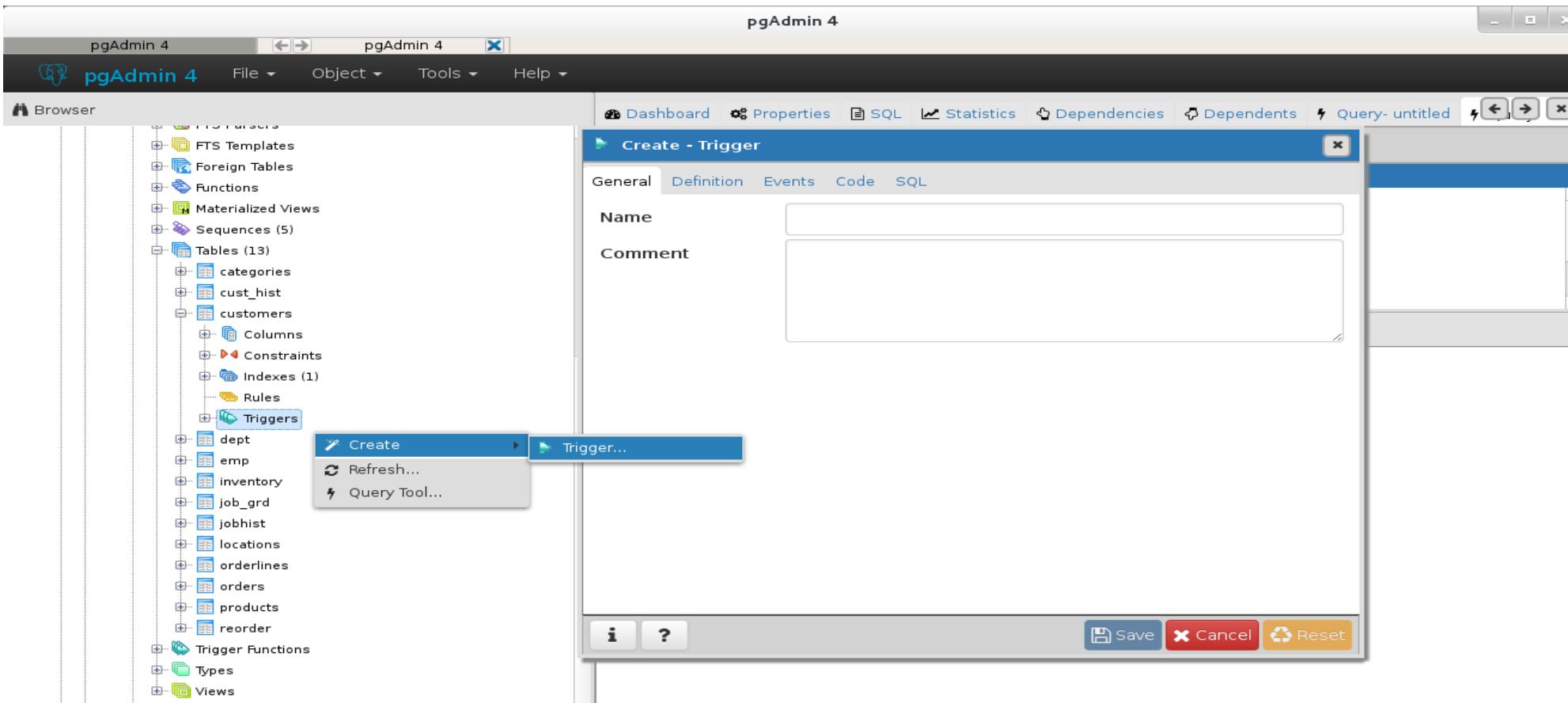
Rules

- Rules can be applied to tables or views

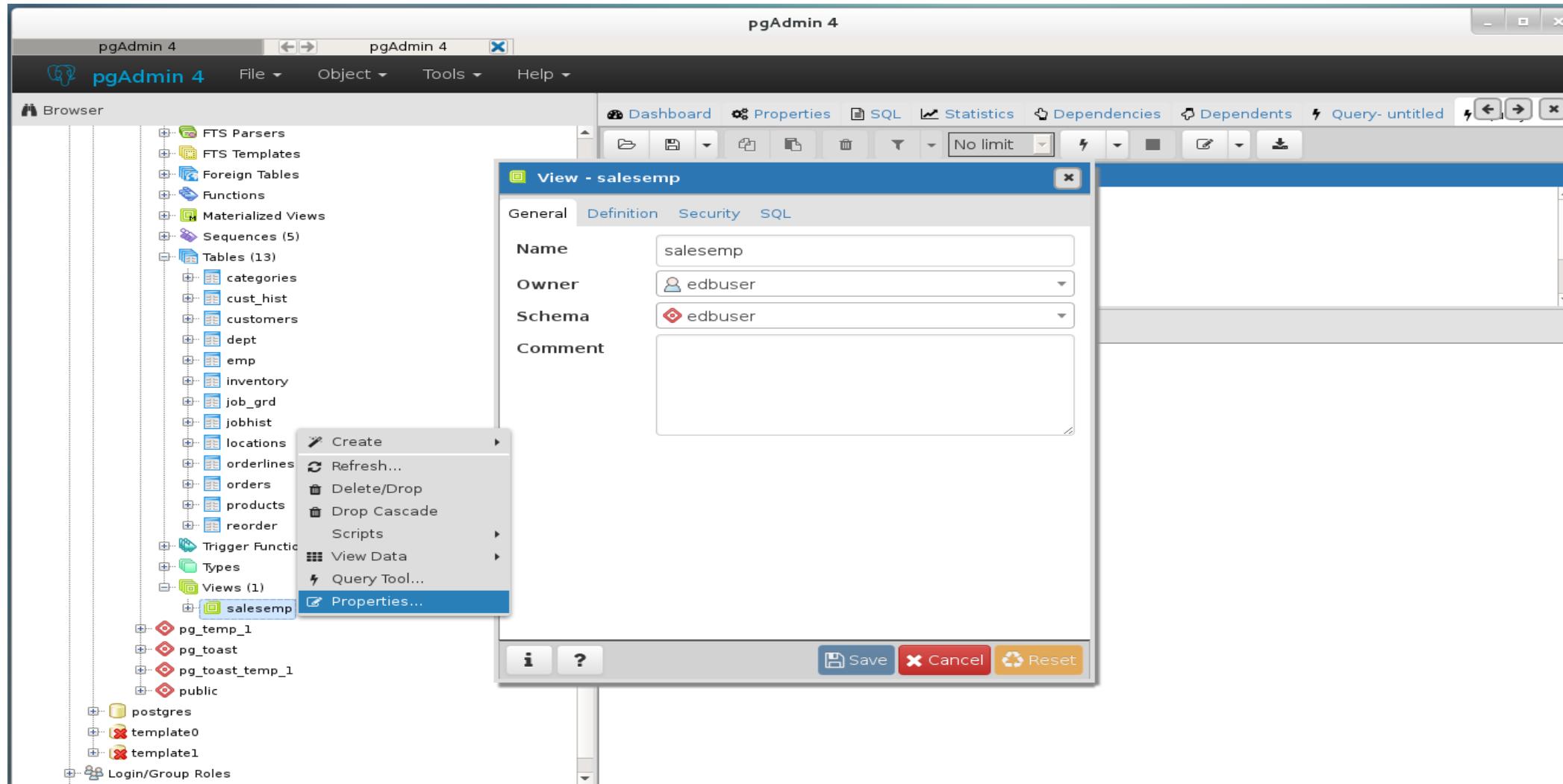


Triggers

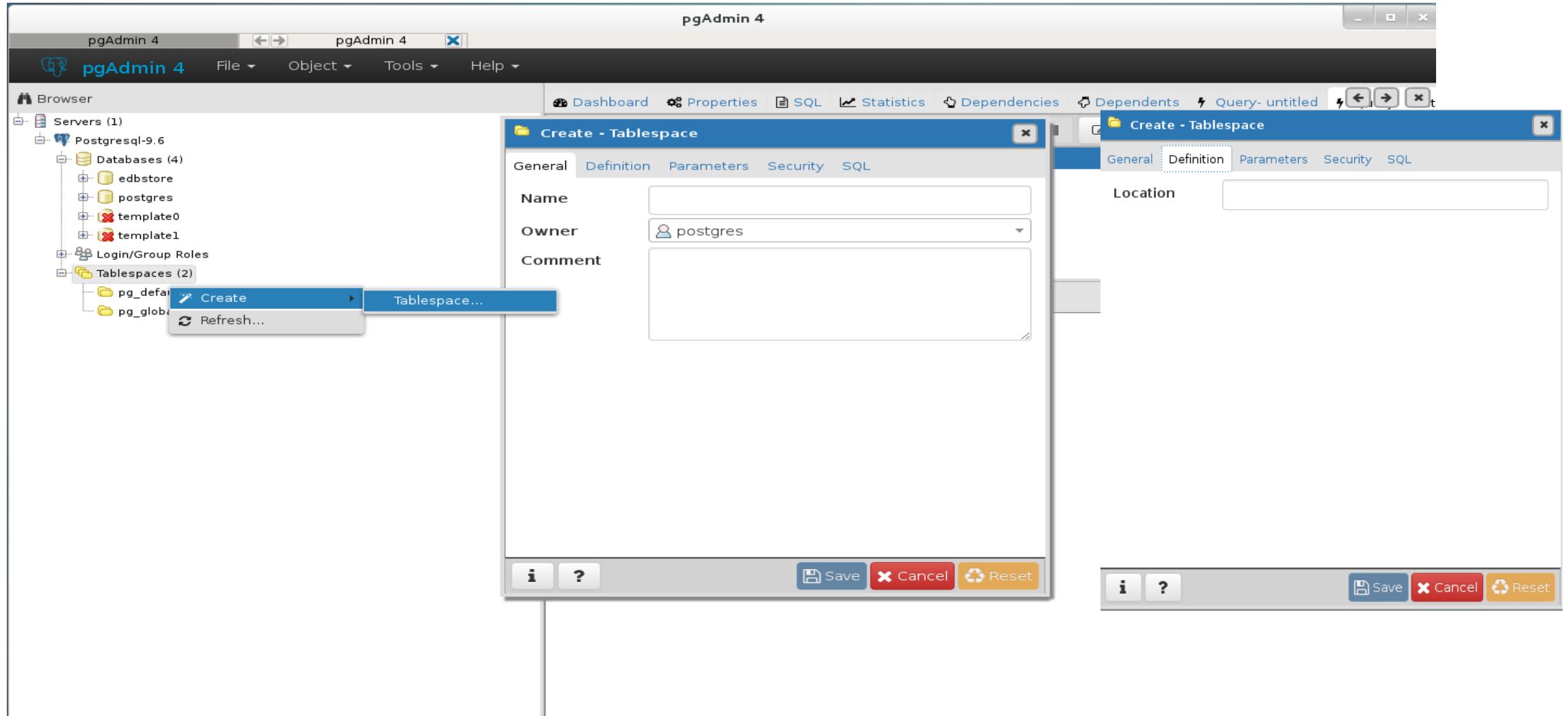
- Create a trigger function before creating a trigger



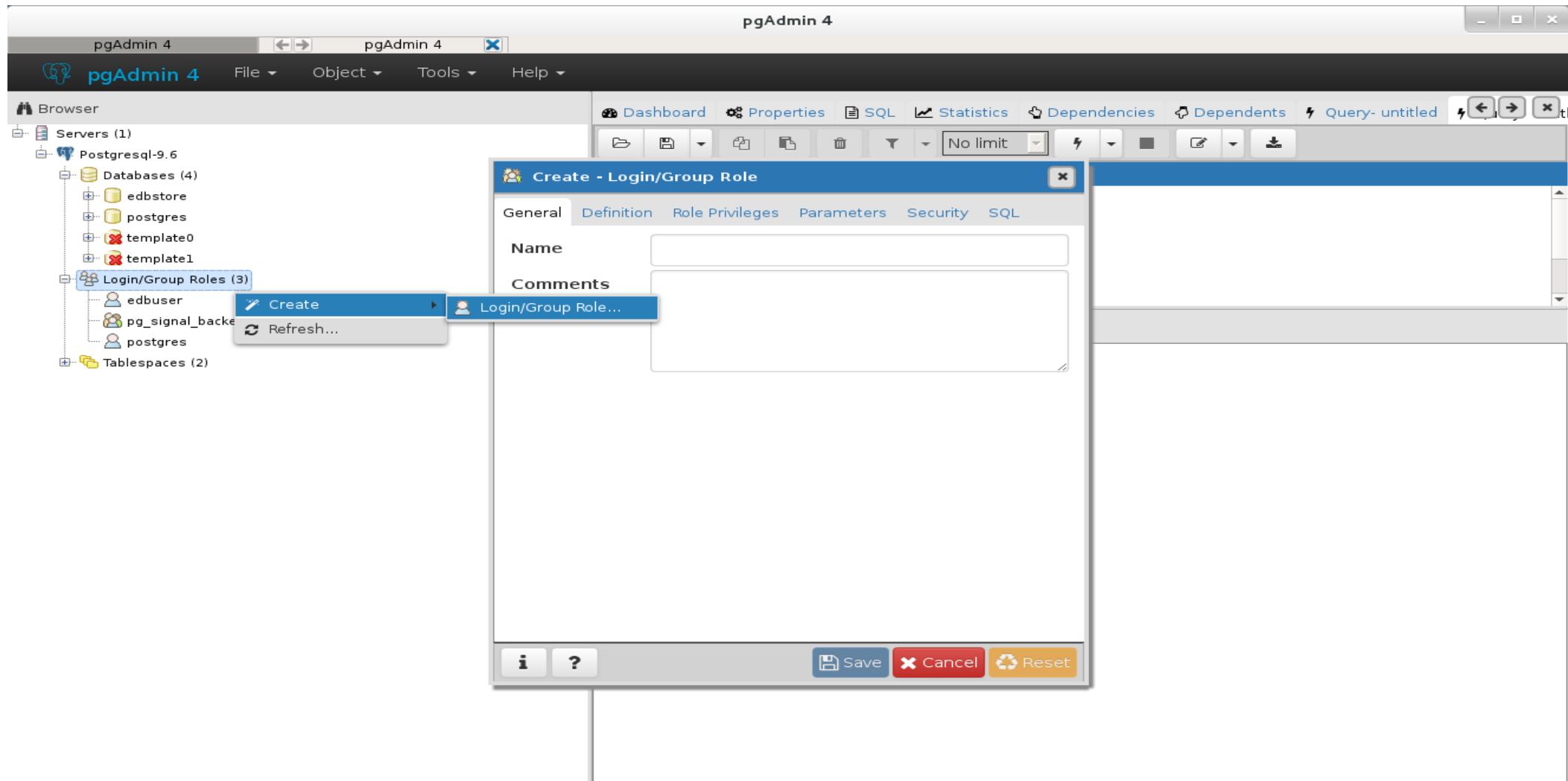
Views



Create Tablespaces

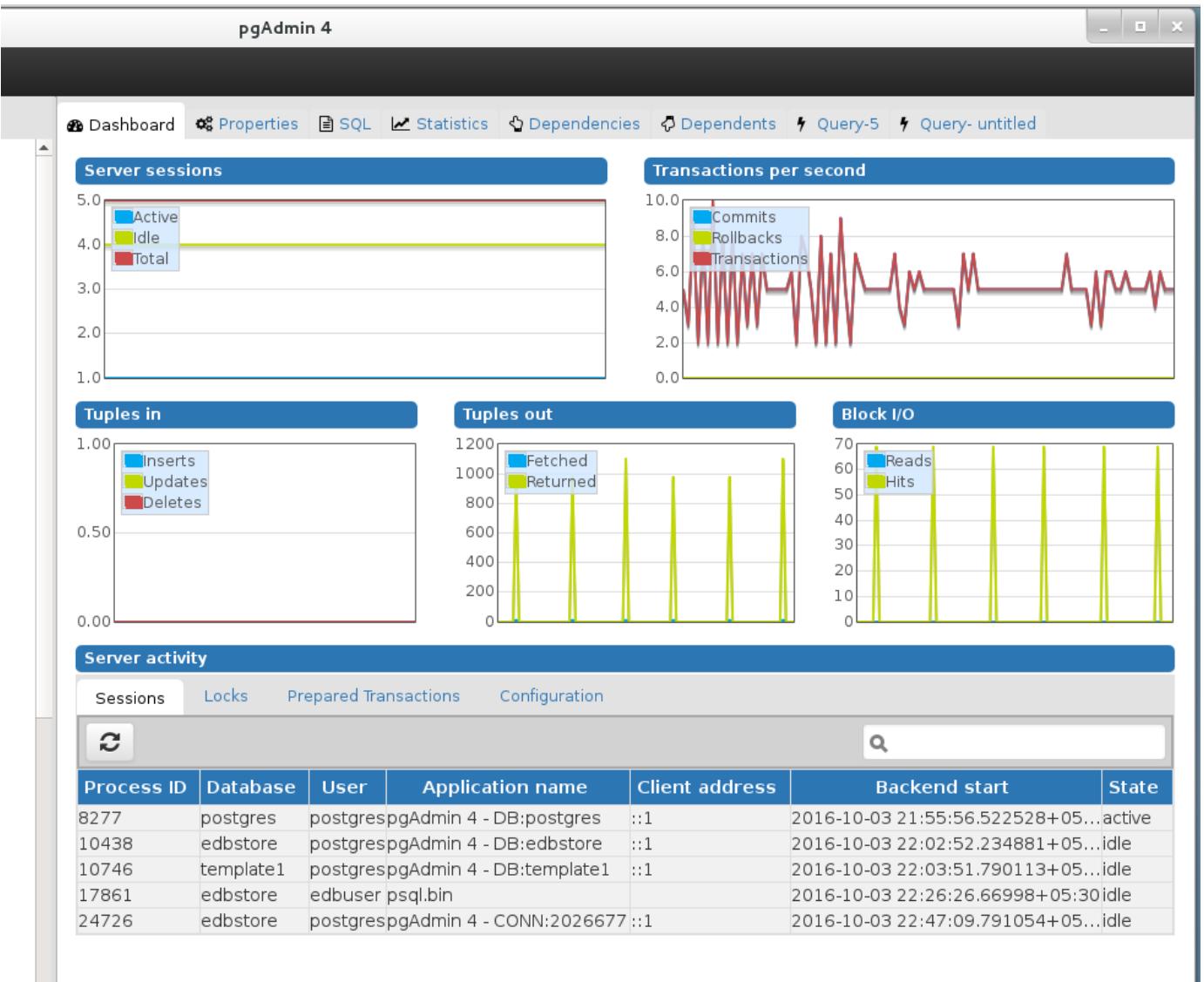


Roles



Dashboard

- Server Sessions
- Transaction per second
- Tuples in
- Tuples out
- Block I/O
- Server activity - sessions



Statistics

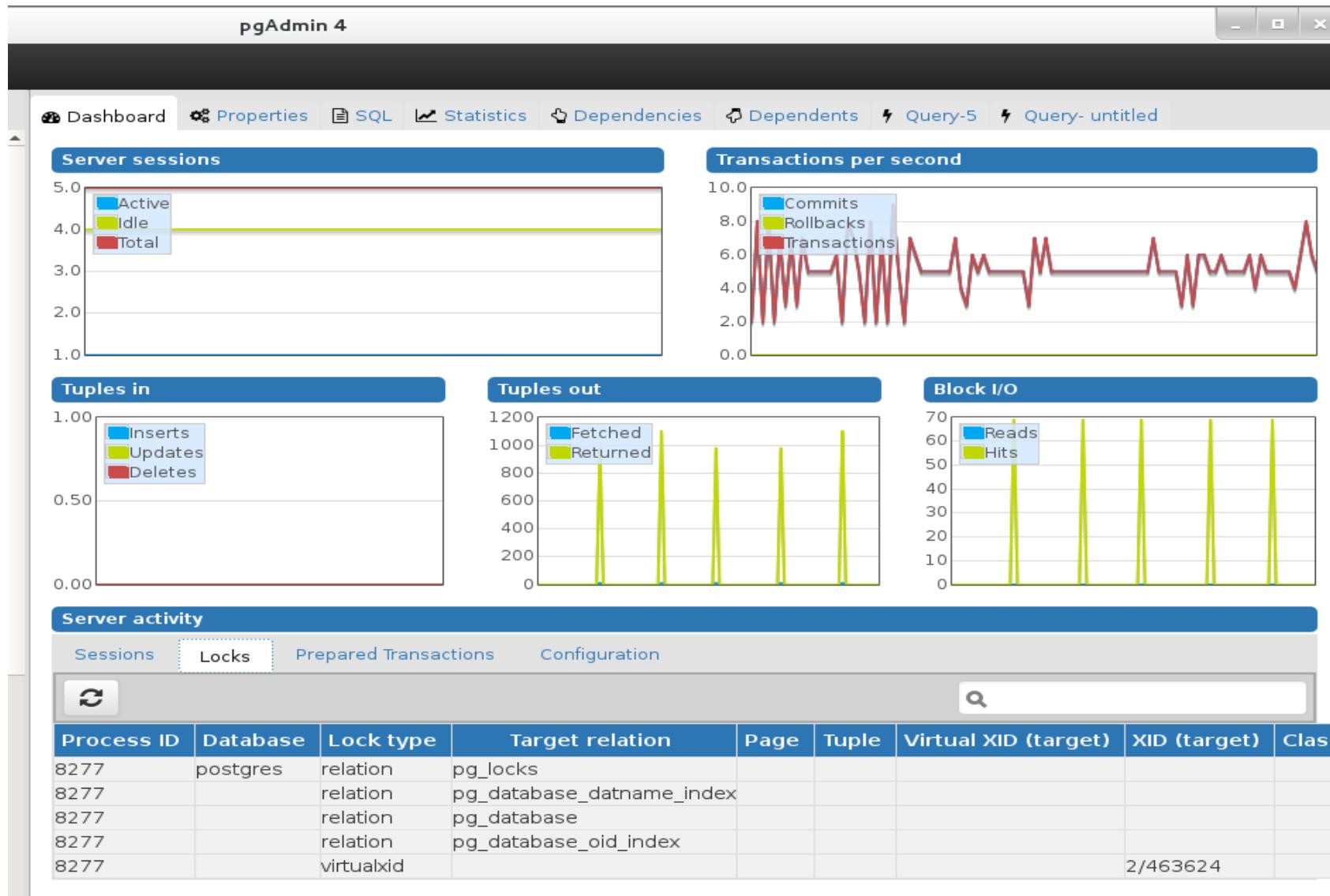
pgAdmin 4

Object ▾ Tools ▾ Help ▾

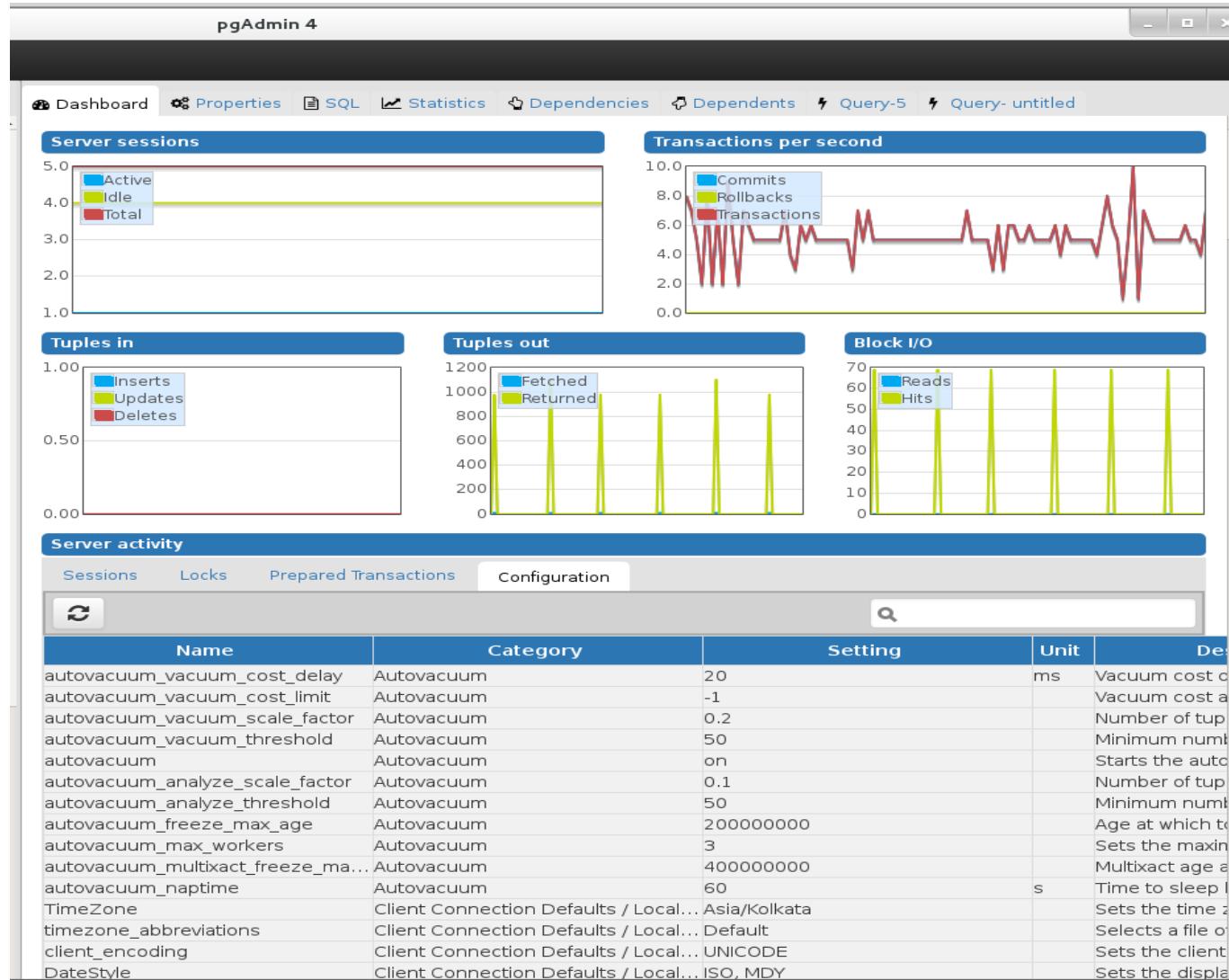
Dashboard Properties SQL Statistics Dependencies Dependents Query-5 Query- untitled

PID	User	Database	Backend start	Client	Application	Wait event type	Wait event name	Query	Query start
8,277	postgres	postgres	2016-10-03 21:55:56.522528+...	::1/128:38585	pgAdmin 4 - DB:postgres			SELECT pid AS "PID", username A...	2016-10-06 15:38:51.082271+..
10,438	postgres	edbstore	2016-10-03 22:02:52.234881+...	::1/128:40752	pgAdmin 4 - DB:edbstore			SELECT nsp.oid, nsp.nspname a...	2016-10-03 22:58:08.639589+..
10,746	postgres	template1	2016-10-03 22:03:51.790113+...	::1/128:41072	pgAdmin 4 - DB:template1			SELECT oid as id, rolname as na...	2016-10-03 22:03:51.816743+..
17,861	edbuser	edbstore	2016-10-03 22:26:26.66998+0...	local pipe	psql.bin			GRANT ALL ON SCHEMA public T...	2016-10-03 22:26:55.285135+..
24,726	postgres	edbstore	2016-10-03 22:47:09.791054+...	::1/128:54996	pgAdmin 4 - CONN:2026677			SELECT oid, format_type(oid,null)...	2016-10-03 22:48:30.90474+0..

Server Activity - Locks



Server Activity - Configuration



Introduction to EDB Postgres Enterprise Manager (PEM)

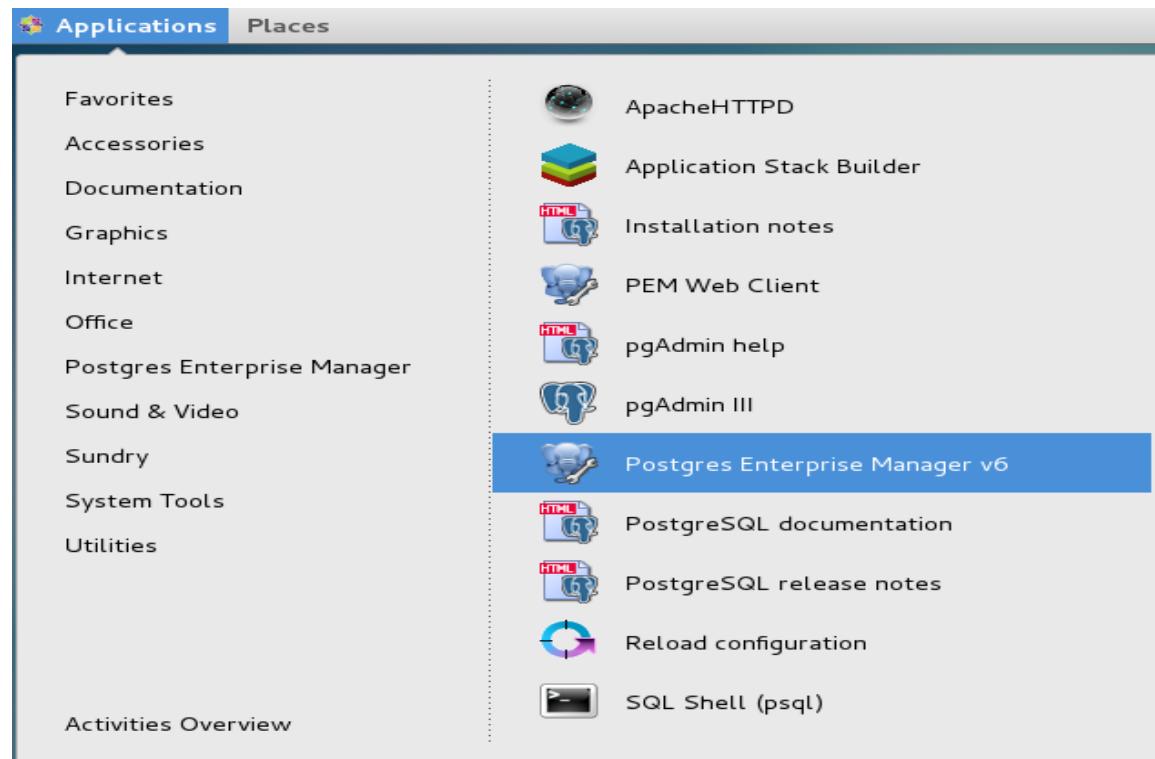
- EDB Postgres Enterprise Manager (PEM) is designed for administering, monitoring and tuning EDB Postgres Advanced Server
- Can be used to manage multiple database clusters
- Supports both EDB Postgres Advanced Server and PostgreSQL
- Can be installed using the Advanced Server installer or separate binary can be downloaded from www.enterprisedb.com
- PEM Client is based on the Open Source pgAdmin 4 project

Features - PEM

- PEM provides a number of key benefits:
 - Manage multiple EDB Postgres Advanced Server and PostgreSQL instances
 - Global dashboard display status, alerting and other information for all the managed clusters
 - Can collect statistics information regardless of the OS platform used by database clusters
 - Full SQL IDE including graphical debugger
 - Proactive Alerting
 - Capacity Planning
 - Audit Manager, Log Manager for viewing audit and error log
 - SQL Profiler

Open Postgres Enterprise Manager Client

- Postgres Enterprise Manager(PEM) Client can be run using the application menu of your OS



PEM Client First Look

Postgres Enterprise Manager

File Edit Plugins View Management Tools Help

Object browser

Property	Value
Name	postgres
OID	13294
Owner	postgres
ACL	
Tablespace	pg_default
Default tablespace	pg_default
Encoding	UTF8
Collation	en_US.UTF-8
Character type	en_US.UTF-8
Default schema	
Default table ACL	
Default sequence ACL	
Default function ACL	
Default type ACL	
Allow connections?	Yes
Connected?	Yes
Connection limit	-1
System database?	No
Comment	default administrative connection database

Properties Statistics Dependencies Dependents

Properties Statistics Dependencies Dependents

Module Summary

- Introduction to pgAdmin 4
- Registering a server
- Viewing and Editing Data
- Query Tool
- Databases
- Languages
- Schemas
- Database Objects
- Maintenance
- Tablespaces
- Roles
- Introduction to PEM Client

Lab Exercise - 1

1. Open pgAdmin 4 and connect to the default PostgreSQL database cluster
 - Create a user named pguser
 - Create a database named pgdb owned by pguser
 - After creating the pgdb database change its connection limit to 2
 - Create a schema named pguser inside the pgdb database
The schema owner should be pguser

Lab Exercise - 2

1. You have created the pgdb database with the pguser schema.
Create following objects in the pguser schema:
 - Table - Teams with columns TeamID, TeamName, TeamRatings
 - Sequence - seq_teamid start value - 1 increment by 1
 - Columns - Change the default value for the TeamID column to seq_teamid
 - Constraint - TeamRatings must be between 1 and 10
 - Index - Primary Key TeamID
 - View - Display all teams in ascending order of their ratings. Name the view as vw_top_teams

Lab Exercise - 3

1. View all rows present in Teams table.
2. Using the Edit data window you just opened in the previous step, insert the following rows into the Teams table:

TeamID	TeamName	TeamRatings
Auto generated	Oilers	1
Auto generated	Rangers	6
Auto generated	Canucks	8
Auto generated	Blackhawks	5
Auto generated	Bruins	2

Lab Exercise - 4

1. Connect to the pgdb database using the query tool
2. Using the graphical query builder retrieve all the rows present in the Teams table
3. Using the graphical query builder retrieve all the rows present in view vw_top_teams

Module 9

Security

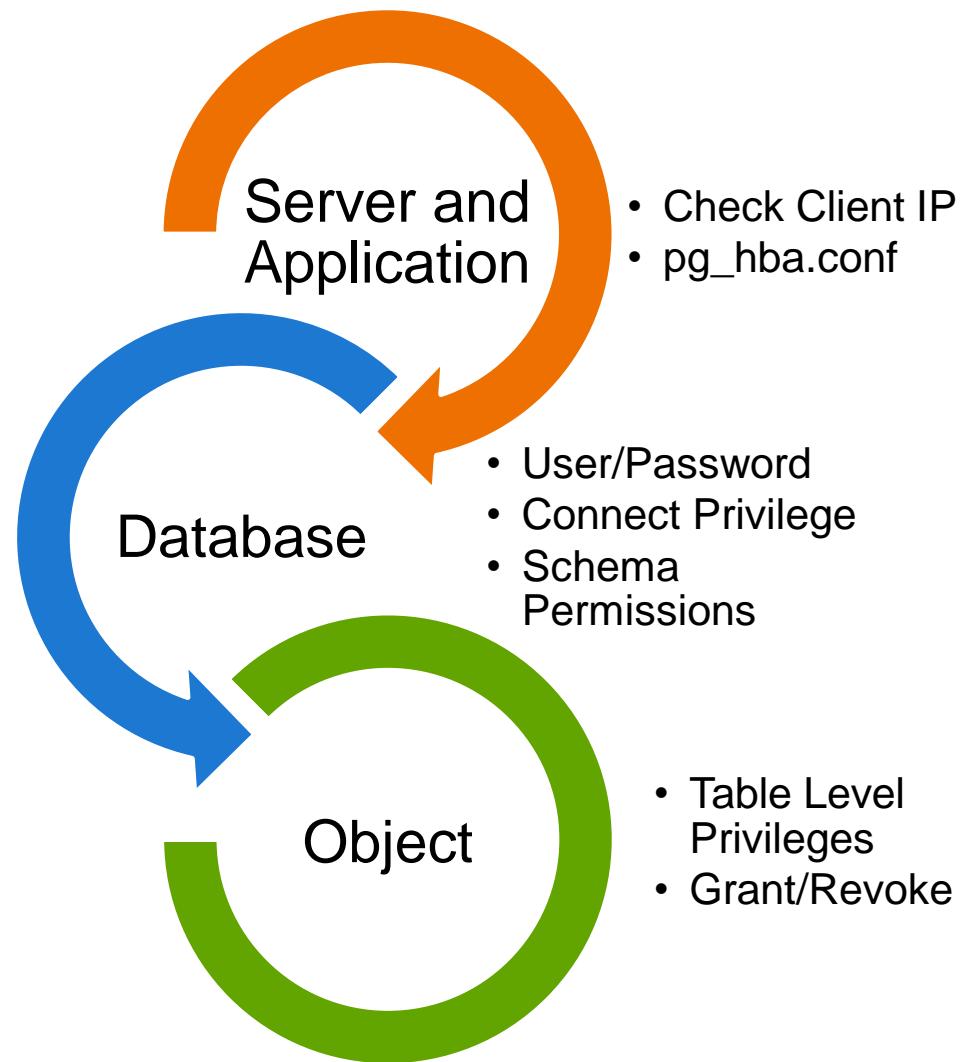
Module Objectives

- Authentication and Authorization
- Levels of Security
- pg_hba.conf File
- Row Level Security
- Object Ownership
- Application Access Parameters
- Protecting Against Injection Attacks with SQL/Protect
- Source Code Protection for Functions

Authentication and Authorization

- Secure access is a two step process:
 - Authentication
 - Ensures a user is who he/she claims to be
 - Authorization
 - Ensures an authenticated user has access to only the data for which he/she has been granted the appropriate privileges

Levels of Security



pg_hba.conf – Access Control

- Host based access control file
- Located in the cluster data directory
- Read at startup, any change requires reload
- Contain set of records, one per line
- Each record specify connection type, database name, user name , client IP and method of authentication
- Top to bottom read
- Hostnames, IPv6 and IPv4 supported
- Authentication methods - trust, reject, md5, password, gss, sspi, krb5, ident, peer, pam, ldap, radius, bsd or cert

pg_hba.conf Example

#	TYPE	DATABASE	USER	ADDRESS	METHOD
#	IPv4	local connections:			
host		all	all	127.0.0.1/32	md5
#	IPv6	local connections:			
host		all	all	::1/128	md5
#	Allow replication connections from localhost, by a user with the replication privilege.				
#host	replication	postgres	postgres	127.0.0.1/32	md5
#host	replication	postgres	postgres	::1/128	md5

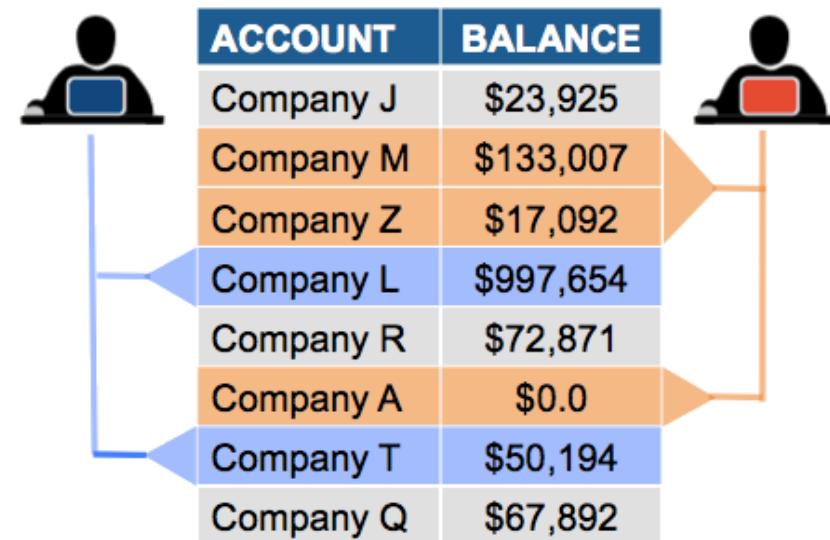
Authentication Problems

```
FATAL: no pg_hba.conf entry for host "192.168.10.23", user "edbstore", database "edbuser"  
FATAL: password authentication failed for user "edbuser"  
FATAL: user "edbuser" does not exist  
FATAL: database "edbstore" does not exist
```

- Self explanatory message is displayed
- Verify database name, username and Client IP in **pg_hba.conf**
- Reload Cluster after changing **pg_hba.conf**
- Check server log for more information

Row Level Security (RLS)

- GRANT and REVOKE can be used at table level
- PostgreSQL supports security policies for limiting access at row level
- By default, all rows of a table are visible
- Once RLS is enabled on a table, all queries must go through the security policy
- Security policies are controlled by DBA rather than application
- RLS offers stronger security as it is enforced by the database



Example - Row Level Security

- For example, to enable row level security for the table accounts :
 - Create the table first
 - => CREATE TABLE accounts (manager text, company text, contact_email text);
 - => ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

Example - Row Level Security (continued)

- To create a policy on the accounts table to allow the managers role to view the rows of their accounts, the CREATE POLICY command can be used:

```
=> CREATE POLICY account_managers ON accounts TO managers USING  
      (manager = current_user);
```

- To allow all users to view their own row in a user table, a simple policy can be used:

```
=> CREATE POLICY user_policy ON users USING (user =  
      current_user);
```

Application Access

- Application access is controlled by settings in both **postgresql.conf** and **pg_hba.conf**
- Set the following parameters in postgresql.conf:
 - listen_addresses
 - max_connections
 - superuser_reserved_connections
 - port
 - unix_socket_directory
 - unix_socket_group
 - unix_socket_permissions



DBA Managed with Centralized SQL Injection Protection

PREVENTION TECHNIQUES

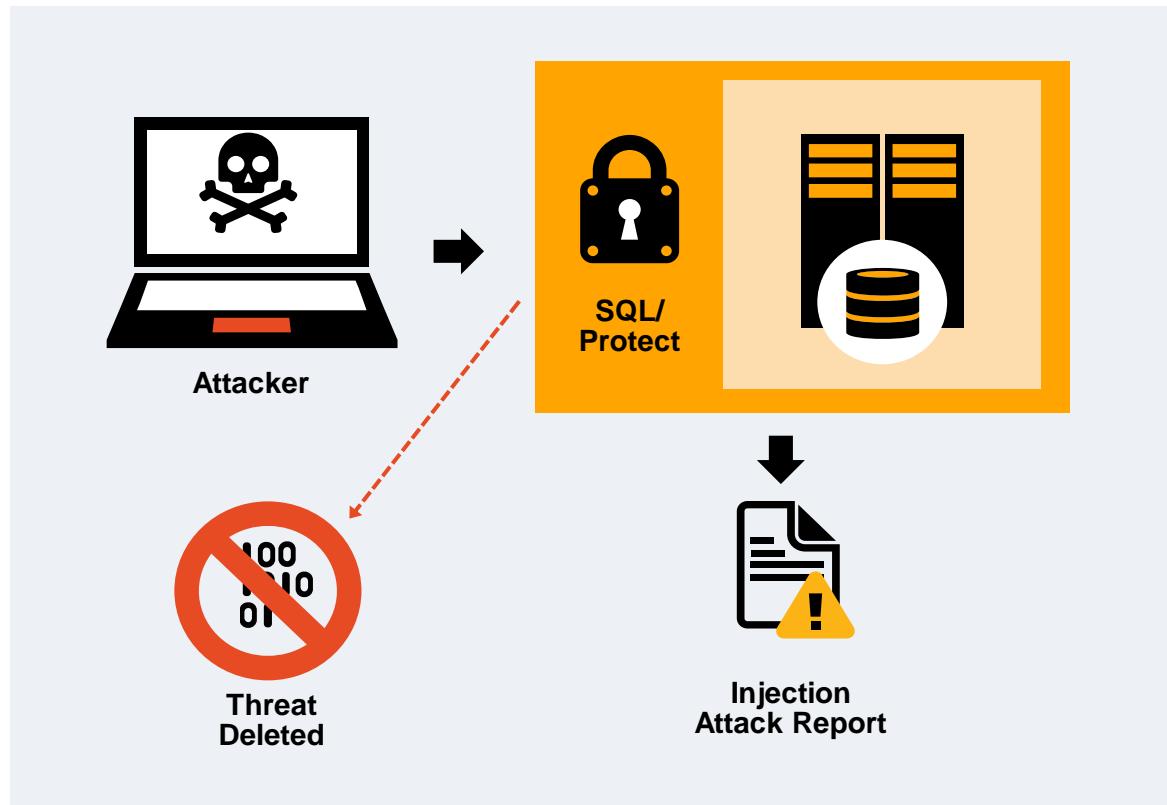
Unauthorized Relations

Utility Commands (e.g. DDL)

SQL Tautology

Unbounded DML

Preventing attacks is normally the responsibility of the application developer. But with SQL/Protect, DBAs can now provide another layer of protection to prevent corruption or co-opting of the database.



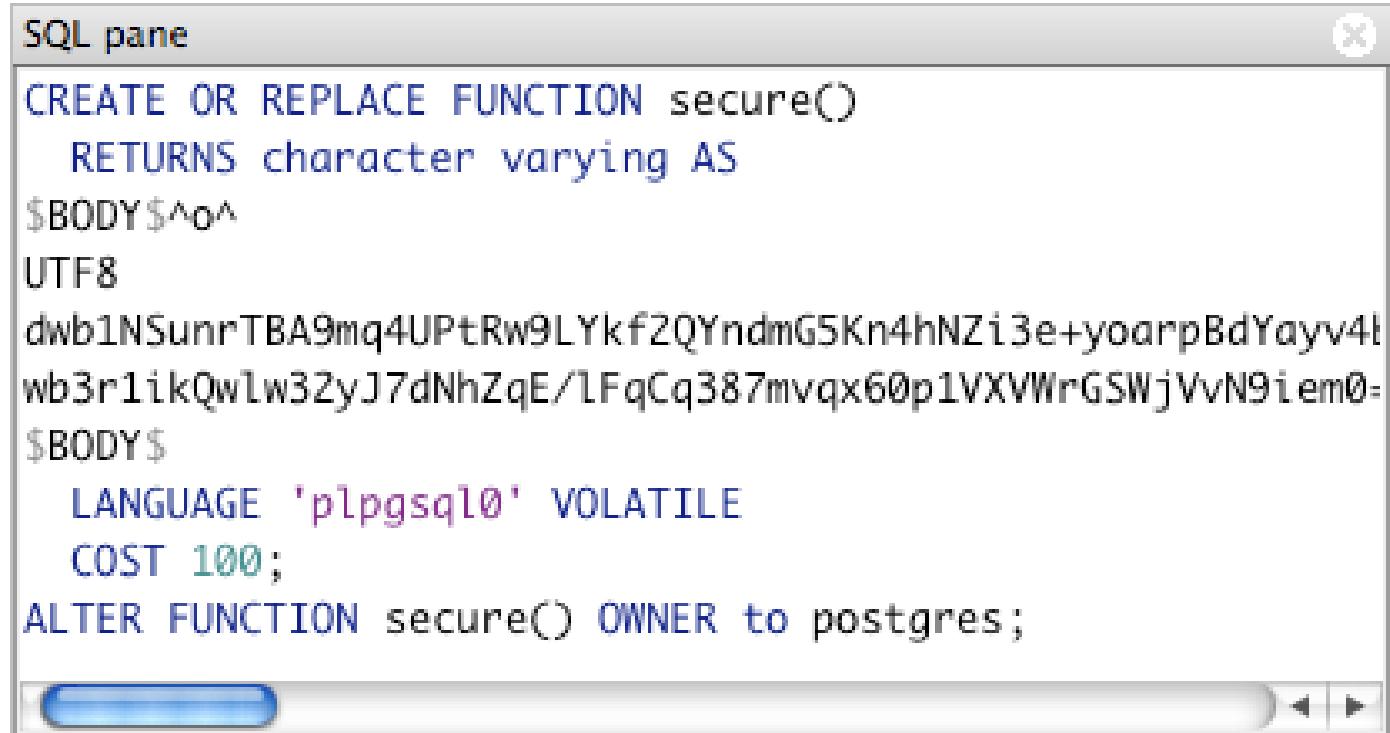


Included in EDB Postgres Enterprise subscriptions!

- Safeguards sensitive code from prying eyes inside your firewall
- Protects critical algorithms, processes, seed values and more
- Restricts access to intellectual property on customer sites
- Additional layer of security beyond standard user ACLs

SQL pane X

```
CREATE OR REPLACE FUNCTION secure()
RETURNS character varying AS
$BODY$^o^
UTF8
dwb1NSunrTBA9mq4UPtRw9LYkf2QYndmG5Kn4hNZi3e+yoarpBdYayv4t
wb3r1ikQwlw3ZyJ7dNhZqE/LFqCq387mvqx60p1VXVWrGSWjVvN9iem0=
$BODY$
LANGUAGE 'plpgsql0' VOLATILE
COST 100;
ALTER FUNCTION secure() OWNER to postgres;
```



Module Summary

- Authentication and Authorization
- Levels of Security
- pg_hba.conf File
- Object Ownership
- Application Access Parameters
- Protecting Against Injection Attacks with SQL/Protect
- Source Code Protection for Functions

Lab Exercise - 1

1. You are working as a PostgreSQL DBA. Your server box has 2 network cards with ip addresses 192.168.30.10 and 10.4.2.10. 192.168.30.10 is used for the internal LAN and 10.4.2.10 is used by the web server to connect users from an external network. Your server should accept TCP/IP connections both from internal and external users.
 - Configure your server to accept connections from external and internal networks.

Lab Exercise - 2

1. You are working as a PostgreSQL DBA. A developer showed you the following error:

```
psql: could not connect to server: Connection refused  
(0x0000274D/10061)
```

Is the server running on host 192.168.30.22" and accepting TCP/IP connections on port 5432?

2. Predict the problem and suggest the solution

Lab Exercise - 3

1. A new developer has joined the team, and his ID number is 89.
 - Create a new user by name dev89 and password password89.
 - Then assign the necessary privileges to dev89 so that he can connect to the edbstore database and view all tables.

Lab Exercise - 4

1. A new developer joins e-music corp. He has IP address 192.168.30.89. He is not able to connect from his machine to the PostgreSQL server and gets the following error on the server:

```
FATAL:  no pg_hba.conf entry for host "1.1.1.89", user "dev89",
database "edbstore", SSL off
```

2. Configure your server so that the new developer can connect from his machine.

Module 10

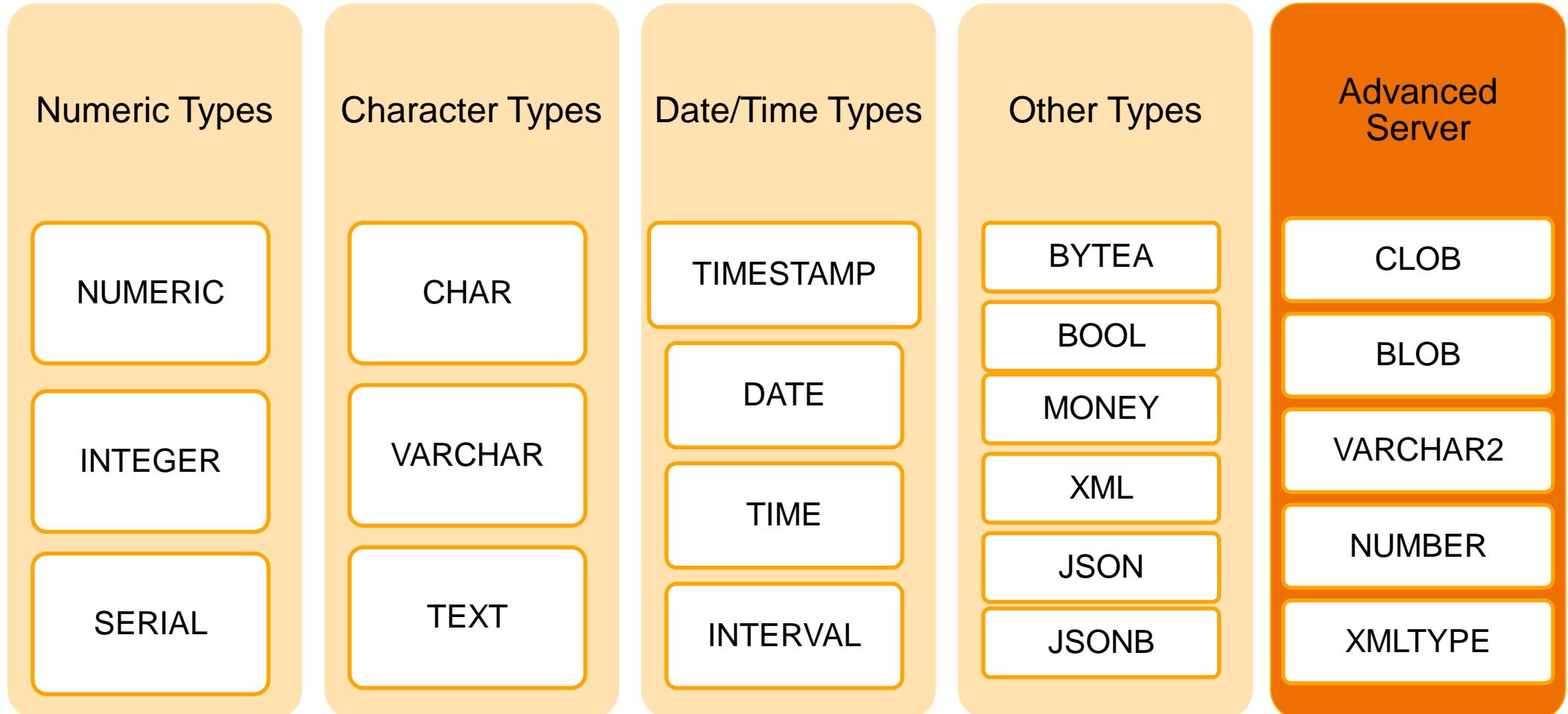
SQL Primer

Module Objectives

- Data Types
- Structured Query Language (SQL)
- Tables and Constraints
- Manipulating Data using INSERT,UPDATE and DELETE
- Creating Other Database Objects
 - Sequences, Views and Domains
- Introduction to various SQL functionalities:
 - Quoting
 - SQL Functions
 - Concatenation
 - Nested Queries
 - Joins
 - Materialized Views
 - Updatable Views
 - Indexes

Data Types

- Common Data Types:



Structured Query Language

Data Definition Language

- CREATE
- ALTER
- DROP
- TRUNCATE

Data Manipulation Language

- INSERT
- UPDATE
- DELETE
- SELECT

Data Control Language

- GRANT
- REVOKE

Transaction Control Language

- COMMIT
- ROLLBACK
- SAVEPOINT
- SET TRANSACTION

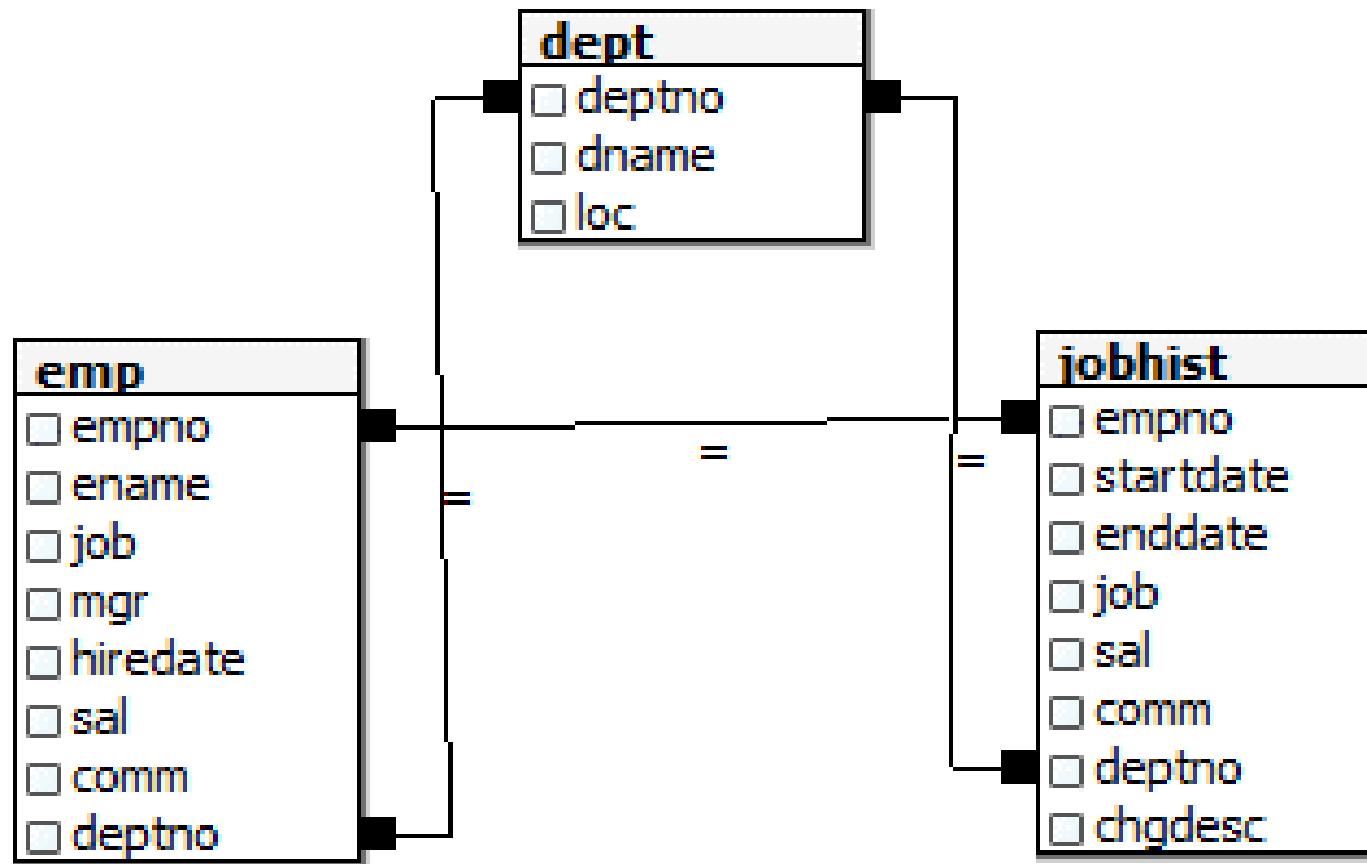
Tables

- A table is a named collection of rows
- Each table row has same set of columns
- Each column has a data type
- Tables can be created using the CREATE TABLE statement
- Syntax:

```
=> CREATE [TEMPORARY] [UNLOGGED] TABLE table_name  
  ( [column_name data_type [ column_constraint] ]  
  [ INHERITS ( parent_table) ]  
  [ TABLESPACE tablespace_name ]  
  [ USING INDEX TABLESPACE tablespace_name ]
```

Constraints

- Constraints are used to enforce data integrity.



Types of Constraints

- PostgreSQL supports different types of constraints:
 - NOT NULL
 - CHECK
 - UNIQUE
 - PRIMARY KEY
 - FOREIGN KEY
- Constraints can be defined at the column level or table level
- Constraints can be added to an existing table using the ALTER TABLE statement
- Constraints can be declared DEFERRABLE or NOT DEFERRABLE
- Constraints prevent the deletion of a table if there are dependencies

Example - CREATE TABLE

```
postgres=# CREATE TABLE products (productid integer PRIMARY KEY,  
postgres(# productname VARCHAR(30) NOT NULL,  
postgres(# productdesc VARCHAR(100));  
CREATE TABLE  
postgres=# █
```

Example - SERIAL Data Type

```
postgres=# CREATE TABLE games (gameid serial, name VARCHAR(30));
CREATE TABLE
postgres=# INSERT INTO games (name) VALUES('HALO');
INSERT 0 1
postgres=# INSERT INTO games (name) VALUES('CALL ON DUTY');
INSERT 0 1
postgres=# INSERT INTO games (name) VALUES('EVOLVE');
INSERT 0 1
postgres=# SELECT * FROM games;
 gameid |      name
-----+-----
      1 | HALO
      2 | CALL ON DUTY
      3 | EVOLVE
(3 rows)

postgres# ■
```

ALTER TABLE

- ALTER TABLE is used to modify the structure of a table
- Can be used to:
 - Add a new column
 - Modify an existing column
 - Add a constraint
 - Drop a column
 - Change schema
 - Rename a column, constraint or table
 - Enable and disable triggers and rules on a table
 - Change ownership
 - Enable or disable logging for a table

DROP TABLE

- The `DROP TABLE` statement is used to drop a table
- All data and structure is deleted
- Related indexes and constraints are dropped
- Syntax:

```
=> DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Inserting Data

- The `INSERT` statement is used to insert one or multiple rows in a table
- Syntax:

```
=> INSERT INTO table_name [ ( column_name [, ...] ) ]  
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...]  
    ) [, ...] | query }
```

Example - INSERT

```
postgres=# INSERT INTO products VALUES(101, 'XBOX CONSOLE', 'Xbox 360 was the first console  
to offer gaming in FULL HD 1080p');  
INSERT 0 1  
postgres=# INSERT INTO products(productid, productname) VALUES(102, 'XBOX ONE');  
INSERT 0 1  
postgres=# INSERT INTO products VALUES(103, 'XBOX Kinect', 'TBD'),(104, 'XBOX One Kinect','  
TBD');  
INSERT 0 2  
postgres=# TABLE products;  
productid | productname | productdesc  
-----+-----+-----  
----  
     101 | XBOX CONSOLE | Xbox 360 was the first console to offer gaming in FULL HD 10  
80p  
     102 | XBOX ONE  
     103 | XBOX Kinect | TBD  
     104 | XBOX One Kinect | TBD  
(4 rows)  
  
postgres=# █
```

Update Data

- The UPDATE statement is used to modify zero or more rows in a table
- The number of rows modified depends on the WHERE condition
- Syntax:

```
=> UPDATE [ ONLY ] table_name  
        SET column_name = { expression | DEFAULT }  
        [ WHERE condition]
```

Example - UPDATE

```
postgres=# UPDATE products SET productdesc='Offer motion detection for playing fun' WHERE productid IN (103, 104);
UPDATE 2
postgres=# \TABLE products;
   productid |    productname    |          productdesc
-----+-----+-----+
      101 | XBOX CONSOLE | Xbox 360 was the first console to offer gaming in FULL HD 1080p
      102 | XBOX ONE     |
      103 | XBOX Kinect   | Offer motion detection for playing fun
      104 | XBOX One Kinect | Offer motion detection for playing fun
(4 rows)

postgres=# █
```

How to Upsert in PostgreSQL

- An upsert is used to switch an insert to an update when a unique violation occurs.
- To do an upsert in PostgreSQL add the ON CONFLICT DO UPDATE clause to an INSERT statement
- Syntax:

```
=> INSERT INTO ... ON CONFLICT [ { (column_name, ...)  
    | ON CONSTRAINT constraint_name } ]  
    { DO NOTHING | DO UPDATE SET column_name = value } [ WHERE ... ]
```

Deleting Data

- The **DELETE** statement is used to delete one or more rows of a table
- Syntax:

```
=> DELETE FROM [ ONLY ] table_name  
      [ WHERE condition ]
```

Example - DELETE

```
postgres=# DELETE FROM products WHERE productid=104;
DELETE 1
postgres=# \TABLE products;
 productid | productname | productdesc
-----+-----+-----
 101 | XBOX CONSOLE | Xbox 360 was the first console to offer gaming in FULL HD 1080p
 102 | XBOX ONE
 103 | XBOX Kinect | Offer motion detection for playing fun
(3 rows)

postgres=# █
```

Views

- A View is a Virtual Table and can be used to hide complex queries
- Can also be used to represent a selected view of data
- Simple views are automatically updatable
- Allow views with updatable and non-updatable columns
- Views with updatable and non-updatable columns
- Views can be created using the CREATE VIEW statement
- Syntax:

```
=> CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ]
      [ WITH ( view_option_name [= view_option_value] [, ...] ) ]
      AS query
```

Example - VIEW

- View Example with updatable and non-updateable columns:

```
edbstore=> CREATE VIEW vw_emp_details AS SELECT ename, sal, sal*12 AS annualsalary FROM emp;
CREATE VIEW
edbstore=> UPDATE vw_emp_details set sal=sal+100;
UPDATE 14
edbstore=> UPDATE vw_emp_details set annualsalary=annualsalary+100;
ERROR:  cannot update column "annualsalary" of view "vw_emp_details"
DETAIL:  View columns that are not columns of their base relation are not updatable.
edbstore=>
```

Materialized Views

- Materialized Views
 - Provides a persistent snapshot of data for the view query
 - Snapshots can be refreshed
 - => CREATE MATERIALIZED VIEW
 - => REFRESH MATERIALIZED VIEW
 - => ALTER MATERIALIZED VIEW
 - => DROP MATERIALIZED VIEW

MATERIALIZED VIEW Example

```
edbstore=> CREATE MATERIALIZED VIEW vm_mat_cust AS SELECT * FROM customers;  
SELECT 20000  
edbstore=> CREATE UNIQUE INDEX idx_vm_mat_custid ON vm_mat_cust(customerid);  
CREATE INDEX  
edbstore=> REFRESH MATERIALIZED VIEW CONCURRENTLY vm_mat_cust;  
REFRESH MATERIALIZED VIEW  
edbstore=> █
```

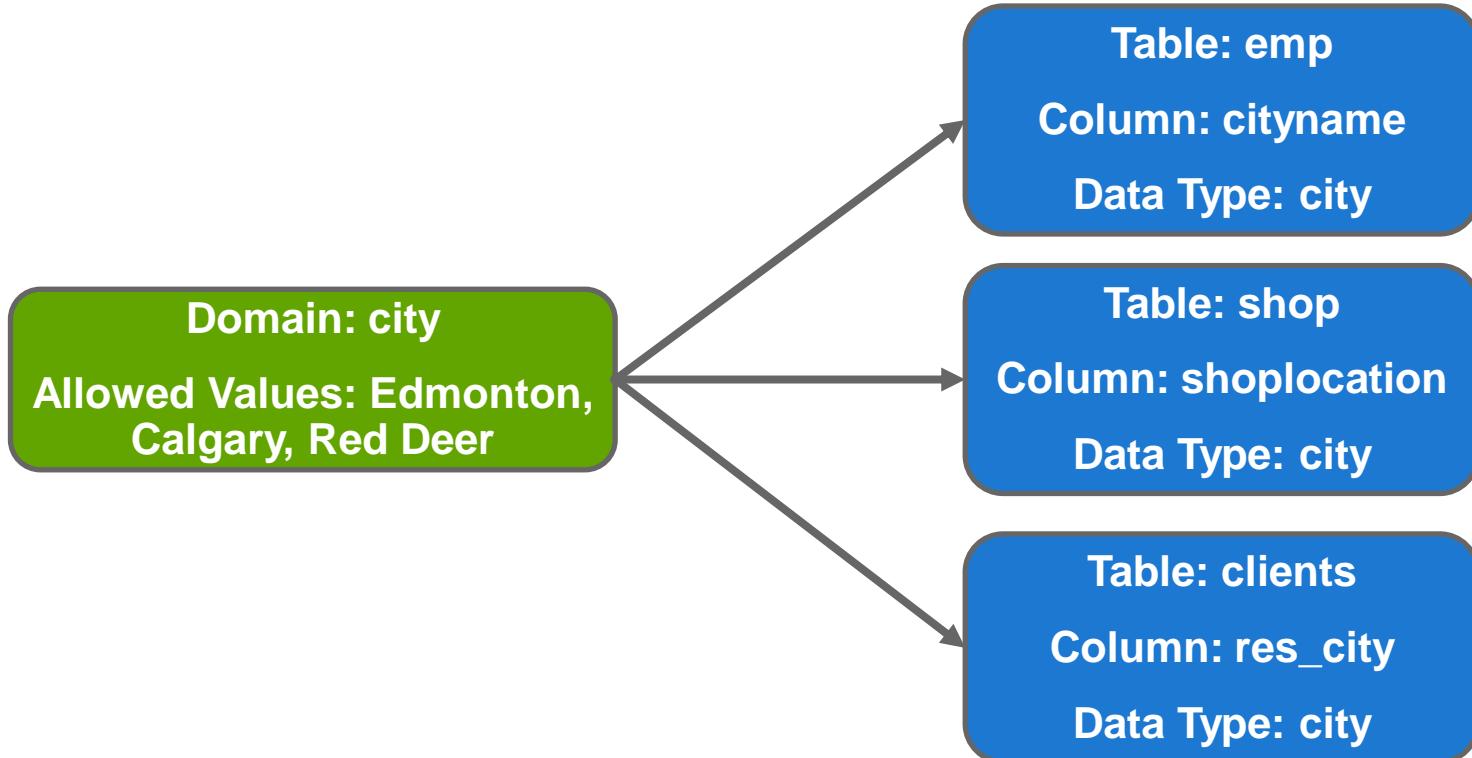
Sequences

- A sequence is used to automatically generate integer values that follow a pattern.
- A sequence has a name, start point and an end point.
- Sequence values can be cached for performance.
- Sequence can be used using CURRVAL and NEXTVAL functions.
- Syntax:

```
=> CREATE SEQUENCE name [ INCREMENT [ BY ] increment ]
      [ MINVALUE minvalue] [ MAXVALUE maxvalue]
      [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
      [ OWNED BY { table_name.column_name | NONE } ]
```

Domains

- A domain is a data type with optional constraints
- Domains can be used to create a data type which allows a selected list of values



Example : Domain

```
postgres=# CREATE DOMAIN city as varchar check (VALUE in ('Edmonton', 'Calgary',
'Red Deer'));
CREATE DOMAIN
postgres=# \dT
      List of data types
 Schema | Name | Description
-----+-----+-----
 public | city | 
(1 row)

postgres=# CREATE TABLE location (location_id numeric, city city);
CREATE TABLE
postgres=# INSERT INTO location VALUES (1, 'Calgary'), (2, 'Edmonton');
INSERT 0 2
postgres=# \T TABLE location;
   location_id |   city
-----+-----
        1 | Calgary
        2 | Edmonton
(2 rows)

postgres=# ■
```

Quoting

- Single quotes and dollar quotes are used to specify non-numeric values
 - Example:

```
'hello world'  
'2011-07-04 13:36:24'  
'{1,4,5}'  
$$A string "with" various 'quotes' in.$$  
$foo$A string with $$ quotes in $foo$
```
- Double quotes are used for names of database objects which either clash with keywords, contain mixed case letters, or contain characters other than a-z, 0-9 or underscore
 - Example:

```
SELECT * FROM "select"  
CREATE TABLE "HelloWorld" ...  
SELECT * FROM "Hi everyone and everything"
```

Using SQL Functions

- Can be used in SELECT statements and WHERE clauses
- Include
 - String Functions
 - Format Functions
 - Date and Time Functions
 - Aggregate Functions
- Example:

```
=> SELECT lower(name) FROM departments;
```

```
=> SELECT * FROM departments  
      WHERE lower(name) = 'development';
```

Format Functions

Function	Return Type	Description	Example
to_char(timestamp, text)	text	convert time stamp to string	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	convert interval to string	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	convert integer to string	to_char(125, '999')
to_char(double precision, text)	text	real/double precision to strconvert ing	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	convert numeric to string	to_char(-125.8, '999D99S')
to_date(text, text)	date	convert string to date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	convert string to numeric	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	convert string to time stamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(double precision)	timestamp with time zone	convert Unix epoch to time stamp	to_timestamp(1284352323)

Concatenating Strings

- Use two vertical bar symbols (||) to concatenate strings together
- Example:

```
=> SELECT 'Department ' || department_id || ' is: ' || name
      FROM departments;
```

Nested SELECT Statements

- WHERE clauses can contain SELECT statements
 - Example:
 - This will only return department names that have corresponding employees
- ```
=> SELECT dname FROM dept
 WHERE deptno IN (SELECT deptno FROM emp) ;
```

# Inner Joins

- Inner joins are the most common
- Only rows that have corresponding rows in the joined table are returned
- Example:

```
=> SELECT ename, dname
 FROM emp, dept
 WHERE emp.deptno = dept.deptno;
```

# Column and Table Aliases

- Used to make complex SQL statements easier to read
- Can also reduce the amount of typing required
- Example:

```
=> SELECT ename, dname
 FROM emp e, dept d
 WHERE e.deptno = d.deptno;
```

# Outer Joins

- Returns all rows even if there is no corresponding row in the joined table
- Add one of the following to the `FROM` clause:
  - `table LEFT [ OUTER ] JOIN table ON condition`
  - `table RIGHT [ OUTER ] JOIN table ON condition`
  - `table FULL [ OUTER ] JOIN table ON condition`
- PostgreSQL syntax example:

```
=> SELECT ename, dname
 FROM emp
 RIGHT OUTER JOIN dept
 ON (emp.deptno = dept.deptno);
```

# Indexes

- Indexes are a common way to enhance performance
- PostgreSQL supports several index types:
  - **B-tree** (default)
  - **Hash** – only used when the WHERE clause contains a simple comparison using the “=” operator (discouraged because they are not crash safe)
  - **Index on Expressions** – use when quick retrieval speed is needed on an often used expression. Inserts and updates will be slower.
  - **Partial Index** – Indexes only rows that satisfy the WHERE clause (the WHERE clause need not include the indexed column). A query must include the same WHERE clause to use the partial index
  - **Block Range Index (BRIN)** is designed for handling very large tables in which certain columns have some natural correlation with their physical location within the table.
  - **SP-GiST Indexes** - space-partitioned GiST supports partitioned search trees

# Example Index

- Syntax:
  - CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON table\_name [ USING method ]  
( { column\_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )  
[ WITH ( storage\_parameter = value [, ...] ) ]  
[ TABLESPACE tablespace\_name ]  
[ WHERE predicate ]

- Example:

```
edbstore=> CREATE INDEX idx_ename ON emp(ename);
CREATE INDEX
edbstore=> █
```

# Module Summary

- Data Types
- Structured Query Language (SQL)
- Tables and Constraints
- Manipulating Data using  
INSERT,UPDATE and DELETE
- Creating Other Database Objects -  
Sequences, Views and Domains
- Introduction to various SQL  
functionalities:
  - Quoting
  - SQL Functions
  - Concatenation
  - Nested Queries
  - Joins
  - Materialized Views
  - Updatable Views
  - Indexes

# Lab Exercise - 1

# Test your knowledge:

1. Initiate a psql session
  2. psql commands access the database True/False
  3. The following SELECT statement executes successfully:  
=> SELECT ename, job, sal AS Salary FROM emp; True/False
  4. The following SELECT statement executes successfully:  
=> SELECT \* FROM emp; True/False
  5. There are coding errors in the following statement. Can you identify them?  
=> SELECT empno, ename, sal \* 12 ANNUAL SALARY FROM emp;

# Lab Exercise - 2

## 1. Write a statement for the following:

- The HR department needs a report of all employees. Write a query to display the name, department number, and department name for all employees.
- Create a report to display employee names and employee numbers along with their manager's name and manager's number. Label the columns Employee, Emp#, Manager, and Mgr#, respectively.
- Create a report for the HR department that displays employee names, department numbers, and all the employees who work in the same department as a given employee. Give each column an appropriate label.

# Lab Exercise - 3

1. Write a query that displays the employee number and name of all employees who work in a department with any employee whose name contains an E. (Use a subquery.)
2. Update and delete data in the emp table.
  - Change the name of employee 7566 to Drexler.
  - Change the salary to \$1,000 for all employees who have a salary less than \$900.
  - Verify your changes to the table.
  - Delete MILLER from the emp table.

# Lab Exercise - 4

1. Create the `EMP2` table based on the structure of the `EMP` table. Include only the `empno`, `ename`, `sal`, and `deptno` columns. Name the columns in your new table `ID`, `FIRST_NAME`, `SALARY`, and `DEPTID`, respectively.
2. The staff in the HR department wants to hide some of the data in the `EMP` table. They want a view called `EMPVU` based on the employee numbers, employee names, and department numbers from the `EMP` table. They want the heading for the employee name to be `EMPLOYEE`.
3. Confirm that the view works. Display the contents of the `EMPVU` view.
4. Using your `EMPVU` view, write a query for the `SALES` department to display all employee names and department numbers.

# Lab Exercise - 5

1. You need a sequence that can be used with the primary key column of the `dept` table. The sequence should start at 60 and have a maximum value of 200. Have your sequence increment by 10. Name the sequence `dept_id_seq`.
2. To test your sequence, write a script to insert two rows in the `dept` table.
3. Create an index on the `deptno` column of the `dept` table.
4. Create and test a partial index.

# Module 11

## Backup, Recovery and PITR

# Module Objectives

- Backup Types
- SQL Dump Backups
- SQL Dump Options
- Handling Large Databases
- Restoring SQL Dumps
- Cluster Dump
- Offline Backups
- Recovering a Database using Offline Backups
- Online Directory Backups
  - Continuous Archiving
  - Backup with Low level API
  - How to use `pg_basebackup` for Online Backups
- Point-In Time Recovery Concepts
- Recovery Example
- Backup and Recovery Tool (BART)

# Types of Backup

- As with any database, PostgreSQL database should be backed up regularly

Logical Backups

**Database SQL Dumps**

- pg\_dump

**Database Cluster SQL Dump**

- pg\_dumpall

Physical  
Backups

**Offline File System Level Backups**

- Copy using OS commands

**Online File System Level Backups**

- Low Level API
- pg\_basebackup

# Database SQL Dump

- Generate a text file with SQL commands
- PostgreSQL provides the utility program **pg\_dump** for this purpose
- **pg\_dump** does not block readers or writers
- Dumps created by **pg\_dump** are internally consistent, that is, the dump represents a snapshot of the database as of the time **pg\_dump** begins running
- Syntax:

```
$ pg_dump [options] [dbname]
```

# pg\_dump Options

- **pg\_dump** Options:
  - a - Data only. Do not dump the data definitions (schema)
  - s - Data definitions (schema) only. Do not dump the data
  - n <schema> - Dump from the specified schema only
  - t <table> - Dump specified table only
  - f <path/file name.backup> - Send dump to specified file
  - Fp - Dump in plain-text SQL script (default)
  - Ft - Dump in tar format
  - Fc - Dump in compressed, custom format
  - Fd - Dump in directory format
  - j njobs - dump in parallel by dumping njobs tables simultaneously. Only supported with -Fd
  - v - Verbose option
  - o - use oids

# SQL Dump – Large Databases

- If operating systems have maximum file size limits, it cause problems when creating large pg\_dump output files
- Standard Unix tools can be used to work around this potential problem.
  - You can use your favorite compression program, for example gzip:  
`$ pg_dump dbname | gzip > filename.gz`
  - Also the split command allows you to split the output into smaller files:  
`$ pg_dump dbname | split -b 1m - filename`

# Restore – SQL Dump

- Backups taken using `pg_dump` with plain text format(Fp)
- Backups taken using `pg_dumpall`

`psql` client

- Backup taken using `pg_dump` with custom(Fc), tar(Ft) or director(Fd) formats
- Supports parallel jobs for during restore
- Selected objects can be restored

`pg_restore` utility

# pg\_restore Options

- pg\_restore Options:
  - l - Display TOC of the archive file
  - F [c|d|t] - Backup file format
  - d <database name> - Connect to the specified database. Also restores to this database if -C option is omitted
  - C - Create the database named in the dump file and restore directly into it
  - a - Restore the data only, not the data definitions (schema)
  - s - Restore the data definitions (schema) only, not the data
  - n <schema> - Restore only objects from specified schema
  - t <table> - Restore only specified table
  - v - Verbose option

# Entire Cluster – SQL Dump

- **pg\_dumpall** is used to dump an entire database cluster in plain-text SQL format
- Dumps global objects - user, groups, and associated permissions
- Use psql to restore
- Syntax:

```
$ pg_dumpall [options...] > filename.backup
```

# pg\_dumpall Options

- **pg\_dumpall** Options
  - a - Data only. Do not dump schema
  - s - Data definitions (schema) only
  - g - Dump global objects only - not databases
  - r - Dump only roles
  - c - Clean (drop) databases before recreating
  - O - Skip restoration of object ownership
  - x - do not dump privileges (grant/revoke)
  - disable-triggers, disable triggers during data-only restore
  - v - Verbose option

# Backup - File system level backup

- An alternative backup strategy is to directly copy the files that PostgreSQL uses to store the data in the database
- You can use whatever method you prefer for doing usual file system backups, for example:

```
$ tar -cf backup.tar /usr/local/pgsql/data
```

- The database server must be shut down in order to get a usable backup
- File system backups only work for complete backup and restoration of an entire database cluster
- File system snapshots work for live servers

# Backup - Continuous Archiving

- PostgreSQL maintains WAL files for all transactions in **pg\_xlog** directory
- PostgreSQL automatically maintains the WAL logs which are full and switched
- Continuous archiving can be setup to keep a copy of switched WAL Logs which can be later used for recovery
- It also enables online file system backup of a database cluster
- Requirements:
  - `wal_level` must be set to `replica`
  - `archive_mode` must be set to `on` (can be set to `always`)
  - `archive_command` must be set in **postgresql.conf** which archives WAL logs and supports PITR

# **pg\_receivexlog**

- **pg\_receivexlog** streams transaction log from a running PostgreSQL cluster
- This tool uses streaming replication protocol to stream transaction logs to a local directory
- These files can be then used for PITR
- Transaction logs are streamed in real time thus **pg\_receivexlog** can be used instead of archive command
- Example:

```
$ pg_receivexlog -h localhost -D /usr/local/pgsql/archive
```

# Continuous Archiving

Step 1 - Edit the **postgresql.conf** file and set the archive parameters:

```
wal_level=replica
```

```
archive_mode=on
```

- For Unix:

```
archive_command='cp -i %p /mnt/server/archivedir/%f </dev/null'
```

- For Windows:

```
archive_command='copy "%p" c:\\mnt\\server\\archivedir\\">%f"'
```

- %p is the absolute path of WAL otherwise you can define the path
- %f is a unique file name which will be created on above path

# Base Backup using Low Level API

## Step 2 - Make a base backup

- Connect using psql and issue the command  
=> `SELECT pg_start_backup('any useful label');`
- Use a standard file system backup utility to back up the /data subdirectory
- Connect using psql and issue the command  
=> `SELECT pg_stop_backup();`
- Continuously archive the WAL segment files

# Base Backup Using pg\_basebackup Tool

- **pg\_basebackup** can take a online base backup of a database cluster
- This backup can be used for PITR or Streaming Replication
- **pg\_basebackup** makes a binary copy of the database cluster files
- System is automatically put in and out of backup mode

# pg\_basebackup - Online Backup

- Steps require to take Base Backup:

- Modify **pg\_hba.conf**

```
host replication postgres [Ipv4 address of client]/32 trust
```

- Modify **postgresql.conf**

```
archive_command = 'cp -i %p /users/postgres/archive/%f'
```

```
archive_mode = on
```

```
max_wal_senders = 3
```

```
wal_keep_segments = 50
```

```
wal_level=replica
```

- Backup Command:

```
$ pg_basebackup [options] ..
```

# Options for pg\_basebackup

- Command line options for **pg\_basebackup**:
  - D <directory name> - Location of backup
  - F <p or t> - Backup files format. Plain(p) or tar(t)
  - R - write recovery.conf after backup
  - T OLDDIR=NEWDIR - relocate tablespace in OLDDIR to NEWDIR
  - x - include required WAL files in backup
  - z - enable gzip compression for files
  - Z level - Compression level
  - P - Progress Reporting
  - h host - host on which cluster is running
  - p port - cluster port
- To create a base backup of the server at localhost and store it in the local directory /usr/local/pgsql/backup

```
$ pg_basebackup -h localhost -D /usr/local/pgsql/backup
```

# Point-in-Time Recovery

- Point-in-time recovery (PITR) is the ability to restore a database cluster up to the present or to a specified point of time in the past
- Uses a full database cluster backup and the write-ahead logs found in the /pg\_xlog subdirectory
- Must be configured before it is needed (write-ahead log archiving must be enabled)

# Performing Point-in-Time Recovery

- Stop the server, if it's running
- If you have enough space keep a copy of data directory and transaction logs
- Remove all directories and files from the cluster data directory
- Restore the database files from your file system backup
- Verify the ownership of restored backup directories (must not be root)
- Remove any files present in `pg_xlog/`
- If you have any unarchived WAL segment files recovered from crashed cluster, copy them into `pg_xlog/`
- Create a recovery command file **recovery.conf** in the cluster data directory
- Start the server
- Upon completion of the recovery process, the server will rename **recovery.conf** to **recovery.done**

# Point-in-Time Recovery Settings

- Settings in the recovery.conf file:

```
$ restore_command(string)
```

- Unix:

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
```

- Windows:

```
restore_command = 'copy c:\\mnt\\server\\archivedir\\\"%f\" \"%p\"'
recovery_target_name(string)
recovery_target_time(timestamp)
recovery_target_xid(string)
recovery_target_inclusive(boolean)
recovery_target_timeline (string)
recovery_target_action (string)
```

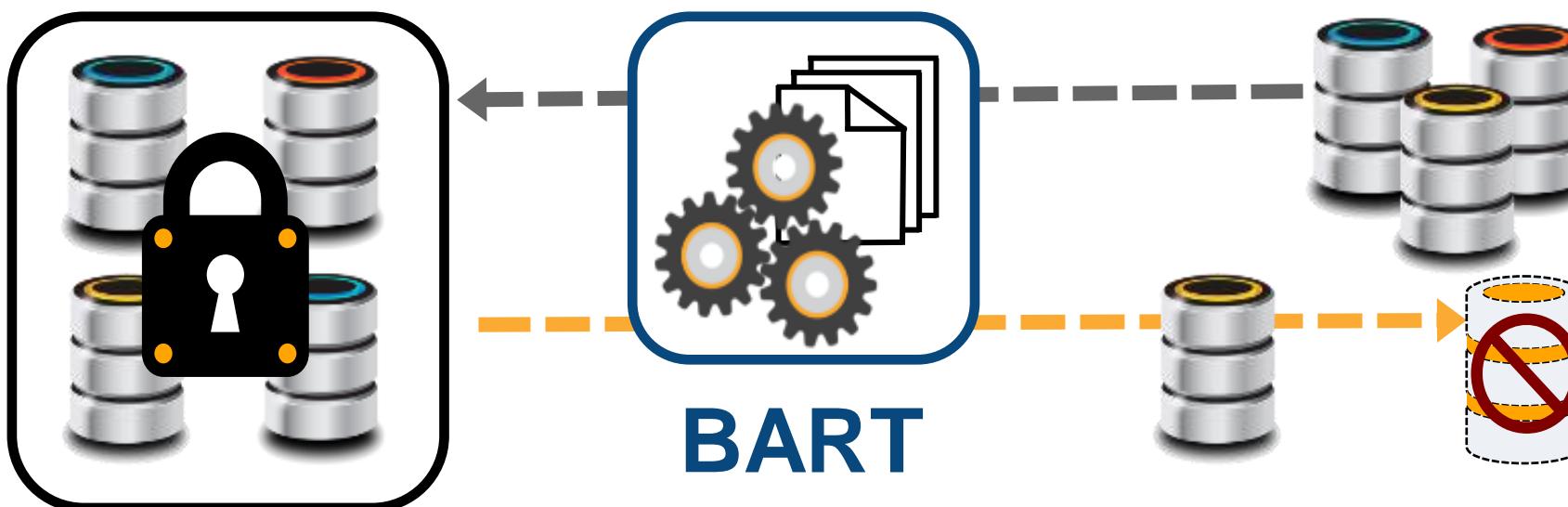
- Note: recovery\_target\_action parameter can have pause, promote or shutdown values. And pause is default

# EDB

## Backup and Recovery

*Simplifies and reduces errors with a system-wide catalog and command line tool that automates online backup and recovery across local and remote PostgreSQL and EDB Postgres Advanced Server. Retention policies conform to compliance needs and data governance.*

Simplifies and reduces  
errors in backup process



# Module Summary

- Backup Types
- SQL Dump Backups
- SQL Dump Options
- Handling Large Databases
- Restoring SQL Dumps
- Cluster Dump
- Offline Backups
- Recovering a Database using Offline Backups
- Online Directory Backups
  - Continuous Archiving
  - Backup with Low level API
  - How to use pg\_basebackup for Online Backups
- Point-In Time Recovery Concepts
- Recovery Example
- Backup and Recovery Tool (BART)

# Lab Exercise - 1

1. The `edbstore` website database is all setup and as a DBA you need to plan a proper backup strategy and implement it
  - As the root user, create a folder `/pgbackup` and assign ownership to the Postgres user using the `chown` utility or the Windows security tab in folder properties
  - Take a full database dump of the `edbstore` database with the `pg_dump` utility. The dump should be in plain text format
  - Name the dump file as `edbstore_full.sql` and store it in the `/pgbackup` directory

## Lab Exercise - 2

1. Take a dump of the `edbuser` schema from the `edbstore` database and name the file as **`edbstore_schema.sql`**
2. Take a data-only dump of the `edbstore` database, disable all triggers for a faster restore, use the `INSERT` command instead of `COPY`, and name the file as **`edbstore_data.sql`**
3. Take a full dump of `customers` table and name the file as **`edbstore_customers.sql`**

## Lab Exercise - 3

1. Take a full database dump of `edbstore` in compressed format using the `pg_dump` utility, name the file as **`edbstore_full_fc.dmp`**
2. Take a full database cluster dump using `pg_dumpall`. Remember `pg_dumpall` supports only plain text format; name the file **`edbdata.sql`**

# Lab Exercise - 4

In this exercise you will demonstrate your ability to restore a database.

1. Drop database `edbstore`.
2. Create database `edbstore` with owner `edbuser`.
3. Restore the full dump from **`edbstore_full.sql`** and verify all the objects and their ownership.
4. Drop database `edbstore`.
5. Create database `edbstore` with `edbuser` owner.
6. Restore the full dump from the compressed file **`edbstore_full_fc.dmp`** and verify all the objects and their ownership.

# Lab Exercise - 5

1. Create a directory /opt/arch or c:\arch and give ownership to the Postgres user.
2. Configure your cluster to run in archive mode and set the archive log location to be /opt/arch or c:\arch.
3. Take a full online base backup of your cluster in the /pgbackup directory using the **pg\_basebackup** utility.

# Lab Exercise - 6

1. A database cluster can encounter different types of failures. Recover your database from variety of simulated failures:
  - Recover from loss of the **postgresql.conf** file.
  - Recover from loss of the `inventory` table data file.
  - Recover from mistakenly dropped table `cust_hist`.

# Module 12

## Routine Maintenance Tasks

# Module Objectives

- Database Maintenance
- Maintenance Tools
- Optimizer Statistics
- Data Fragmentation
- Routine Vacuuming
- Vacuuming Commands
- Preventing Transaction ID Wraparound Failures
- Vacuum Freeze
- The Visibility Map
- Vacuumdb
- Autovacuuming
- Per Table Thresholds
- Routine Reindexing
- CLUSTER

# Database Maintenance

- Data files become fragmented as data is modified and deleted
- Database maintenance helps reconstruct the data files
- If done on time nobody notices but when not done everyone knows
- Must be done before you need it
- Improves performance of the database
- Saves database from transaction ID wraparound failures



# Maintenance Tools

- Maintenance thresholds can be configured using the PEM Client and PEM Server
- PostgreSQL maintenance thresholds can be configured in **postgresql.conf**
- Manual scripts can be written watch stat tables like `pg_stat_user_tables`
- Maintenance commands:
  - ANALYZE
  - VACUUM
  - CLUSTER
- Maintenance command `vacuumdb` can be run from OS prompt
- Autovacuum can help in automatic database maintenance

# Optimizer Statistics

- Optimizer statistics play a vital role in query planning
- Not updated in real time
- Collect information for relations including size, row counts, average row size and row sampling
- Stored permanently in catalog tables
- The maintenance command `ANALYZE` updates the statistics
- Thresholds can be set using PEM Client to alert you when statistics are not collected on time

# Example - Updating Statistics

```
postgres=# CREATE TABLE testanalyze(id integer, name varchar);
CREATE TABLE
postgres=# INSERT INTO testanalyze VALUES(generate_series(1,10000), 'Sample');
INSERT 0 10000
postgres=# SELECT relname, reltuples FROM pg_class WHERE relname='testanalyze';
 relname | reltuples
-----+-----
 testanalyze | 0
(1 row)

postgres=# ANALYZE testanalyze;
ANALYZE
postgres=# SELECT relname, reltuples FROM pg_class WHERE relname='testanalyze';
 relname | reltuples
-----+-----
 testanalyze | 10000
(1 row)
```

# Data Fragmentation and Bloat

- Data is stored in data file pages
- An update or delete of a row does not immediately remove the row from the disk page
- Eventually this row space becomes obsolete and causes fragmentation and bloating
- Set PEM Alert for notifications

# Routine Vacuuming

- Obsoleted rows can be removed or reused using vacuuming
- Helps in shrinking data file size when required
- Vacuuming can be automated using autovacuum
- The `VACUUM` command locks tables in access exclusive mode
- Long running transactions may block vacuuming thus it should be done during low usage times

# Vacuuming Commands

- When executed, the **VACUUM** command:
  - Can recover or reuse disk space occupied by obsolete rows
  - Updates data statistics
  - Updates the visibility map, which speeds up index-only scans
  - Protects against loss of very old data due to transaction ID wraparound
- The **VACUUM** command can be run in two modes:
  - VACUUM
  - VACUUM FULL

# Vacuum and Vacuum Full

- VACUUM
  - Removes dead rows and marks the space available for future reuse
  - Does not return the space to the operating system
  - Space is reclaimed if obsolete rows are at the end of a table
- VACUUM FULL
  - More aggressive algorithm compared to VACUUM
  - Compacts tables by writing a complete new version of the table file with no dead space
  - Takes more time
  - Requires extra disk space for the new copy of the table, until the operation completes

# VACUUM Syntax

```
postgres@pgcentos1:~
postgres=# \h VACUUM
Command: VACUUM
Description: garbage-collect and optionally analyze a database
Syntax:
VACUUM [({ FULL | FREEZE | VERBOSE | ANALYZE } [, ...])] [table_name
[(column_name [, ...])]]
VACUUM [FULL] [FREEZE] [VERBOSE] [table_name]
VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE [table_name [(column_name
[, ...])]]

postgres#
```

# Example - Vacuuming

```
postgres=#
postgres=# CREATE TABLE testvac (id numeric, name varchar);
CREATE TABLE
postgres=# INSERT INTO testvac values (generate_series(1,10000), 'Sample');
INSERT 0 10000
postgres=# SELECT pg_size.pretty(pg_relation_size('testvac'));
pg_size.pretty

440 kB
(1 row)

I
postgres=# UPDATE testvac set name='Sample';
UPDATE 10000
postgres=# SELECT pg_size.pretty(pg_relation_size('testvac'));
pg_size.pretty

872 kB
(1 row)
```

# Example – Vacuuming (continued)

```
postgres=# VACUUM testvac;
VACUUM
postgres=# UPDATE testvac set name='Sample';
UPDATE 10000
postgres=# SELECT pg_size.pretty(pg_relation_size('testvac'));
 pg_size.pretty

 872 kB
(1 row)

postgres=# VACUUM FULL testvac;
VACUUM
postgres=# SELECT pg_size.pretty(pg_relation_size('testvac'));
 pg_size.pretty

 440 kB
(1 row)

postgres=# █
```

# Preventing Transaction ID Wraparound Failures

- MVCC depends on transaction ID numbers
- Transaction IDs have limited size (32 bits at this writing)
- A cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound
- This causes a catastrophic data loss
- To avoid this, every table in the database must be vacuumed at least once for every two billion transactions

# Vacuum Freeze

- VACUUM FREEZE will mark rows as frozen
- Postgres reserves a special XID, FrozenTransactionId
- FrozenTransactionId is always considered older than every normal XID
- VACUUM FREEZE replaces transaction IDs with FrozenTransactionId, thus rows will appear to be “in the past”
- vacuum\_freeze\_min\_age controls when a row will be frozen
- VACUUM normally skips pages without dead row versions but some rows may need FREEZE
- vacuum\_freeze\_table\_age controls when whole table must be scanned

# The Visibility Map

- Each heap relation has a Visibility Map which keeps track of which pages contain only tuples
- Stored at `<relfilenode>_vm`
- Helps vacuum to determine whether pages contain dead rows
- Can also be used by index-only scans to answer queries
- VACUUM command updates the visibility map
- The visibility map is vastly smaller, so can be cached easily

# Vacuumdb Utility

- The VACUUM command has a command-line executable wrapper called vacuumdb
- vacuumdb can VACUUM all databases using a single command

**Usage:**

```
vacuumdb [OPTION]... [DBNAME]
```

**Options:**

|                                           |                                                                                                                |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| -a, --all                                 | vacuum all databases                                                                                           |
| -d, --dbname=DBNAME                       | database to vacuum                                                                                             |
| -e, --echo                                | show the commands being sent to the server                                                                     |
| -f, --full                                | do full vacuuming                                                                                              |
| -F, --freeze                              | freeze row transaction information                                                                             |
| -q, --quiet                               | don't write any messages                                                                                       |
| -t, --table='TABLE [(COLUMNS)]'           | vacuum specific table(s) only                                                                                  |
| -v, --verbose                             | write a lot of output                                                                                          |
| -V, --version                             | output version information, then exit                                                                          |
| -z, --analyze                             | update optimizer statistics                                                                                    |
| -Z, --analyze-only<br>--analyze-in-stages | only update optimizer statistics<br>only update optimizer statistics, in multiple<br>stages for faster results |
| -?, --help                                | show this help, then exit                                                                                      |

# Autovacuuming

- Highly recommended feature of Postgres
- It automates the execution of VACUUM, FREEZE and ANALYZE commands
- Autovacuum consists of a launcher and many worker processes
- A maximum of autovacuum\_max\_workers worker processes are allowed
- Launcher will start one worker within each database every autovacuum\_naptime seconds
- Workers check for inserts, updates and deletes and execute VACUUM and/or ANALYZE as needed
- track\_counts must be set to TRUE as autovacuum depends on statistics
- Temporary tables cannot be accessed by autovacuum

# Autovacuuming Parameters

```
postgres=# SELECT name, setting FROM pg_settings WHERE name LIKE 'autovacuum%';
 name | setting
-----+-----
 autovacuum | on
 autovacuum_analyze_scale_factor | 0.1
 autovacuum_analyze_threshold | 50
 autovacuum_freeze_max_age | 200000000
 autovacuum_max_workers | 3
 autovacuum_multixact_freeze_max_age | 400000000
 autovacuum_naptime | 60
 autovacuum_vacuum_cost_delay | 20
 autovacuum_vacuum_cost_limit | -1
 autovacuum_vacuum_scale_factor | 0.2
 autovacuum_vacuum_threshold | 50
 autovacuum_work_mem | -1
(12 rows)
```

- vacuum threshold = vacuum base threshold + vacuum scale factor \* number of tuples
- analyze threshold = analyze base threshold + analyze scale factor \* number of tuples

# Per-Table Thresholds

- Autovacuum workers are resource intensive
- Table-by-table autovacuum parameters can be configured for large tables
- Configure the following parameters using `ALTER TABLE` or `CREATE TABLE`:
  - `autovacuum_enabled`
  - `autovacuum_vacuum_threshold`
  - `autovacuum_vacuum_scale_factor`
  - `autovacuum_analyze_threshold`
  - `autovacuum_analyze_scale_factor`
  - `autovacuum_freeze_max_age`
  - `autovacuum_vacuum_cost_delay`
  - `autovacuum_vacuum_cost_limit`

# Routine Reindexing

- Indexes are used for faster data access
- UPDATE and DELETE on a table modify underlying index entries
- Indexes are stored on data pages and become fragmented over time
- REINDEX rebuilds an index using the data stored in the index's table
- Time required depends on:
  - Number of indexes
  - Size of indexes
  - Load on server when running command

# When to Reindex

- There are several reasons to use REINDEX:
  - An index has become "bloated", meaning it contains many empty or nearly-empty pages
  - You have altered a storage parameter (such as fillfactor) for an index
  - An index built with the CONCURRENTLY option failed, leaving an "invalid" index
- Syntax:

```
$ REINDEX [(VERBOSE)] { INDEX | TABLE | SCHEMA | DATABASE |
SYSTEM } name
```

# CLUSTER

- Sort data physically according to the specified index
- One time operations and changes are not clustered
- An ACCESS EXCLUSIVE lock is acquired
- When some data is accessed frequently and can be grouped using an index, CLUSTER is helpful
- CLUSTER lowers disk accesses and speeds up the query when accessing a range of indexed values
- Setting `maintenance_work_mem` to a large values is recommended before running CLUSTER
- Run ANALYZE afterwards

# Example - CLUSTER

```
postgres=# CREATE TABLE testcluster (id numeric, name varchar);
CREATE TABLE
postgres=# INSERT INTO testcluster VALUES (2,'B'),(1,'A'),(3,'A'),(5,'C'),(4,'B');
INSERT 0 5
postgres=# CREATE INDEX idx_id on testcluster(id);
CREATE INDEX
postgres=# CREATE INDEX idx_name on testcluster(name);
CREATE INDEX
postgres=# CLUSTER testcluster USING idx_id;
CLUSTER
postgres=# SELECT * FROM testcluster;
 id | name
----+-----
 1 | A
 2 | B
 3 | A
 4 | B
 5 | C
(5 rows)

postgres=# CLUSTER testcluster USING idx_name;
CLUSTER
postgres=# SELECT * FROM testcluster;
 id | name
----+-----
 1 | A
 3 | A
 2 | B
 4 | B
 5 | C
(5 rows)
```

# Module Summary

- Database Maintenance
- Maintenance Tools
- Optimizer Statistics
- Data Fragmentation
- Routine Vacuuming
- Vacuuming Commands
- Preventing Transaction ID Wraparound Failures
- Vacuum Freeze
- The Visibility Map
- Vacuumdb
- Autovacuuming
- Per Table Thresholds
- Routine Reindexing
- CLUSTER

# Lab Exercise - 1

1. While monitoring table statistics on the edbstore database, you found that some tables are not automatically maintained by autovacuum. You decided to perform manual maintenance on these tables. Write a SQL script to perform the following maintenance: Write a SQL script to perform the following .
  - Reclaim obsolete row space from the `customers` table.
  - Update statistics for `emp` and `dept` tables.
  - Mark all the obsolete rows in the `orders` table for reuse.
2. Execute the newly created maintenance script on edbstore database.

## Lab Exercise – 2

1. The composite index named `ix_orderlines_orderid` on (`orderid`, `orderlineid`) columns of the `orderlines` table is performing very slow. Write a statement to reindex this index for better performance.

# Module 13

## Data Dictionary

# Module Objectives

- The System Catalog Schema
- System Information Tables
- System Information Functions
- System Administration Functions
- System Information Views
- Oracle-like Dictionaries

# The System Catalog Schema

- Store information about table and other objects
- Created and maintained automatically in `pg_catalog` schema
- `pg_catalog` is always effectively part of the `search_path`
- Contains:
  - System Tables like `pg_class` etc.
  - System Function like `pg_database_size()` etc.
  - System Views like `pg_stat_activity`

# System Information Tables

- `\ds` in `psql` prompt will give you the list of `pg_*` tables and views
- This list is from `pg_catalog` schema
- Example:
  - `pg_tables` – list of tables
  - `pg_constraints` – list of constraints
  - `pg_indexes` – list of indexes
  - `pg_trigger` – list of triggers
  - `pg_views` – list of views

# More System Information Tables

- pg\_file\_settings – provides summary of the contents of the server configuration file
- pg\_policy – stores row level security for tables
- pg\_policies – provides access to useful information about each row-level security policy in the database

# System Information Functions

- `current_database`, `current_schema`, `inet_client_addr`, `inet_client_port`, `inet_server_addr`, `inet_server_port`, `pg_postmaster_start_time`, `version`
- `current_user` - user used for permission checking, must be called without ()
- `session_user` - normally user who started the session, but superusers can change
- `current_schemas(boolean)` - returns array of schemas in the search path, optionally including implicit schemas

# System Administration Functions

- `current_setting`, `set_config` - return or modify configuration variables
- `pg_cancel_backend` - cancel a backend's current query
- `pg_terminate_backend` – terminates backend process
- `pg_reload_conf` - reload configuration files
- `pg_rotate_logfile` - rotate the server's log file
- `pg_start_backup`, `pg_stop_backup` - used with Point In Time Recovery

# More System Administration Functions

- `pg_*_size` - disk space used by a tablespace, database, relation or total\_relation (**includes indexes and toasted data**)
- `pg_column_size` - bytes used to store a particular value
- `pg_size.pretty` - convert a raw size to something more human-readable
- `pg_ls_dir`, `pg_read_file`, `pg_stat_file` - file operation functions. Restricted to superuser use and only on files in the data or log directories
- `pg_blocking_pids()` - function to reliably identify which sessions block which others

# System Information Views

- pg\_stat\_activity - details of open connections and running transactions
- pg\_locks - list of current locks being held
- pg\_stat\_database - details of databases
- pg\_stat\_user\_\* - details of tables, indexes and functions
- pg\_stat\_archiver - status of the archiver process
- pg\_stat\_progress\_vacuum - provides progress reporting for VACUUM operations

# EDB Postgres Advanced Server Oracle-Like Dictionaries

- The sys schema contains Oracle compatible catalog views
- EDB Postgres Advanced Server provides DBA\_, ALL\_ and USER\_ dictionary views to view database object information

| View                | Description                                                   |
|---------------------|---------------------------------------------------------------|
| <code>user_*</code> | User's view (what is in your schema; what you own)            |
| <code>all_*</code>  | Expanded user's view (what you can access)                    |
| <code>dba_*</code>  | Database administrator's view (what is in everyone's schemas) |
| <code>edb\$</code>  | Performance-related data                                      |

# Module Summary

- The System Catalog Schema
- System Information Tables
- System Information Functions
- System Administration Functions
- System Information Views
- Oracle-like Dictionaries

# Lab Exercise - 1

1. You are working with different schemas in a database. You need to determine all the schemas in your search path. Write a query to find the list of schemas currently in your search path.

## Lab Exercise - 2

1. You need to determine the names and definitions of all of the views in your schema. Create a report that retrieves view information - the view name and definition text.

## Lab Exercise - 3

1. Create report of all the users who are currently connected. The report must display total session time of all connected users.
2. You found that a user has connected to the server for a very long time and have decided to gracefully kill its connection. Write a statement to perform this task.

## Lab Exercise - 4

1. Write a query to display the name and size of all the databases in your cluster. Size must be displayed using a meaningful unit.

# Module 14

# Moving Data

# Module Objectives

- Loading flat files
- Import and export data using COPY
- Examples of COPY Command
- Using COPY FREEZE for performance
- Introduction to EDB\*Loader for EDB Postgres Advanced Server

# Loading Flat Files into Database Tables

- A "flat file" is a plain text or mixed text file which usually contains one record per line
- PostgreSQL offers the following option to load flat files into a database table:
  - COPY Command
- EDB Postgres Advanced Server offers two options to load flat files into a database table:
  - EDB\*Loader
  - COPY Command

# The COPY Command

- `COPY` moves data between PostgreSQL tables and standard file-system files
- `COPY TO` copies the contents of a table or a query to a file
- `COPY FROM` copies data from a file to a table
- The file must be accessible to the server

# COPY TO

- Syntax of COPY TO Command:

```
=> COPY table_name [(column_name [, ...])]
 FROM { 'filename' | PROGRAM 'command' | STDIN }
 [[WITH] (option [, ...])]
```

where option can be one of:

```
FORMAT format_name
OIDS [boolean]
FREEZE [boolean]
DELIMITER 'delimiter_character'
NULL 'null_string'
HEADER [boolean]
QUOTE 'quote_character'
ESCAPE 'escape_character'
FORCE_QUOTE { (column_name [, ...]) | * }
FORCE_NOT_NULL (column_name [, ...])
FORCE_NULL (column_name [, ...])
ENCODING 'encoding_name'
```

# COPY FROM

- Syntax of COPY FROM Command:

```
=> COPY { table_name [(column_name [, ...])] | (query) }
 TO { 'filename' | PROGRAM 'command' | STDOUT }
 [[WITH] (option [, ...])]
```

where option can be one of:

```
FORMAT format_name
OIDS [boolean]
FREEZE [boolean]
DELIMITER 'delimiter_character'
NULL 'null_string'
HEADER [boolean]
QUOTE 'quote_character'
ESCAPE 'escape_character'
FORCE_QUOTE { (column_name [, ...]) | * }
FORCE_NOT_NULL (column_name [, ...])
FORCE_NULL (column_name [, ...])
ENCODING 'encoding_name'
```

# Example of COPY TO

```
edbstore=# COPY emp (empno,ename,job,sal,comm,hiredate) TO '/tmp/emp.csv' CSV HEADER;
edbstore=# \! cat /tmp/emp.csv

empno,ename,job,sal,comm,hiredate

7369,SMITH,CLERK,800.00,,17-DEC-80 00:00:00
7499,ALLEN,SALESMAN,1600.00,300.00,20-FEB-81 00:00:00
7521,WARD,SALESMAN,1250.00,500.00,22-FEB-81 00:00:00
7566,JONES,MANAGER,2975.00,,02-APR-81 00:00:00
7654,MARTIN,SALESMAN,1250.00,1400.00,28-SEP-81 00:00:00
7698,BLAKE,MANAGER,2850.00,,01-MAY-81 00:00:00
7782,CLARK,MANAGER,2450.00,,09-JUN-81 00:00:00
7788,SCOTT,ANALYST,3000.00,,19-APR-87 00:00:00
7839,KING,PRESIDENT,5000.00,,17-NOV-81 00:00:00
7844,TURNER,SALESMAN,1500.00,0.00,08-SEP-81 00:00:00
7876,ADAMS,CLERK,1100.00,,23-MAY-87 00:00:00
7900,JAMES,CLERK,950.00,,03-DEC-81 00:00:00
7902,FORD,ANALYST,3000.00,,03-DEC-81 00:00:00
7934,MILLER,CLERK,1300.00,,23-JAN-82 00:00:00
```

# Example of COPY FROM

```
edbstore=# CREATE TEMP TABLE empcsv (LIKE emp);
CREATE TABLE

edbstore=# COPY empcsv (empno, ename, job, sal, comm, hiredate)
edbstore-# FROM '/tmp/emp.csv' CSV HEADER;
```

## COPY

```
edbstore=# SELECT * FROM empcsv;
empno | ename | job | mgr | hiredate | sal | comm | deptno
-----+-----+-----+-----+-----+-----+-----+-----
 7369 | SMITH | CLERK | | 17-DEC-80 00:00:00 | 800.00 | |
 7499 | ALLEN | SALESMAN | | 20-FEB-81 00:00:00 | 1600.00 | 300.00 |
 7521 | WARD | SALESMAN | | 22-FEB-81 00:00:00 | 1250.00 | 500.00 |
 7566 | JONES | MANAGER | | 02-APR-81 00:00:00 | 2975.00 | |
 7654 | MARTIN | SALESMAN | | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |
 7698 | BLAKE | MANAGER | | 01-MAY-81 00:00:00 | 2850.00 | |
 7782 | CLARK | MANAGER | | 09-JUN-81 00:00:00 | 2450.00 | |
 7788 | SCOTT | ANALYST | | 19-APR-87 00:00:00 | 3000.00 | |
 7839 | KING | PRESIDENT | | 17-NOV-81 00:00:00 | 5000.00 | |
 7844 | TURNER | SALESMAN | | 08-SEP-81 00:00:00 | 1500.00 | 0.00 |
 7876 | ADAMS | CLERK | | 23-MAY-87 00:00:00 | 1100.00 | |
 7900 | JAMES | CLERK | | 03-DEC-81 00:00:00 | 950.00 | |
 7902 | FORD | ANALYST | | 03-DEC-81 00:00:00 | 3000.00 | |
 7934 | MILLER | CLERK | | 23-JAN-82 00:00:00 | 1300.00 | |
(14 rows)
```

# Example - COPY Command on Remote Host

- COPY command on remote host using psql

```
$ cat emp.csv | ssh 192.168.192.83 "psql -U edbstore edbstore -c
'copy emp from stdin;'"
```

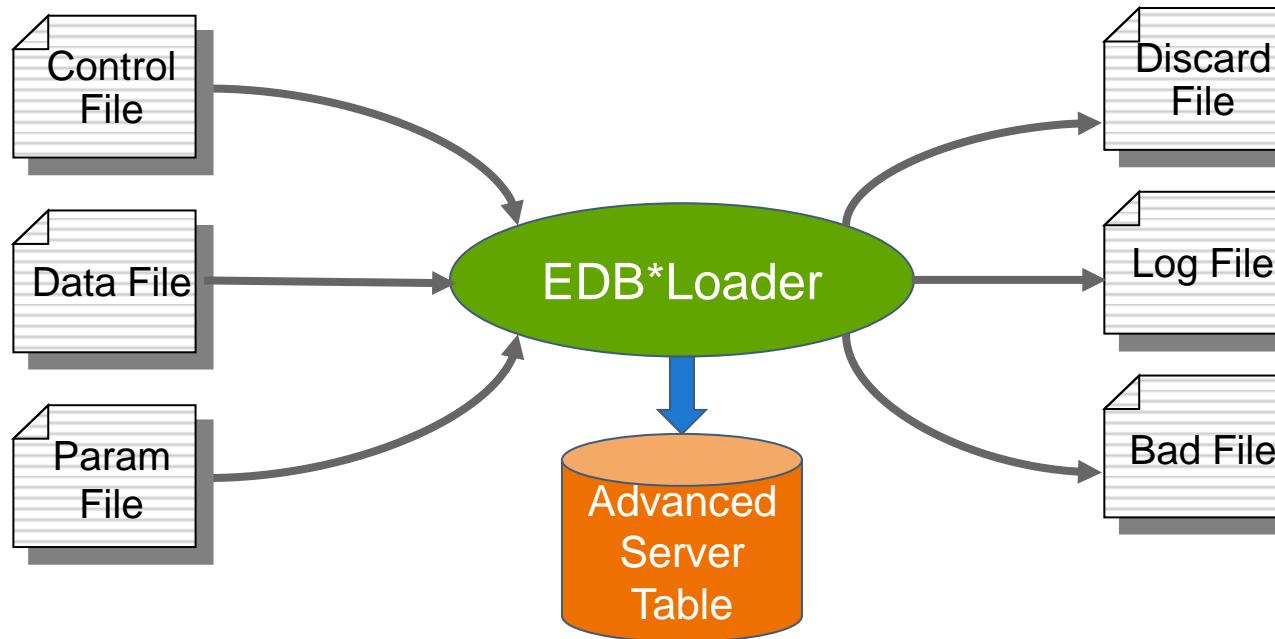
# COPY FREEZE

- **FREEZE** is a new option in the **COPY** statement
  - Add rows to a newly created table and freezes them
  - Table must be created or truncated in current subtransaction
  - Improves performance of initial bulk load
  - Does violate normal rules of MVCC
- **Usage:**

```
=> COPY tablename FROMfilename FREEZE;
```

# EDB\*Loader

- EDB\*Loader is a high-performance bulk data loader
- Supports Oracle SQL\*Loader data loading methods -
  - Conventional
  - Direct
  - Parallel



# Data Loading Methods

- Conventional path loading uses basic insert processing to add rows to the table
- Constraints, Indexes and triggers are enforced during Conventional path data loading
- Direct path loading is faster than conventional path loading, but is non-recoverable
- Direct path loading also requires removal of constraints and triggers from the table
- Conventional path data loading is slower than Direct path loading, but is fully recoverable
- A Parallel direct path load provides even greater performance

# Module Summary

- Loading flat files
- Import and export data using COPY
- Examples of COPY Command
- Using COPY FREEZE for performance
- Introduction to EDB\*Loader for EDB Postgres Advanced Server AS

# Lab Exercise – 1

1. Unload the `emp` table from the `edbstore` schema to a csv file, with column headers.
2. Create a `copyemp` table with the same structure as the `emp` table.
3. Load the csv file (from step 1) into the `copyemp` table.

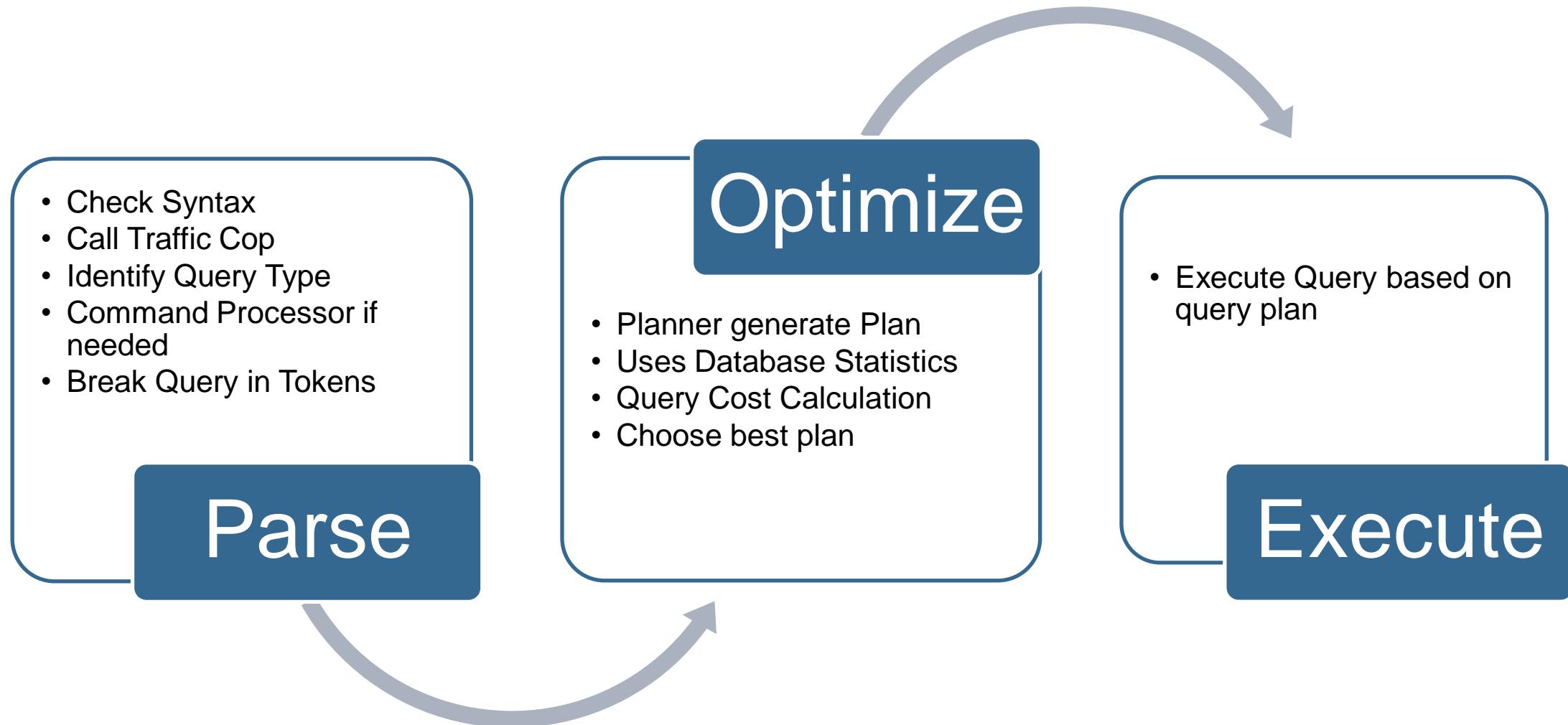
# Module 15

## SQL Tuning

# Module Objectives

- Statement Processing
- Common Query Performance Issues
- SQL Tuning Goals
- SQL Tuning Steps
  - Identify slow queries
  - Review the query execution plan
  - Optimizer statistics and behaviour
  - Restructure SQL statements
  - Indexes

# Statement Processing



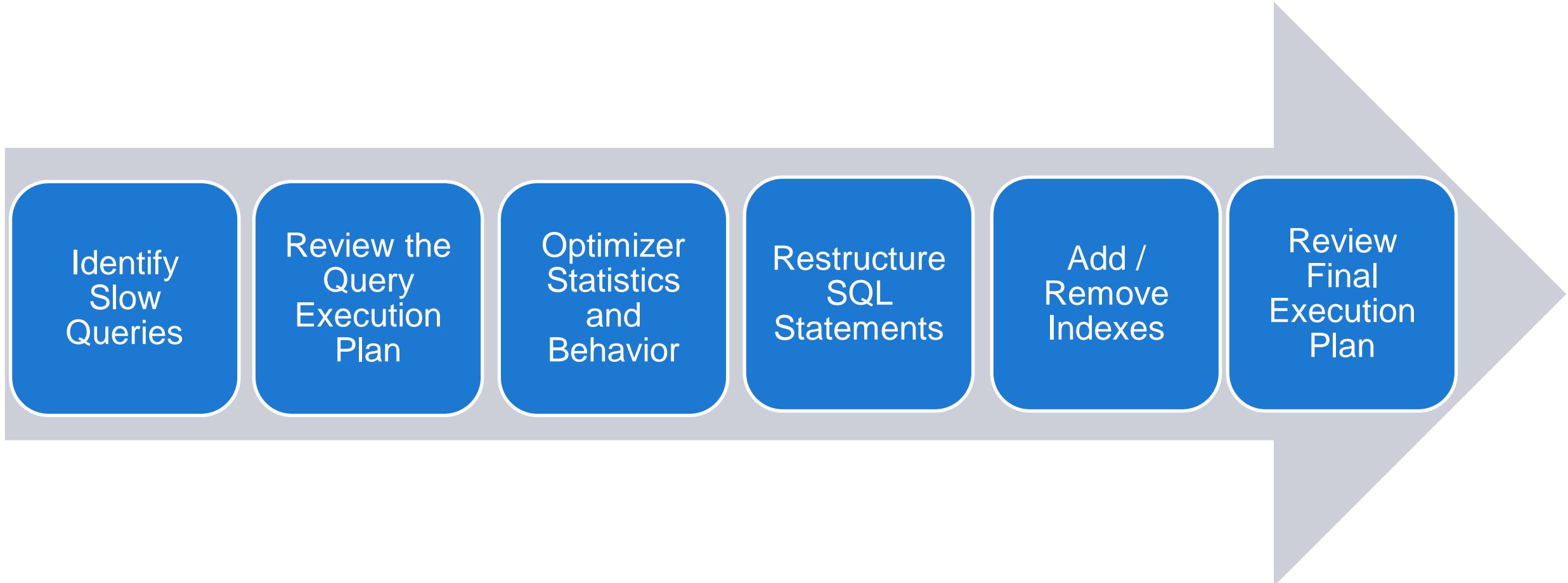
# Common Query Performance Issues

- Full tables scans
- Bad SQL
- Sorts using disk
- Join orders
- Old or missing statistics
- I/O issues
- Bad connection management

# SQL Tuning Goals

- Identify bad or slow SQL
- Find the possible performance issue in a query
- Reduce total execution time
- Reduce the resource usage of a query
- Determine most efficient execution plan
- Balance or parallelize the workload

# SQL Tuning Steps



## Step 1 - Identify Slow Queries

# Tracking Slow Queries

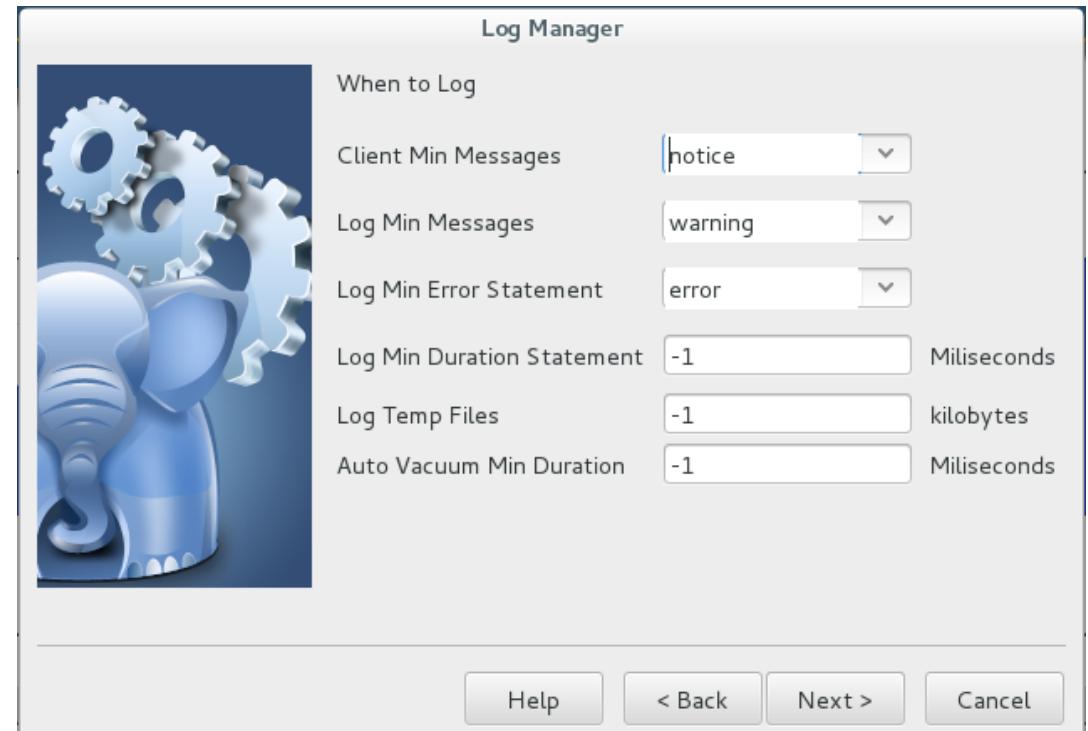
- `log_min_duration_statement` tracks slow running SQL
- PEM's Log Manager Wizard can configure log min duration statements
- Review log messages in Server Log Analysis Dashboard
- Use PEM's Postgres Log Analysis Expert Wizard to analyze log messages
- PEM's SQL Profiler can run a trace to find, troubleshoot, and optimize slow running SQL

# Log Slow Queries Parameter

- `log_min_duration_statement`
  - This statement sets a minimum statement execution time (in milliseconds) that causes a statement to be logged.
  - All SQL statements that run for the time specified or longer will be logged with their duration.
  - Enabling this option can be useful in tracking down non-optimized queries in your applications.

# PEM Log Manager

- Instance must be registered as a PEM-managed server
- Open Log Manager from the Management menu in the PEM Client
- Configure Log Min Duration Statement in When to Log Wizard
- Use the Server Log Analysis Dashboard to review the log files
- If required, analyze the log files using PEM Log Analysis Expert



# PEM Log Analysis Expert

Postgres Log Analysis Expert

Select the desired analysers to run:

⊕ ► **Analyzers**

- Summary Statistics
- Hourly DML Statistics
- DML Statistics
- DDL Statistics
- Commit/Rollback Statistics
- Checkpoint Statistics
- Log Event Statistics
- Log Statistics

**Select all** **Unselect all**

**Help** **< Back** **Next >** **Cancel**



## **Step 2 - Review the Query Execution Plan**

# Execution Plan

- An execution plan shows the detailed steps necessary to execute a SQL statement
- Planner is responsible for generating the execution plan
- The Optimizer determines the most efficient execution plan
- Optimization is cost-based, cost is estimated resource usage for a plan
- Cost estimates rely on accurate table statistics, gathered with ANALYZE
- Costs also rely on seq\_page\_cost, random\_page\_cost, and others
- The EXPLAIN command is used to view a query plan
- EXPLAIN ANALYZE is used to run the query to get actual runtime stats

# Execution Plan Components

- Execution Plan Components:
  - Cardinality - Row Estimates
  - Access Method - Sequential or Index
  - Join Method - Hash, Nested Loop etc.
  - Join Type, Join Order
  - Sort and Aggregates
- Syntax:  
= # EXPLAIN [ ( option [, ...] ) ] statement  
where option can be one of:
  - ANALYZE [ boolean ]
  - VERBOSE [ boolean ]
  - COSTS [ boolean ]
  - BUFFERS [ boolean ]
  - TIMING [ boolean ]
  - FORMAT { TEXT | XML | JSON | YAML }

# Explain Example - One Table

- Example:

```
=# EXPLAIN SELECT * FROM emp;
 QUERY PLAN

Seq Scan on emp (cost=0.00..1.14 rows=14 width=135)
```

- The numbers that are quoted by EXPLAIN are:
  - Estimated start-up cost
  - Estimated total cost
  - Estimated number of rows output by this plan node
  - Estimated average width (in bytes) of rows output by this plan node

# Explain Example - Multiple Tables

- Create the tables

```
=# CREATE TABLE city (cityid numeric(5) primary key, cityname
 varchar(30));
=# CREATE TABLE office (officeid numeric(5) primary key, cityid numeric(5)
 references city(cityid));
```

- Let's see the plan without data and updating statistics:

```
=# EXPLAIN ANALYZE SELECT city.cityname, office.officeid, office.cityid
 FROM city, office WHERE office.cityid = city.cityid;
```

- Output:

```
Hash Join (cost=25.30..71.16 rows=1510 width=102) (actual
time=0.002..0.002 rows=0 loops=1)
 Hash Cond: (office.cityid = city.cityid)
 -> Seq Scan on office (cost=0.00..25.10 rows=1510 width=24) (actual
time=0.001..0.001 rows=0 loops=1)
 -> Hash (cost=16.80..16.80 rows=680 width=90) (never executed)
 -> Seq Scan on city (cost=0.00..16.80 rows=680 width=90)
(never executed)
```

Planning time: 0.456 ms

Execution time: 0.067 ms

# Explain Example - Multiple Tables – Load and Analyze

- Load data:

```
=# INSERT INTO city
 values(1,'Edmonton') , (2,'Calgary') , (3,'Sherwood Park') , (4,'ST
Albert');
=# INSERT INTO office VALUES(generate_series(1,100),4);
=# INSERT INTO office VALUES(generate_series(101,200),3);
=# INSERT INTO office VALUES(generate_series(201,300),2);
=# INSERT INTO office VALUES(generate_series(301,400),1);
```

- Update the statistics for city and office table:

```
=# ANALYZE city;
=# ANALYZE office;
```

# Explain Example - Multiple Tables – Explain Analyze

- Plan:

```
=# EXPLAIN ANALYZE SELECT city.cityname, office.officeid,
 office.cityid FROM city, office WHERE office.cityid =
 city.cityid;
Hash Join (cost=1.09..12.59 rows=400 width=20) (actual
 time=0.057..0.770 rows=400 loops=1)
 Hash Cond: (office.cityid = city.cityid)
 -> Seq Scan on office (cost=0.00..6.00 rows=400 width=10)
 (actual time=0.012..0.128 rows=400 loops=1)
 -> Hash (cost=1.04..1.04 rows=4 width=15) (actual
 time=0.013..0.013 rows=4 loops=1)
 Buckets: 1024 Batches: 1 Memory Usage: 1kB
 -> Seq Scan on city (cost=0.00..1.04 rows=4 width=15)
 (actual time=0.002..0.005 rows=4 loops=1)
Planning time: 0.577 ms
Execution time: 0.893 ms
```

# PEM - Query Tool's Visual Explain

Query – edbstore on postgres@127.0.0.1:5432 \*

File Edit Query Favourites Macros View Help

SQL Editor Graphical Query Builder

Previous queries

```
SELECT
 emp.empno, emp.ename, emp.job, dept.deptno, dept.dname
FROM
 edbuser.emp, edbuser.dept
WHERE
 emp.deptno = dept.deptno;
```

Output pane

Data Output Explain Messages History

The visual explain diagram illustrates the execution plan for the given SQL query. It shows two input tables, 'emp' and 'dept', represented by grid icons. The 'emp' table has a single row selected, indicated by a blue arrow pointing to it. The 'dept' table also has a single row selected. These two rows are then processed by a 'Hash' operation, represented by a green icon. The output of the 'Hash' operation is then joined with the original 'emp' table via a 'Hash Join' operation, represented by a purple icon. The final result is a single row containing columns from both tables.

# Reviewing Explain Plans

- Examine the various costs at different levels in the explain plan
- Look for sequential scans on large tables
- All sequential scans are not inefficient
- Check whether a join type is appropriate for the number of rows returned
- Check for indexes used in the explain plan
- Examine the cost of sorting and aggregations
- Review whether views are used efficiently

## Step 3 - Optimizer Statistics and Behavior

# Optimizer Statistics

- The Postgres Optimizer and Planner use table statistics for generating query plans
- Choice of query plans are only as good as table stats
- Table statistics
  - Stored in catalog tables like `pg_class`, or `pg_stats`
  - Stores row sampling information

# Updating Planner Statistics

- Table Statistics
  - Absolutely critical to have accurate statistics to ensure optimal query plans
  - Are not updated in real time, so should be manually updated after bulk operations
  - Can be updated using ANALYZE command or OS command vacuumdb with -Z option
  - Stored in pg\_class and pg\_statistics
  - You can run the ANALYZE command from psql on specific tables and just specific columns
  - **Autovacuum** will run ANALYZE as configured
- Syntax for ANALYZE

```
=# ANALYZE [VERBOSE] [table_name [(column_name [, ...])]]
```

# Controlling Statistics Collection

- Postgres gathers and maintains table and column level statistics
- Statistics collection level can be controlled using:

```
=# ALTER TABLE <table> ALTER COLUMN <column> SET
STATISTICS <number>;
```
- The <number> can be set between 1 and 10000
- A higher <number> will signal the server to gather and update more statistics but may have slow autovacuum and analyze operation on stat tables. Higher numbers only useful for tables with large irregular data distributions

# TABLESAMPLE Clause

- Samples or retrieve random records from a table
- Useful when working with big data
- Syntax:

```
=> TABLESAMPLE sampling_method (argument [, ...]) [
 REPEATABLE (seed)]
```

- Supports SYSTEM and BERNOULLI sampling methods
  - SYSTEM uses random I/O, is page level and faster
  - BERNOULLI uses sequential I/O, scans full table picking tuples randomly

# Example - TABLESAMPLE Clause

- Create a table and insert data:

```
postgres=# CREATE TABLE ts_test (
 id SERIAL PRIMARY KEY,
 title TEXT);
postgres=# INSERT INTO ts_test (title)
postgres-# SELECT 'Record #' || i FROM generate_series(1,1000000) i;
```

- Create extension tsm\_system\_rows:

```
postgres=# CREATE EXTENSION tsm_system_rows;
postgres=
```

- EXPLAIN ANALYZE output:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM ts_test TABLESAMPLE SYSTEM_ROWS(10);
 QUERY PLAN
```

```


Sample Scan on ts_test (cost=0.00..4.10 rows=10 width=18) (actual time=0.018..
0.019 rows=10 loops=1)
 Sampling: system_rows ('10'::bigint)
Planning time: 0.227 ms
Execution time: 0.069 ms
(4 rows)
```

## Step 4 - Restructuring SQL Statements

# Restructure SQL Statements

- Rewriting inefficient SQL is often easier than repairing it
- Avoid implicit type conversion
- Avoid expressions as the optimizer may ignore the indexes on such columns
- Use equijoins wherever possible to improve SQL efficiency
- Try changing the access path and join orders with hints
- Avoid full table scans if using an index is more efficient
- The join order can have a significant effect on performance
- Use views or materialized views for complex queries

## Step 5 - Add / Remove Indexes

# General Indexing Guidelines

- Create indexes as and when needed
- Remove unused indexes
- Adding an index for one SQL can slow down other queries
- Verify index usage using the EXPLAIN command

# Indexes

- PostgreSQL provides several index types:
  - B-tree - equality and range queries on data that can be sorted  
*(Default index type)*
  - Hash - only simple equality comparisons. NOT CRASH SAFE!
  - GiST - can be used depending on the indexing strategy (the *operator class*)
  - SP-GiST - like GiST, offer an infrastructure that supports various kinds of searches
  - GIN - can handle values that contain more than one key, for example arrays
- Available v9.5 onwards
  - BRIN (Block Range Index) - accelerates scanning of large tables by maintaining summary data about block ranges. Very small index size compared to B-Tree, the tradeoff being you can't select a specific row so they are not useful for all data sets

# Multicolumn Indexes

- An index can be defined on more than one column of a table
- Currently, only the B-tree, GiST, and GIN index types support multicolumn indexes
- Example:
  - => CREATE INDEX test\_idx1 ON test (id\_1, id\_2);
    - This index will be used when you write:
      - => SELECT \* FROM test WHERE id\_1=1880 AND id\_2= 4500;
- Multicolumn indexes should be used sparingly

# Indexes and ORDER BY

- You can adjust the ordering of a B-tree index by including the options ASC, DESC, NULLS FIRST, and/or NULLS LAST when creating the index; for example:

```
=> CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
=> CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

- This will save sorting time spent by the query
- By default, B-tree indexes store their entries in ascending order with nulls last

# Unique Indexes

- Indexes can also be used to enforce uniqueness of a column's value or the uniqueness of the combined values of more than one column.
  - => CREATE UNIQUE INDEX name ON table (column [ , . . . ]);
    - Currently, only B-tree indexes can be declared unique.
- Postgres automatically creates a unique index when a unique constraint or primary key is defined for a table.

# Functional Indexes

- An index can be created on a computed value from the table columns.
  - For example:  
=> CREATE INDEX test1\_lower\_col1\_idx ON test1 (lower(col1));
- Index expressions are relatively expensive to maintain
- Indexes on expressions are useful when retrieval speed is more important than insertion and update speed
- They can also be created on user defined functions

# Partial Indexes

- A partial index is an index built over a subset of a table.
- The index contains entries only for those table rows that satisfy the predicate.
  - Example:

```
=> CREATE INDEX idx_test2 on test(id_1) where province='AB' ;
```

# BRIN Indexes

- New in PostgreSQL v9.5
- BRIN (Block Range Index) stores metadata on range (min and max) of pages
- Block Range are by default 1MB and `pages_per_range` storage parameter determines the size of the block range
- BRIN is small in size, easier to maintain and eliminates disk I/O by storing summary of data distribution on disk
- Designed to handle large table columns, which correlate to the physical location within the table

# Example - BRIN Indexes

```
postgres=# CREATE TABLE brin_example AS
 SELECT generate_series(1,100000000) AS id;
postgres=#
postgres=# CREATE INDEX brin_index on brin_example USING brin(id);
postgres=#
postgres=# CREATE INDEX btree_index ON brin_example(id);
postgres=#
postgres=# SELECT relname, pg_size.pretty(pg_relation_size(oid))
postgres-# FROM pg_class
postgres-# WHERE relname LIKE 'brin_%' OR relname='btree_index'
postgres-# ORDER BY relname;
 relname | pg_size.pretty
-----+-----
 brin_example | 3457 MB
 brin_index | 104 kB
 btree_index | 2142 MB
(3 rows)
```

# Examining Index Usage

- It is difficult to formulate a general procedure for determining which indexes to create.
  - Always run `ANALYZE` first.
  - Use real data for experimentation.
  - When indexes are not used, it can be useful for testing to force their use.
  - The `EXPLAIN ANALYZE` command can be useful here.

# PEM - Index Advisor

SQL Profiler - CoolTrace (20140209093120000000)

File Edit View Trace Help

Trace Data

Number of queries per page: 500 << < 1 OF 1 > >>

311  
312  
313  
314  
315  
316  
317

Default View  
Scheduled Traces  
Clean up  
Index Advisor  
Columns

Properties

SQL Query Metrics

```
SELECT *
FROM public.dept,
 public.emp
WHERE dept.deptno = emp.deptno;
```

Explain

Graphical Plan Text-based Plan

dept Hash Hash emp

The graphical explain plan illustrates the execution flow. It starts with the 'dept' table being scanned (indicated by a blue arrow pointing down). This table is then hashed. Simultaneously, the 'emp' table is also scanned (blue arrow down) and hashed. Finally, the two hashed tables are joined using a hash join operation (indicated by a green arrow connecting the two hash symbols).

Run the index advisor.

311 / 378

## **Step 6 - Review Final Execution Plan**

# Last Step - Review the Final Plan

- Check again for missing indexes
- Check table statistics are showing correct estimates
- Check large table sequential scans
- Compare the cost of first and final plans

# Module Summary

- Statement Processing
- Common Query Performance Issues
- SQL Tuning Goals
- SQL Tuning Steps
  - Identify slow queries
  - Review the query execution plan
  - Optimizer statistics and behavior
  - Restructure SQL statements
  - Indexes

# Lab Exercise - 1

1. You are working as a DBA. Users are complaining about long running queries and high execution times. Configure your database instance to log slow queries. Any query taking more than 5 seconds must be logged.
2. After logging slow queries you found following query taking longer than expected time:

```
=> SELECT * FROM customers JOIN orders USING(customerid);
```

3. View the explain plan of the above query.
4. View the execution time for the above query in psql terminal.

# Lab Exercise - 2

## 1. Create a table using following queries:

- CREATE TABLE lab\_test1 (c1 int4, c2 int4);
- INSERT INTO lab\_test1(c1, c2) values(generate\_series(1, 100000), 1);
- INSERT INTO lab\_test1(c1, c2) values(generate\_series(100001, 200000), 2);
- INSERT INTO lab\_test1(c1, c2) values(generate\_series(200001, 300000), 3);

## 2. Create three partial indexes on lab\_test1 table as following:

| Index Name | Predicate                    |
|------------|------------------------------|
| idx1_c1    | c1 between 1 and 100000      |
| idx2_c1    | c1 between 100001 and 200000 |
| idx3_c3    | c1 between 200001 and 300000 |

## Lab Exercise - 3

1. Detect the index usage for all the user indexes in edbstore database.
2. Reindex all the indexes.
3. Manually update the statistics for all the objects in edbstore database.

# **Module 16**

# **Performance Tuning**

# Module Objectives

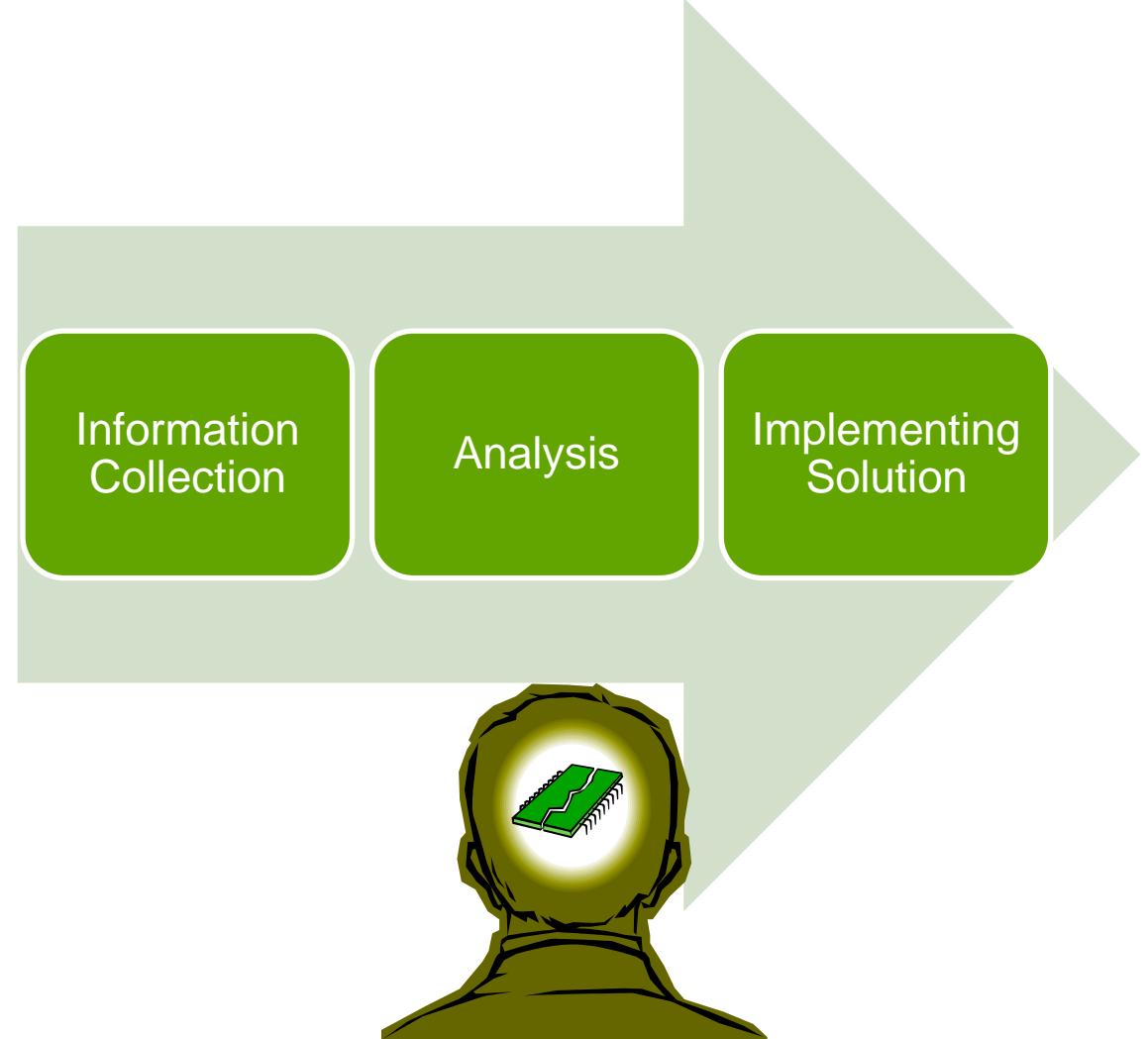
- Performance Tuning - Overview
- Performance Monitoring using PEM
- A Tuning Technique
- Operating System Considerations
- Server Parameter Tuning
- Memory Parameters
- Temporary File Parameters
- WAL Parameters

# Performance Tuning - Overview

- Performance Tuning is group of activities used to optimize a database
- Database Tuning is used to correct:
  - Poorly written SQL
  - Poor session management
  - Misconfigured database parameters
  - Operating system I/O issues
  - Database maintenance deficiencies

# The Performance Tuning Process

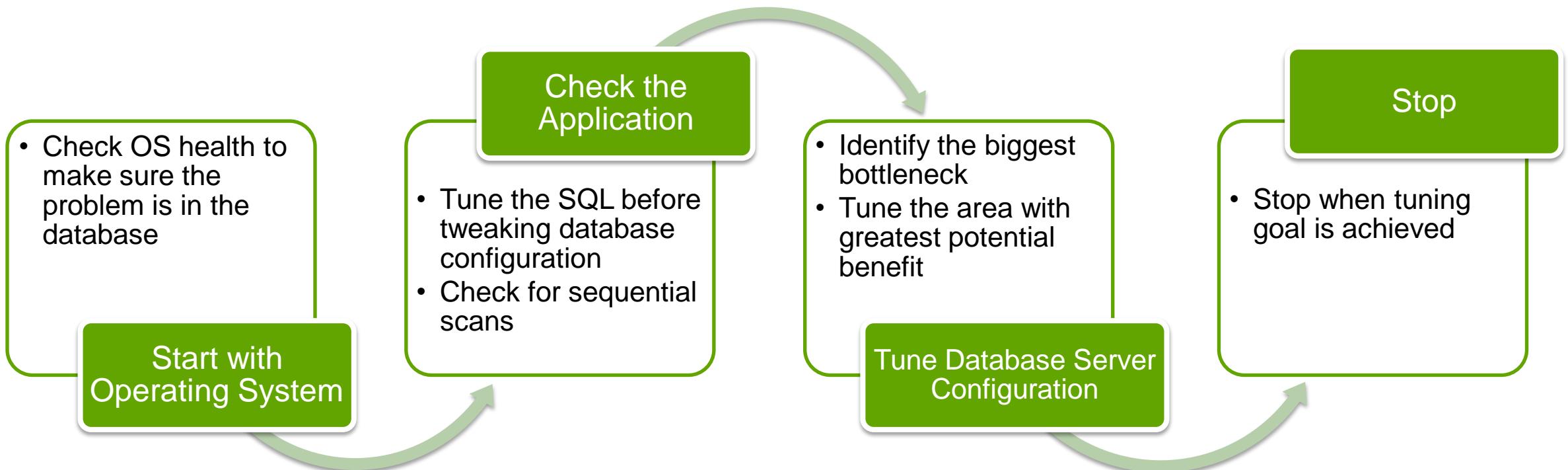
- Identify the information relevant to diagnosing performance problems
- Collect that information on a regular basis
- Expert analysis is needed to understand and correlate all the relevant statistics together
- Multiple solutions for different problems
- Use your own judgment to prioritize and quantify the solutions by impact



# Performance Monitoring Using PEM

- Postgres Enterprise Manager (PEM) simplifies collection of performance data
- Automatic analysis of performance and diagnostics data
- Performance Dashboards - view I/O, memory usage, session activity, and wait statistics
- SQL Profiler - optimize slow SQL
- Index Advisor
- Setup alerts and thresholds

# Tuning Technique



# Operating System Considerations

# Operating System Issues

- Memory
  - Increased memory demand may lead to 100% memory usage and swapping
  - Check memory usage
  - Solution is to reduce memory usage or increase system RAM
- CPU
  - CPU may be the bottleneck when load or process wait is high
  - Check CPU usage (%) for the database connections
  - Solution is to reduce CPU usage (%)
- Disk (I/O)
  - High wait times or request rates are symptoms of an I/O problem
  - Check for I/O spikes
  - Solution is to reduce demand or increase capacity
- PEM can be used to monitor memory, CPU, and I/O
- vmstat, sar and iostat can also be used

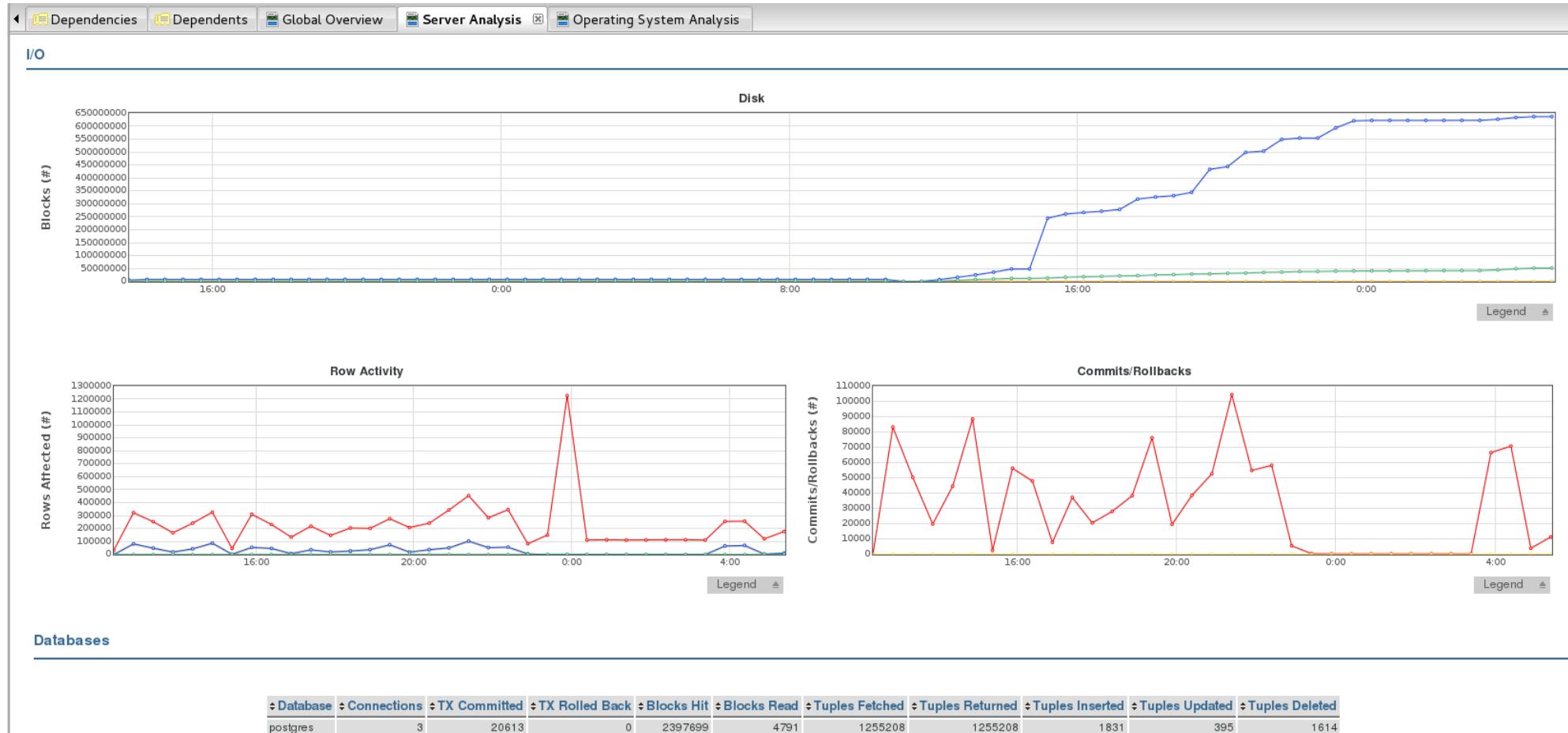
# PEM - Memory Graphs and Alerts



# PEM - CPU Graphs and Alerts



# PEM - I/O Graphs and Alerts



# Hardware Configuration

- Focus on disk speed and RAM over CPU speed
  - Like other RDBMS, Postgres is IO intensive
- Separate the transaction log and indexes
  - Put the database transaction log (`pg_xlog`) on a dedicated disk resource
  - Tablespaces can be used to create indexes on separate drives
- Setup disks to match speed requirements, not just size requirements
  - IOPS is just as important as GB when purchasing hardware
  - RAID 0 + 1 is optimal for speed and redundancy
  - Consider disk speed during failures (e.g. RAID-5 parity rebuild after failed disk)
  - Write caches must be persistent battery backed “Write-Back”, or you will get corruption

# OS Configuration

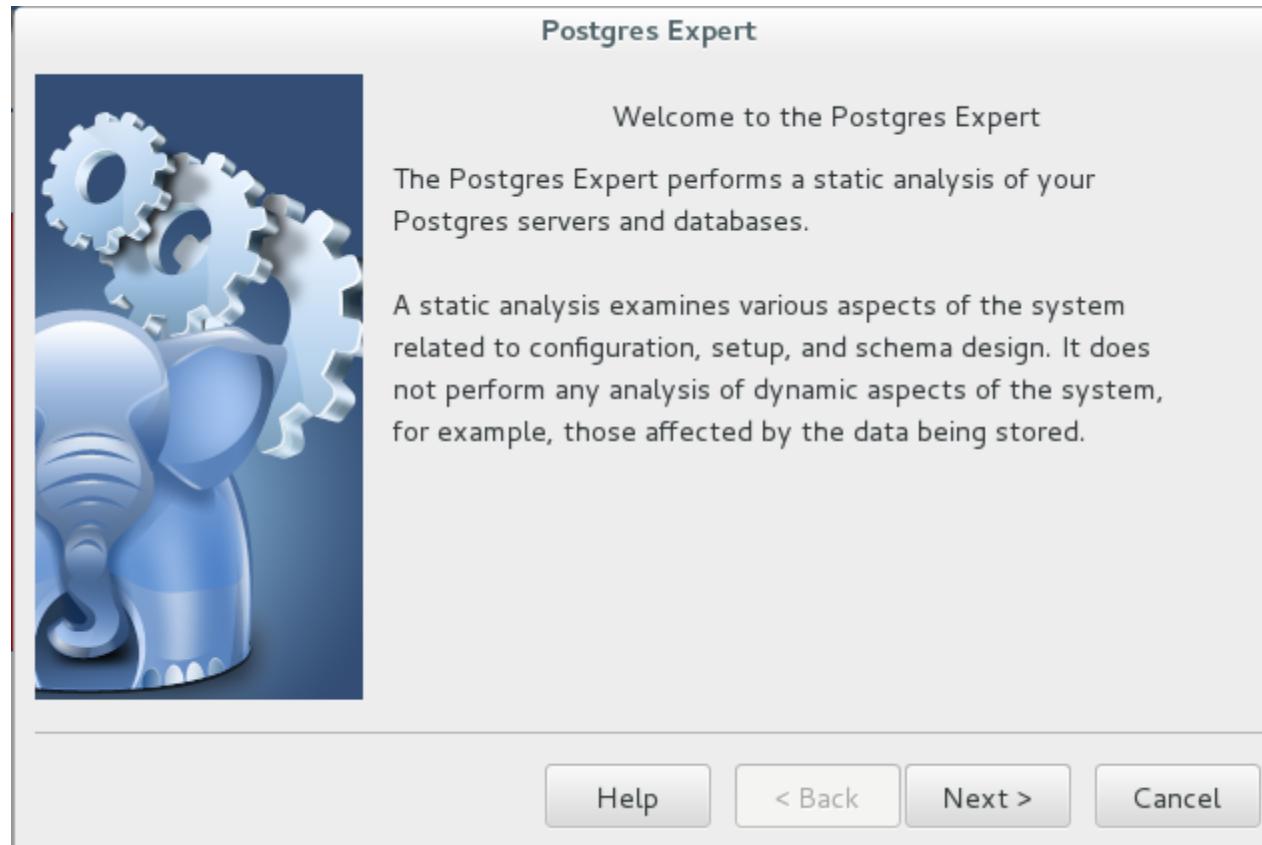
- The filesystem makes a difference
  - A journaling file system is not required for the transaction log
  - Multiple options are available for Linux:
    - EXT2 with sync enabled
    - EXT3 with write-back
    - EXT4 and XFS
    - Remote file systems are not recommended
- Choose the best based on better writes, recoverability, support from multiple vendors and reliability
- Eliminate unnecessary filesystem overhead, such as “noatime”
- Consider a virtual “ram” disk for stats\_temp\_directory

# Server Parameter Tuning

# Server Parameter Tuning

- Default **postgresql.conf** server parameters are configured for wide compatibility
- Server parameters must be tuned to optimal values for better performance
- Parameters can be evaluated using different methods and can be set permanently in the **postgresql.conf** file
- Some parameters require a restart
- Basic information needed for tuning but not limited to:
  - Database size
  - Largest table size
  - Type and frequency of queries
  - Available RAM
  - Number of concurrent connections

# PEM - Postgres Expert



# PEM - Postgres Expert Report

Properties Statistics Dependencies Dependents Database Analysis Postgres Expert Report

Postgres Expert Report

Generated: 2016-02-18 12:49:51

Jump to: Postgres Enterprise▼



**Summary:**

| Settings                  | Value |
|---------------------------|-------|
| Number of servers tested: | 1     |
| Number of rules checked:  | 31    |
| Number of High alerts:    | 1     |
| Number of Medium alerts:  | 3     |
| Number of Low alerts:     | 5     |

Server: Postgres Enterprise Manager Server (127.0.0.1:5432)

Advisor: Configuration Expert

| Rule                                               | Database | Severity |
|----------------------------------------------------|----------|----------|
| <a href="#">Check_checkpoint_completion_target</a> | -        | Medium   |
| <a href="#">Check_effective_cache_size</a>         | -        | Medium   |
| <a href="#">Check_effective_ioConcurrency</a>      | -        | Low      |
| <a href="#">Check_reducing_random_page_cost</a>    | -        | Low      |

Advisor: Schema Expert

| Rule                                                         | Database | Severity |
|--------------------------------------------------------------|----------|----------|
| <a href="#">Check data and transaction log on same drive</a> | -        | High     |
| <a href="#">Check for missing foreign key indexes</a>        | edbstore | Medium   |
| <a href="#">Check for missing primary keys</a>               | edbstore | Low      |
| <a href="#">Check for missing primary keys</a>               | testdb   | Low      |
| <a href="#">Check for missing primary keys</a>               | postgres | Low      |

Report generated by: Postgres Enterprise Manager™

# Connection Settings

- max\_connections
  - Sets the maximum number of concurrent connections
  - Each user connection has an associated user backend process on the server
  - User backend processes are terminated when a user logs off
  - Connection pooling can decrease the overhead on postmaster by reusing existing user backend processes

# Memory Parameters - shared\_buffers

- `shared_buffers`
  - Sets the number of shared memory buffers used by the database server
  - Each buffer is 8K bytes
  - Minimum value must be 16 and at least  $2 \times \text{max\_connections}$
  - 6% - 25% of available memory is a good general guideline
  - You may find better results keeping the setting relatively low and using the operating system cache more instead

# Memory Parameters - work\_mem

- `work_mem`
  - Amount of memory in KB to be used by internal sorts and hash tables before switching to temporary disk files
  - Minimum allowed value is 64 KB
  - It is set in KB
  - Increasing the `work_mem` often helps in faster sorting
  - `work_mem` settings can also be changed on a per session basis

# Memory Parameters - maintenance\_work\_mem

- maintenance\_work\_mem
  - Maximum memory in KB to be used in maintenance operations such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY
  - Minimum allowed value is 1024 KB
  - It is set in KB
  - Performance for vacuuming and restoring database dumps can be improved by increasing this value

# Memory Parameters - autovacuum\_work\_mem

- autovacuum\_work\_mem
  - Maximum amount of memory to be used by each autovacuum worker process
  - Default value is -1, indicates that maintenance\_work\_mem to be used instead

# Memory Parameters - huge\_pages

- huge\_pages
  - Enables/disables the use of huge memory pages
  - Valid values are `try` (the default), `on`, and `off`
  - May help in increasing performance by using smaller page tables thus less CPU time on memory management
  - This parameter is only supported on Linux

# Memory Settings for Planner

- `effective_cache_size`
  - Size of memory available for disk cache that is available to a single query
  - A higher value favors index scans
  - This parameter does not reserve kernel disk cache; it is used only for estimation purposes
  - $\frac{1}{2}$  of RAM is a conservative setting
  - $\frac{3}{4}$  of RAM is more aggressive
  - Find the optimal value looking at OS stats after increasing or decreasing this parameter

# Temporary Files

- `temp_file_limit`
  - Maximum amount of disk space that a session can use for temporary files
  - A transaction attempting to exceed this limit will be cancelled
  - Default is `-1` (no limit)
  - This setting constrains the total space used at any instant by all temporary files used by a given Postgres session

# WAL Parameters - wal\_level

- wal\_level
  - wal\_level determines how much information is written to the WAL
  - The default value is minimal, which writes only the information needed to recover from a crash or immediate shutdown
  - The value replica adds logging required for WAL archiving as well as information required to run read-only queries on standby server
  - The value logical is used to add information required for logical decoding
  - This parameter can only be set at server start

# WAL Parameters - wal\_buffers

- wal\_buffers
  - Number of disk-page buffers allocated in shared memory for WAL data
  - Each buffer is 8K bytes
  - Needs to be only large enough to hold the amount of WAL data created by a typical transaction since the WAL data is flushed out to disk upon every transaction commit
  - Minimum allowed value is 4
  - Default setting is -1 (auto-tuned)

# WAL Parameters – Checkpoints and max\_wal\_size

- Checkpoints
  - Writes the current in-memory modified pages (known as dirty pages) to the disk
  - An automatic checkpoint occurs each time the `max_wal_size` is reached
- `max_wal_size`
  - Maximum distance between automatic WAL checkpoints
  - Each log file segment is 16 megabytes
  - A checkpoint is forced when the `max_wal_size` is reached
  - A larger setting results in fewer checkpoints
  - The default is 1 GB
  - `max_wal_size` is soft limit and WAL size may exceed during heavy load, failed archive command, or high `wal_keep_segments`
  - Increase in `max_wal_size` also increases mean time to recover

# WAL Parameters - checkpoint\_timeout

- `checkpoint_timeout`
  - Maximum time between automatic WAL checkpoints in seconds before a checkpoint is forced
  - A larger setting results in fewer checkpoints
  - Range is 30 – 3600 seconds
  - The default is 300 seconds

# WAL Parameters - fsync

- `fsync`
  - Ensures that all the WAL buffers are written to the WAL logs at each COMMIT
  - When on, `fsync()` or other `wal_sync_method` is forked
  - Turning this off will be a performance boost but there is a risk of data corruption
  - Can be turned off during initial loading of a new database cluster from a backup file
  - `synchronous_commit = off` can provide similar benefits for noncritical transactions without any risk of data corruption

# `pg_test_fsync` Tool

- `wal_sync_method` is used for forcing WAL updates out to disk
- **`pg_test_fsync`** can determine the fastest `wal_sync_method` on your specific system
- **`pg_test_fsync`** reports average file sync operation time
- Diagnostic information for an identified I/O problem

# Parallel Query Scan Parameters

- PostgreSQL supports parallel execution of read-only queries
- Can be enable and configures using configuration parameters
  - **max\_parallel\_workers\_per\_gather(default 0)**: Enables parallel query scan
  - **parallel\_tuple\_cost( default 0.1)**: Estimated cost of transferring one tuple from a parallel worker process to another process
  - **parallel\_setup\_cost( default 1000.0)**: Estimates cost of launching parallel worker processes
  - **min\_parallel\_relation\_size( default 8MB)**: Relation larger than this parameter is considered for parallel scan
  - **force\_parallel\_mode( default off)**: Useful when testing parallel query scan even when there is no performance benefit

# Example – Parallel Query Scan

- Create table and insert data

```
postgres=# CREATE TABLE test (id int);
CREATE TABLE
postgres=# INSERT INTO test values(generate_series(1,100000000));
INSERT 0 100000000
postgres=# ANALYZE test;
ANALYZE
```

- Explain Analyze:

```
postgres=# SHOW max_parallel_workers_per_gather;
max_parallel_workers_per_gather

0
(1 row)

postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE id = 1;
 QUERY PLAN

Seq Scan on test (cost=0.00..1692478.40 rows=1 width=4) (actual time=0.711..15616.771 rows=1
loops=1)
 Filter: (id = 1)
 Rows Removed by Filter: 99999999
Planning time: 0.062 ms
Execution time: 15616.784 ms
(5 rows)
```

# Example – Parallel Query Scan (continued)

- Set `max_parallel_workers_per_gather` to 2 and EXPLAIN ANALYZE:

```
postgres=# SET max_parallel_workers_per_gather TO 2;
SET
postgres=# SHOW max_parallel_workers_per_gather;
 max_parallel_workers_per_gather

 2
(1 row)

postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE id = 1;
 QUERY PLAN

Gather (cost=1000.00..964311.60 rows=1 width=4) (actual time=6.906..15687.398 rows=1 loops=1)
 Workers Planned: 2
 Workers Launched: 2
 -> Parallel Seq Scan on test (cost=0.00..963311.50 rows=1 width=4) (actual time=10339.931..
.15566.332 rows=0 loops=3)
 Filter: (id = 1)
 Rows Removed by Filter: 33333333
Planning time: 0.047 ms
Execution time: 15687.620 ms
(8 rows)
```

# Loading a Table into Memory

- pg\_prewarm
  - Can be used to load relation data into either the operating system buffer cache or into the PostgreSQL buffer cache
  - Supports prefetch or read methods for OS cache and buffer method for shared\_buffers

```
postgres=# CREATE EXTENSION pg_prewarm;
postgres=#
postgres=# \df pg_prewarm

 List of functions
 Schema | Name | Result data type | Argument data types
 | | |
 | | | Type
-----+-----+-----+
-----+-----+
-----+-----+
 public | pg_prewarm | bigint | regclass, mode text DEFAULT 'buffer'::
 text, fork text DEFAULT 'main'::text, first_block bigint DEFAULT NULL::bigint, l
 ast_block bigint DEFAULT NULL::bigint | normal
(1 row)

postgres=# □
```

# Tips for Inserting Large Amounts of Data

- While inserting data using multiple inserts use BEGIN at the start and COMMIT at the end
- Use COPY to load all the rows in one command, instead of using a series of INSERT commands
- If you cannot use COPY, it might help to use PREPARE to create a prepared INSERT statement, and then use EXECUTE as many times as required
- If you are loading a freshly created table, the fastest method is to create the table, bulk load the table's data using COPY, then create any indexes needed for the table. EDB\*Loader of Advanced Server is 2x faster than COPY
- It might be useful to drop foreign key constraints, load the data, and then re-create the constraints

# Tips for Inserting Large Amounts of Data (Continued)

- Temporarily increasing the maintenance `work_mem` and `max_wal_size` configuration variables when loading large amounts of data can lead to improved performance
- Disable WAL Archival and Streaming Replication
- Triggers and Autovacuum can also be disabled
- Certain commands run faster if `wal_level` is minimal:
  - `CREATE TABLE AS SELECT`
  - `CREATE INDEX` (and variants such as `ALTER TABLE ADD PRIMARY KEY`)
  - `ALTER TABLE SET TABLESPACE`
  - `CLUSTER`
  - `COPY FROM`, when the target table has been created or truncated earlier in the same transaction

# Non-Durable Settings

- Durability guarantees the recording of committed transactions but adds significant overhead.
- Postgres can be configured to run without durability.
- Place the database cluster's data directory in a memory-backed file system (i.e. RAM disk).
- Turn off `fsync`; there is no need to flush data to disk.
- Turn off `full_page_writes`; there is no need to guard against partial page writes.
- Increase `max_wal_size` and `checkpoint_timeout`; this reduces the frequency of checkpoints.
- Turn off `synchronous_commit`; there might be no need to write the WAL to disk on every commit.

# Module Summary

- Performance Tuning - Overview
- Performance Monitoring using PEM
- A Tuning Technique
- Operating System Considerations
- Server Parameter Tuning
- Memory Parameters
- Temporary File Parameters
- WAL Parameters

# Lab Exercise - 1

1. Users are complaining about slower than normal performance on edbstore database
  - Tune **postgresql.conf** parameters for optimal performance based on the edbstore database size, largest table, and other necessary information collected from edbstore database
  - Change maximum concurrent connections on the cluster to 50

## Lab Exercise - 2

1. Write a statement to load customers table from `edbstore` database to the PostgreSQL buffer cache

## Lab Exercise - 3

1. `pg_test_fsync` can determine the fastest `wal_sync_method` on a specific system. Run this tool on your local machine and determine the best `wal_sync_method` settings for your system.

# **Module 17**

# **Streaming Replication**

# Module Objectives

- Data Replication
- Data Replication in PostgreSQL
- Sync or Async
- Log-Shipping Standby Servers
- Log-Shipping Architecture
- Streaming Replication
- Hot Streaming Architecture
- Cascading Replication
- Setup Replication Using Archive
- Setup Streaming Replication
  - Prepare the Primary Server
  - Synchronous Streaming Replication Setup
  - Configure Authentication
  - Take a Full Backup of Primary Server
  - Setting up the Standby Server
  - Adding Cascading Replicated Standby Server
  - Monitoring Hot Standby
  - Recovery Control Functions

# Data Replication

- Replication is the process of copying data and changes to a secondary location for data safety and availability
- Data loss can occur due to several reasons
- Replication is aimed towards availability of the data when a primary source goes offline
- Data can be recovered from backup but downtimes are costly
- Replication aims towards lowering downtime
- Failovers can be configured to such a level where application may not notice the primary source is offline



# Data Replication in PostgreSQL

- Data Replication in PostgreSQL can be achieved by implementing:
  - Log-Shipping Standby Servers
  - Streaming Replication
  - xDB Replication
- Failovers can be automated using:
  - EDB Failover Manager for streaming replicated servers
  - pgpool-II

# Asynchronous Replication

- Log shipping is only asynchronous
- PostgreSQL streaming replication is asynchronous by default but can be configured as synchronous
- Asynchronous
  - Disconnected architecture
  - Transaction is committed on primary and flushed to WAL segment
  - Later transaction is transmitted to standby server(s) using stream
  - In case of log shipping, replication happens when the WAL segment is archived
  - Some data loss is possible

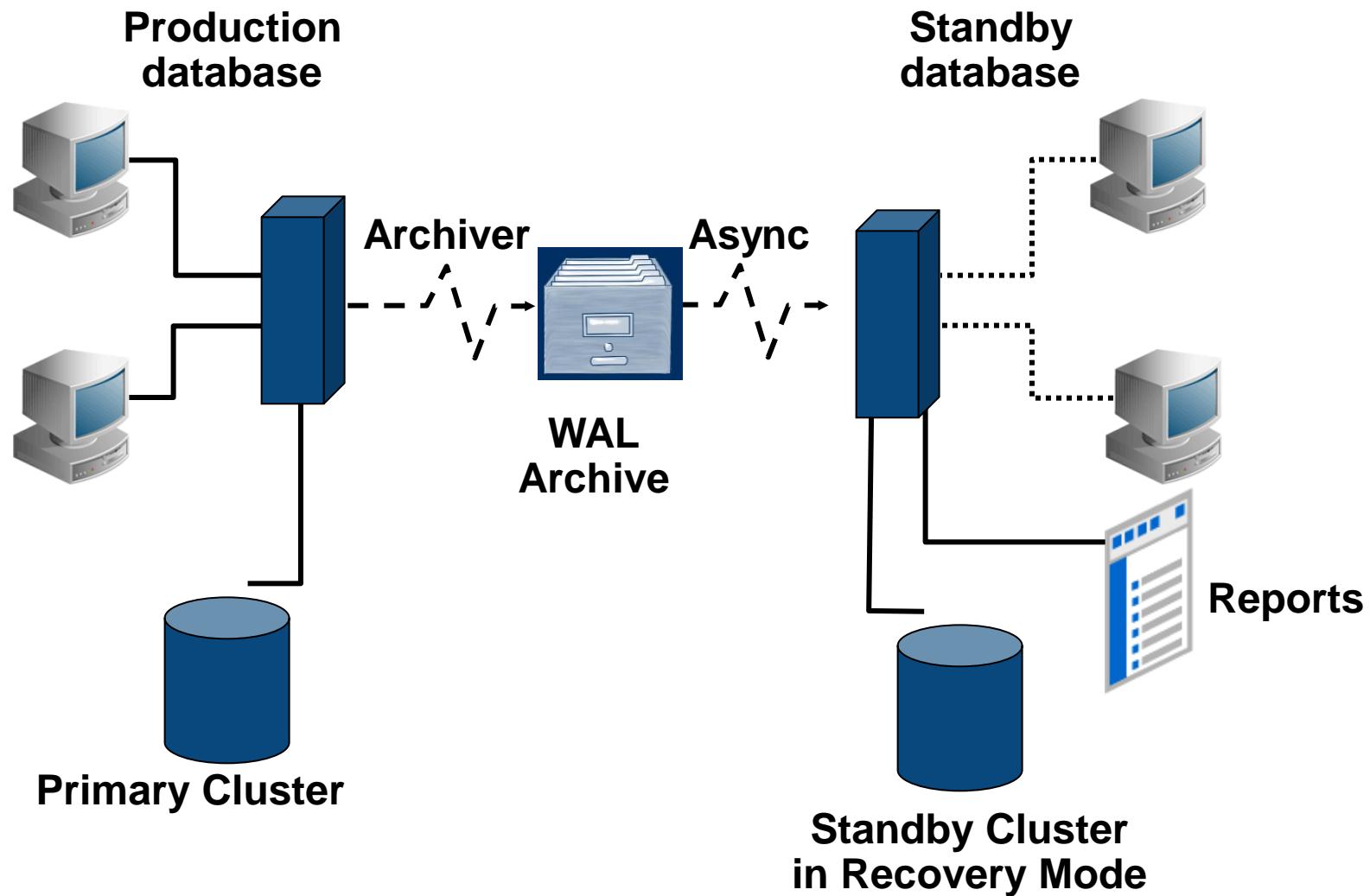
# Synchronous Replication

- Only streaming replication supports synchronous replication method and supports one or more synchronous standby servers
- Its 2-safe replication
- Two-phase commit actions
- Transaction is committed on primary and flushed to WAL segment of primary and standby server
- User gets a commit message after confirmation from both primary and standby
- This will introduce a delay in committing transactions

# Log-Shipping Standby Servers

- WAL archiving method for High Availability (HA) cluster
- Primary server is in WAL archive mode
- Standby server is in continuous recovery mode
- In continuous recovery mode standby server is reading and playing WAL logs from WAL archive area of primary server
- Low performance impact on primary
- Low administration overhead
- Supports warm and hot standby

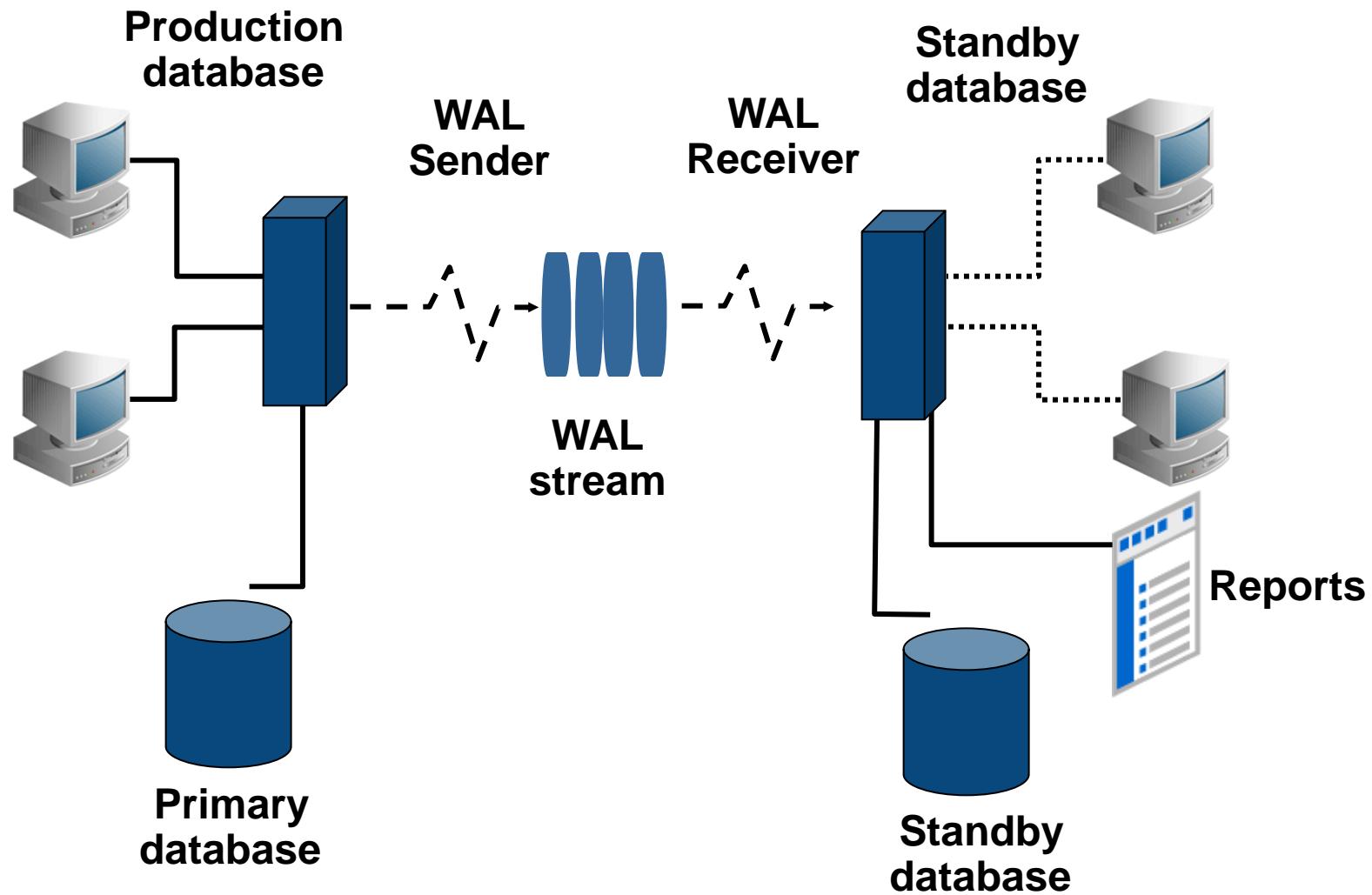
# Log-Shipping Architecture



# Streaming Replication

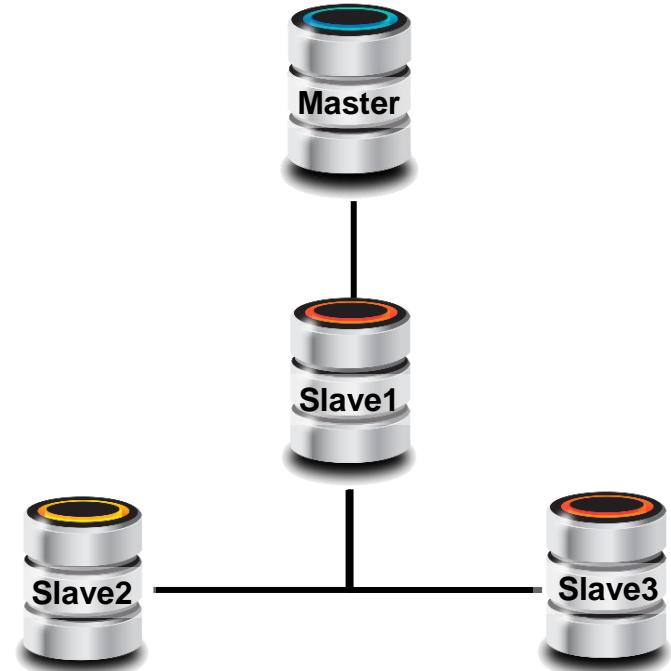
- Streaming Replication (Hot Standby) is a major feature of PostgreSQL
- Standby connects to the primary node using REPLICATION protocol
- WAL segments are streamed to standby server
- No log shipping delays. The transaction commits and immediately streams across to the standby
- Synchronous/Asynchronous option available
- Supports cascading replication, so the standby can also send replication changes sharing the overhead of a master

# Hot Streaming Architecture



# Cascading Replication

- Streaming replication supports single master node
- The ability of a standby server to send stream or archive changes to other standby servers is cascading replication
- Such standby servers are called cascading standby
- Cascading Replication helps minimize inter-site bandwidth overheads on master node
- Cascading Replication can only be Asynchronous



# Setting Up Replication Using Archive

1. WAL Archive must be configured on primary server
2. Archive location must be accessible to standby server
3. Take a full base backup of primary database cluster
4. Copy the backup to the standby server
5. If using database tablespaces, directory structure must be same on master and standby
6. Create a **recovery.conf** file inside data directory of standby database cluster
7. Turn on `standby_mode` and set `restore_command` to copy files from the WAL archive location

# Example - Replication Using Archive - Configure WAL Archiving

- Configure WAL archiving:

```
$ sudo mkdir /arch_dest
```

```
$ sudo chown postgres:postgres /arch_dest
```

```
$ vi /opt/PostgreSQL/9.6/data/postgresql.conf
```

```
wal_level = hot_standby
```

```
archive_mode = on
```

```
archive_command = 'cp %p /arch_dest/%f'
```

```
$ pg_ctl -D /opt/PostgreSQL/9.6/data -mf restart
```

# Example - Replication Using Archive - Take a Base Backup

- Take a base backup of primary and copy it to standby

```
$ psql -c "select pg_start_backup('standby1');"
$ cp -rp /opt/PostgreSQL/9.6/data /home/postgres/standby1
$ psql -c "select pg_stop_backup();"
$ rm -rf /home/postgres/standby1/postmaster.pid
```

- Note: **postmaster.pid** must be removed from standby and never from the master server data directory

# Example - Replication Using Archive – recovery.conf

- Create **recovery.conf**

```
$ vi /home/postgres/standby1/recovery.conf
standby_mode = on
restore_command='cp -rp /arch_dest/%f %p'
archive_cleanup_command = 'pg_archivecleanup /arch_dest %r'
trigger_file='/home/Postgres/create_me_for_failover_5432'
```

- Open **postgresql.conf** and turn on hot standby mode

```
$ vi /home/postgres/standby1/postgresql.conf
```

```
hot_standby = on
```

– If testing on single machine then comment archive parameters and change port:

```
port = 5433
```

- Save and start the standby

```
$ pg_ctl -D standby1/ -l standby1/startlog start
```

# Example - Replication Using Archive - Verify

- Connect to primary and add data
- Verify data is replicated on standby

```
postgres@pgcentos1:~$ psql -p 5432
Welcome to psql Terminal
postgres=# create table test_standby as select generate_series(1,10);
postgres=# table test_standby;
 generate_series

 1
 2
 3
 4
 5
 6
 7
 8
 9
10
(10 rows)

postgres=#
```

```
postgres=# \q
[postgres@pgcentos1 ~]$ psql -p 5433
Welcome to psql Terminal
postgres=# table test_standby;
 generate_series

 1
 2
 3
 4
 5
 6
 7
 8
 9
10
(10 rows)

postgres=#
```

# Setup Streaming Replication

# Prepare the Primary Server

- Change WAL Content parameter:

`wal_level = replica`

- Set only the minimum number of segments retained in **pg\_xlog**

`wal_keep_segments = 50`

- Two options to allow streaming connection:

`max_wal_senders`

- Set maximum number of concurrent connections from standby servers or stream clients. This enables the ability to stream WAL to the standby

`max_replication_slots`

- maximum number of replication slots that the server can support

- `wal_sender_timeout`

- Specify time in ms to terminate inactive replication connections. Default 60 sec

# Synchronous Streaming Replication Setup

- Default level of Streaming Replication is Asynchronous
- Synchronous level can also be configured to provide 2-safe replication
- Additional parameters need to be configured:

`synchronous_commit=on`

`synchronous_standby_names`

- Specifies a comma-separated list of standby names that can support synchronous replication

- Transactions can be configured to wait for replication by setting the `synchronous_commit` parameter to `remote_apply` or `remote_write`
- Transactions can be configured not to wait for replication by setting the `synchronous_commit` parameter to `local` or `off`
- During synchronous setup `pg_start_backup()` and `pg_stop_backup()` are run in a session with `synchronous_commit = off`

# Configure Authentication

- Authentication setting on the primary server must allow replication connections from the standby server(s)
- Provide a suitable entry or entries in **pg\_hba.conf** with the database field set to `replication`
- Open **pg\_hba.conf** of primary server:

```
host replication all 127.0.0.1/32 trust
```

Note - You will need to reload the primary server

# Take a Full Backup of the Primary Server

- Connect to the database as a superuser and issue the command:  
=# SELECT pg\_start\_backup('label');
- Perform the backup, using any convenient file-system-backup tool such as tar  
\$ cp -rp /opt/PostgreSQL/9.6/data /backup/data1
- Again connect to the database as a superuser, and issue the command:  
=# SELECT pg\_stop\_backup();
- Copy the backup directory on to the standby server
- You might change the port to 5445 for the standby server in case you configure standby on same machine

# Setting up Standby Server

- Remove **postmaster.pid** file
  - \$ rm -rf /backup/data1/postmaster.pid
- Standby server config parameters:
  - hot\_standby
  - max\_standby\_archive\_delay
  - max\_standby\_streaming\_delay
  - wal\_receiver\_status\_interval
  - hot\_standby\_feedback
  - wal\_receiver\_timeout

Note - `hot_standby` must be set to `on` for read-only transaction support on standby

# Setting up Standby Server – the recovery.conf

- Create **recovery.conf** file inside the data directory of the standby server
- Setup `recovery.conf` parameters
  - `standby_mode`
  - `primary_conninfo`
  - `primary_slot_name`
  - `recovery_min_apply_delay`
  - `trigger_file`
- Last step - start the standby

# Adding Cascading Replicated Standby Server

- Backup the Primary Server using `pg_basebackup`:  

```
$ pg_basebackup -h localhost -U replication_user -p 5432 -D /backup/data2
```
- A new cluster will be created with data directory “data2”
- Change the port if testing on same machine
- Create `recovery.conf` file in new standby
  - `standby_mode = on`
  - `primary_conninfo = 'host=localhost port=5433 user=replication_user password=secret'`
  - Note - `primary_conninfo` will connect to standby instead of master
- Change **`pg_hba.conf`** file of standby to allow connection from data2 cluster
- Start the cluster

# Monitoring Hot Standby

- pg\_stat\_replication
  - Show slaves connected to the master and other useful information
- Streaming Replication can also be monitored by:
  - Size of not yet applied WAL records
  - Lag time between WAL apply
- Recovery information functions:
  - pg\_is\_in\_recovery()
  - pg\_current\_xlog\_location
  - pg\_last\_xlog\_receive\_location
  - pg\_last\_xact\_replay\_timestamp()

# Recovery Control Functions

| Name                                    | Return Type       | Description                        |
|-----------------------------------------|-------------------|------------------------------------|
| <code>pg_is_xlog_replay_paused()</code> | <code>bool</code> | True if recovery is paused         |
| <code>pg_xlog_replay_pause()</code>     | <code>void</code> | Pauses recovery immediately        |
| <code>pg_xlog_replay_resume()</code>    | <code>void</code> | Restarts recovery if it was paused |

# Streaming Replication Example

- In this example we will setup async streaming replication on the local machine where default cluster is up and running
- Verify PostgreSQL Cluster is running on 5432 port and data directory is located at /opt/PostgreSQL/9.6/data before proceeding

# Streaming Replication Example – Steps on Primary

1. Login as postgres OS user `su - postgres`
2. Open **postgresql.conf** file using vi editor
3. Change following parameters:

```
wal_level = replica
max_wal_senders = 2
wal_keep_segments = 32
```

4. Save and close **postgresql.conf** file
5. Open **pg\_hba.conf** file using vi editor and add the following entry:

```
host replication all 127.0.0.1/32 trust
```

6. Restart the primary server

```
$ pg_ctl -D /opt/PostgreSQL/9.6/data -mf restart
```

# Streaming Replication Example

- Make a base backup by copying the primary server's data directory to the standby server

```
$ pg_basebackup -h localhost -U postgres -D
/home/postgres/data
```

- Next steps can be performed on a separate standby box if available
  - In that case don't forget to change the **pg\_hba.conf** entry added in the previous slides and copy the backup to standby server

# Streaming Replication Example – Steps on Hot Standby

1. Login as postgres OS user su – postgres
2. Open **postgresql.conf** file using vi editor
  - Port = 5433 --This is not required if standby has to run on different server
  - hot\_standby = on
  - Comment wal\_level, archive\_mode and archive\_command

3. Create recovery.conf file inside data directory of standby server

```
$ vi /home/postgres/data/recovery.conf
```

- standby\_mode = 'on'
- primary\_conninfo = 'host=localhost port=5432 user=postgres'
- trigger\_file = '/home/postgres/trigger\_hot\_standby\_failover'

4. Start the server using pg\_ctl command

```
$ pg_ctl -D /home/postgres/data -l /home/postgres/data logfile start
```

5. Verify the real time changes been transferred through streaming replication by running transaction on primary

# Example - Monitoring Streaming Replication

- Execute:

```
=# SELECT * FROM pg_stat_replication;
```

- Find lag (bytes):

```
=# Select pg_xlog_location_diff(sent_location,
replay_location) from pg_stat_replication;
```

- Find lag (seconds):

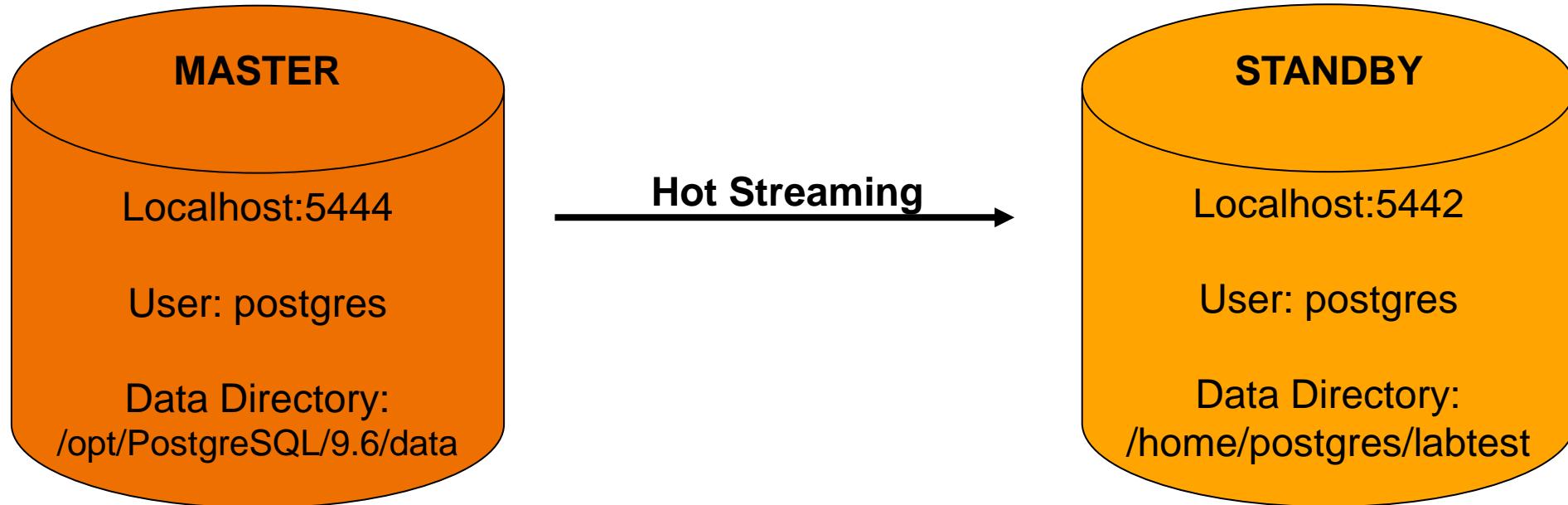
```
=# SELECT CASE WHEN pg_last_xlog_receive_location() =
pg_last_xlog_replay_location()
THEN 0 ELSE
EXTRACT (EPOCH FROM now() -pg_last_xact_replay_timestamp())
END AS stream_delay;
```

# Module Summary

- Data Replication
- Data Replication in PostgreSQL
- Sync or Async
- Log-Shipping Standby Servers
- Log-Shipping Architecture
- Streaming Replication
- Hot Streaming Architecture
- Cascading Replication
- Setup Replication Using Archive
- Setup Streaming Replication
  - Prepare the Primary Server
  - Synchronous Streaming Replication Setup
  - Configure Authentication
  - Take a Full Backup of Primary Server
  - Setting up the Standby Server
  - Adding Cascading Replicated Standby Server
  - Monitoring Hot Standby
  - Recovery Control Functions

# Lab Exercise – 1 Hot Standby

1. Implement Streaming Replication between two clusters running on your server as following:



# **Module 18**

# **Connection Pooling**

# Module Objectives

- Connection Pooling Overview
- pgpool-II - Features
  - Connection pooling
  - Replication
  - Load balance
  - Limiting exceeding connections
  - Automatic failover
- pgpool II - Installation and Configuration
- Configuring pcp.conf, pgpool.conf
- Setting up pool\_hba.conf for Client Authentication (HBA)
- Starting/Stopping pgpool-II

# Connection Pooling

- Each database client acquires a database connection from the database
- The Postmaster is responsible for listening and providing database connections
- Connection processes are destroyed once the clients log off
- Connection pooling:
  - Provides a mechanism to reuse the existing connection processes once the clients log off
  - Lowers connection overhead on the postmaster
  - Improves application database access performance
  - Spreads connection cost over repeated clients

# pgpool-II

- Open source
- Middleware between PostgreSQL servers and a PostgreSQL database client
- pgpool-II talks to PostgreSQL backend and frontend protocol, and relays a connection between them
- Features:
  - Connection Pooling
  - Replication
  - Load Balancing
  - Automatic Failover
  - Parallel Query

# Connection Pooling

- pgpool-II saves connections to the PostgreSQL servers
- It reuses them whenever a new connection with the same properties (i.e. username, database, protocol version) comes in
- It reduces connection overhead, and improves system's overall throughput
- Provides cluster-based connection pooling

# Replication

- pgpool-II can manage multiple PostgreSQL servers
- Using the replication function enables creating a real-time backup on 2 or more physical disks, so that the service can continue without stopping servers in case of a disk failure

# Load Balance

- If a database is replicated, executing a SELECT query on any server will return the same result
- pgpool-II automatically detects SELECT and can load balance reads among multiple servers, improving the system's overall throughput
- Load balance works best in a situation where there are a lot of users executing many queries at the same time

# Limiting Exceeding Connections

- There is a limit on the maximum number of concurrent connections with PostgreSQL, and connections are rejected after this limit has been reached
- Setting the maximum number of connections, however, increases resource consumption and affects system performance
- pgpool-II also has a limit on the maximum number of connections, but extra connections will be queued instead of returning an error immediately

# Automatic Failover

- pgpool-II automatically detects a failed master and can take action to promote a slave to be the new master
- After failover it automatically redirects all write load to the new master and forgets the failed master

# pgpool-II Installation

- pgpool-II works on Linux, Solaris, FreeBSD, and most of the UNIX-like architectures
- pgpool-II can be downloaded from  
<http://pgpool.net/mediawiki/index.php/Downloads>
- Installing pgpool-II requires gcc 2.9 or higher, GNU make and the libpq library
- After extracting the source tarball, execute the configure script
  - ./configure
  - make
  - make install

# Configuring pgpool-II

- Configuration files for pgpool-II are `$prefix/etc/pgpool.conf` and `$prefix/etc/pcp.conf` by default
- There are several operation modes in pgpool-II
- Each mode has associated functions which can be enabled or disabled, and specific configuration parameters to control their behaviors

# Configuring pcp.conf

- pgpool-II provides a control interface for its administration
- **pcp.conf** is the user/password file for authentication with this interface
- After installing pgpool-II, \$prefix/etc/pcp.conf.sample is created
  - \$ cp \$prefix/etc/pcp.conf.sample \$prefix/etc/pcp.conf
  - username: [password encrypted in md5]
- Password can be produced with the \$prefix/bin/pg\_md5 command:
  - \$ pg\_md5 -p password: <your password>

# Configuring pgpool.conf

- Each operation mode has specific configuration parameters in **pgpool.conf**
- After installing pgpool-II, \$prefix/etc/pgpool.conf.sample is created  
cp \$prefix/etc/pgpool.conf.sample \$prefix/etc/pgpool.conf
- There are additional sample pgpool.conf for each mode

pgpool.conf.sample-replication

pgpool.conf.sample-master-slave

pgpool.conf.sample-stream

# pgpool.conf parameters

## Connections

- listen\_addresses, port, pcp\_port

## Pools

- num\_init\_children, child\_life\_time, child\_max\_connections, client\_idle\_limit, enable\_pool\_hba, pool\_passwd

## Logs

- log\_destination, log\_connections, log\_statement

## File Location

- pid\_file\_name, logdir

## Failovers

- failover\_command, fallback\_command, follow\_master\_command

## Load Balancing

- replication\_mode, master\_slave\_sub, load\_balance\_mode, replicate\_select

## Backends

- backend\_hostname, backend\_port, backend\_weight, backend\_data\_directory, backend\_flag

# Setting up pool\_hba.conf for client authentication (HBA)

- Just like **pg\_hba.conf** with PostgreSQL, pgpool supports a similar client authentication
- This is done using `pool_hba.conf`
- By default, `pool_hba` authentication is disabled
- Change `enable_pool_hba` to `on` to enable it
- Copy `pool_hba.conf.sample` as `pool_hba.conf` and edit it if necessary
- Format and usage is similar to **pg\_hba.conf**
- Only "trust", "reject", "md5" and "pam" for METHOD field are supported
- To use md5 authentication, you must register username and password in `pool_passwd`
- Users can connect directly to the database and in such case **pg\_hba.conf** file settings are used for authentication

# Starting/Stopping pgpool-II

- All the backends and the System DB (if necessary) must be started before starting pgpool-II
- If pgpool is installed using rpm packages, use the init scripts from /etc/init.d/

- Syntax for pgpool command to start pgpool-II:

```
$ pgpool [-c] [-f config_file] [-a hba_file] [-F
pcp_config_file] [-n] [-D] [-d
```

- There are two ways to stop pgpool-II
- One is via a PCP command in interface and second is a pgpool command
- Syntax for pgpool command:

```
$ pgpool [-f config_file] [-F pcp_config_file] [-m
{s[mart] | f[ast] | i[mmediate]}] stop
```

# pgpool-II Example - Install

- In this example we will learn how to install pgpool-II
- The following steps can be performed using root user or sudo

## 1. Download pgpool-II

```
$ wget http://www.pgpool.net/download.php?f=pgpool-II-3.4.3.tar.gz
```

## 2. Unzip the tar file

```
$ tar -zxvf pgpool-II-3.4.3.tar.gz
$ cd pgpool-II-3.4.3/
```

## 3. You can view the parameters that can be configured before compiling pgpool-II sources

```
$./configure --help
```

## 4. Compile and install the sources

```
$./configure --with-pgsql=/home/postgres/pgsql/bin --with-pgsql-
 includedir=/home/postgres/pgsql/include --with-pgsql-
 libdir=/home/postgres/pgsql/lib --prefix=/home/postgres/pgpool/
$ mkdir /home/postgres/pgpool/
$ make
$ make install
```

# pgpool-II Example - Setup

## 5. After installation lets copy the sample config files

```
$ cd /home/postgres/pgpool
$ cp etc/pcp.conf.sample etc/pcp.conf
$ cp etc/pgpool.conf.sample etc/pgpool.conf
```

## 6. Generate password:

```
$ bin/pg_md5 -p edb
```

## 7. Add a user entry in pcp.conf file

```
$ vi etc/pcp.conf
postgres:e8a48653851e28c69d0506508fb27fc5
```

## 8. Save and close

# pgpool-II Example - pgpool.conf

9. Open pgpool.conf file and configure pgpool for providing connection pooling for our default PostgreSQL Cluster running on port 5432

```
$ vi etc/pgpool.conf
```

10. Edit following parameters

```
listen_addresses = '*'
port = 9999
backend_hostname0 = 'localhost'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/opt/PostgreSQL/9.6/data'
pid_file_name = '/home/postgres/pgpool/pgpool.pid'
logdir = '/home/postgres/pgpool'
```

11. Save and close

# pgpool-II Example – Final Steps

## 12. Start pgpool-II

```
$ bin/pgpool -f etc/pgpool.conf -F etc/pcp.conf -n
```

## 13. Connect to Postgres Cluster using pgpool

```
$ psql -p 9999 postgres postgres
```

## 14. Exit from the psql terminal and stop pgpool

```
$ bin/pgpool -mf stop
```

# Module Summary

- Connection Pooling Overview
- pgpool-II - Features
  - Connection pooling
  - Replication
  - Load balance
  - Limiting exceeding connections
  - Automatic failover
- pgpool II - Installation and Configuration
- Configuring pcp.conf, pgpool.conf
- Setting up pool\_hba.conf for Client Authentication (HBA)
- Starting/Stopping pgpool-II

# Lab Exercise - 1

1. PostgreSQL default cluster is running on port 5432. Install and configure pgpool-II and configure connection pooling for default cluster
2. Configure pgpool-II to pre-fork 64 server processes at startup
3. Disconnect a client if it has been idle for 5 mins
4. Configure 100 maximum number of cached connections in pgpool-II children processes

# **Module 19**

# **Table Partitioning**

# Module Objectives

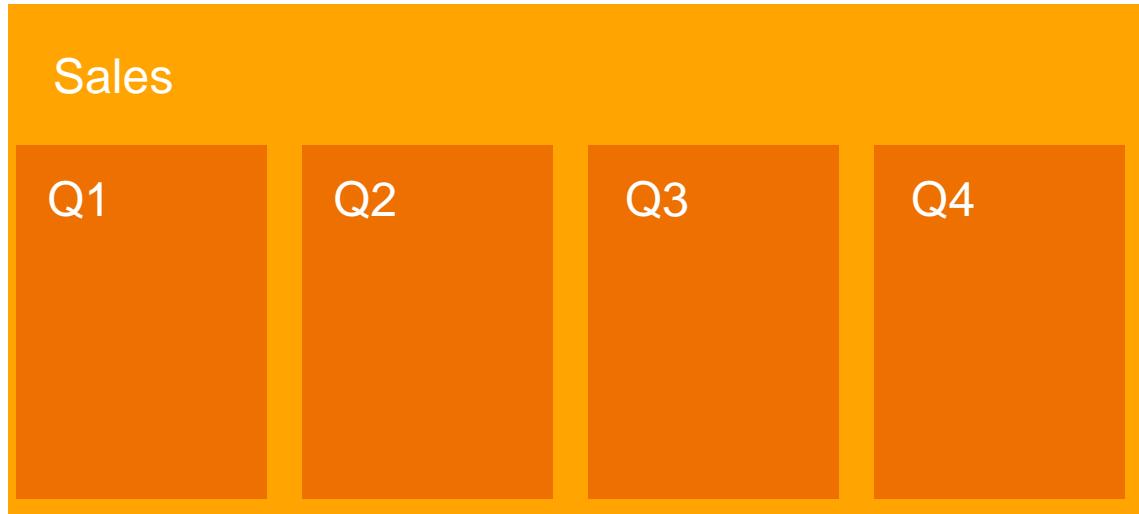
- Partitioning
- Partitioning Methods
- When to Partition
- Partitioning Setup
- Partitioning Example
- Partitioning and Constraint Exclusion
- Caveats

# Partitioning

- Partitioning refers to splitting what is logically one large table into smaller physical pieces
- Query performance can be improved dramatically for certain kinds of queries
- Improved UPDATE performance. When an index no longer fits easily in memory, both read and write operations on the index take progressively more disk accesses

# Partitioning in PostgreSQL

- PostgreSQL supports basic table partitioning
- Partitioning is entirely transparent to applications



# Partitioning Uses

- Bulk deletes may be accomplished by simply removing one of the partitions
- Seldom-used data can be migrated to cheaper and slower storage media
- PostgreSQL manages partitioning via table inheritance. Each partition must be created as a child table of a single parent table
- The parent table itself is normally empty; it exists just to represent the entire data set
- All child tables must have same logical attributes like columns and data types but can have different physical attributes like tablespace, fillfactor etc.

# Partitioning Methods

- Range Partitioning
  - Range partitions are defined via key column(s) with no overlap or gaps
- List Partitioning
  - Each key value is explicitly listed for the partitioning scheme

# When to Partition

- Here are some suggestions for when to partition:
  - When a table is very large and you have some reporting performance issues on the table
  - Tables greater than 10GB should always be considered for table partitioning
  - Tables containing historical data where newest data can be added to a new partition
    - A typical example is where 1 month of data is frequently updated and 11 months of data is read only
  - When the content of a table needs to be distributed on different storage devices depending on the usage of data

# Partitioning Setup

- Partitioning can be implemented without requiring any modifications to your applications
- Create the "base" or "Master" table
  - This table will contain no data
  - Only define constraints that will apply to ALL partitions
- Create the child or partition tables which each inherit from the master table
- Add table constraints to the partition tables to define the allowed key values in each partition
- Ensure that the constraints guarantee that there is no overlap between the key values permitted in different partitions
  - Note that there is no difference in syntax between range and list partitioning; those terms are descriptive only

# Partitioning Setup (Continued)

- Create Indexes for each partition
- Optionally, define a rule or trigger to redirect modifications of the master table to the appropriate partition
- Ensure that the **constraint\_exclusion** configuration parameter is enabled in **postgresql.conf**. Without this, queries will not be optimized as desired

# Partitioning Example

- In this example we will setup partitioning of sales table on the base of `sale_date` column
- Each partitioned (child) table will store sales information for a quarter
- We will use triggers



# Partitioning Example – Create Master Table

- Creating the Master Table

```
=> CREATE TABLE sales
 (sale_id NUMERIC NOT NULL, sale_date DATE, amount NUMERIC);
```

# Partitioning Example - Create Partition or Child Tables

- Creating Partition or Child Tables

```
=> CREATE TABLE sales_q1_2017 (
 CHECK (sale_date >= DATE '2017-01-01' AND
 sale_date < DATE '2017-04-01')
) INHERITS (sales);
```

```
=> CREATE TABLE sales_q2_2017 (
 CHECK (sale_date >= DATE '2017-04-01' AND
 sale_date < DATE '2017-07-01')
) INHERITS (sales);
```

```
=> CREATE TABLE sales_q3_2017 (
 CHECK (sale_date >= DATE '2017-07-01' AND
 sale_date < DATE '2017-10-01')
) INHERITS (sales);
```

```
=> CREATE TABLE sales_q4_2017 (
 CHECK (sale_date >= DATE '2017-10-01' AND
 sale_date < DATE '2018-01-01')
) INHERITS (sales);
```

# Partitioning Example – Create Indexes

- Create indexes on each partitioned table:

```
=> CREATE INDEX sales_10q1_sale_date ON sales_q1_2017 (sale_date);
=> CREATE INDEX sales_10q2_sale_date ON sales_q2_2017 (sale_date);
=> CREATE INDEX sales_10q3_sale_date ON sales_q3_2017 (sale_date);
=> CREATE INDEX sales_10q4_sale_date ON sales_q4_2017 (sale_date);
```

# Partitioning Example - Create Trigger Function

- Create a trigger function that will be called by trigger:

```
=> CREATE OR REPLACE FUNCTION sales_part_func () RETURNS trigger AS $$
BEGIN
IF NEW.sale_date >= DATE '2017-01-01' and NEW.sale_date < DATE '2017-04-
01' then
 INSERT INTO sales_q1_2017 VALUES (NEW.sale_id,NEW.sale_date,NEW.amount);
ELSEIF NEW.sale_date >= DATE '2017-04-01' and NEW.sale_date < DATE '2017-07-
01' then
 INSERT INTO sales_q2_2017 VALUES (NEW.sale_id,NEW.sale_date,NEW.amount);
ELSEIF NEW.sale_date >= DATE '2017-07-01' and NEW.sale_date < DATE '2017-10-
01' then
 INSERT INTO sales_q3_2017 VALUES (NEW.sale_id,NEW.sale_date,NEW.amount);
ELSEIF NEW.sale_date >= DATE '2017-10-01' and NEW.sale_date < DATE '2018-01-
01' then
 INSERT INTO sales_q4_2017 VALUES (NEW.sale_id,NEW.sale_date,NEW.amount);
END IF;
RETURN NULL;
END; $$ language PLPGSQL;
```

# Partitioning Example - Creating a Partitioning Trigger

- Creating a Partitioning Trigger on the `sales` table:

```
=> CREATE TRIGGER sales_part_trig
 BEFORE INSERT ON sales
 FOR EACH ROW
 EXECUTE PROCEDURE sales_part_func();
```

# Partitioning Example – Verify Partitions

- Insert some rows in the sales table and verify that the rows are inserted into partitioned tables using select statement with only keyword
- Remember applications will never know about this partition but can access the child tables directly for any bulk load or delete directly into child partition
- Always view the explain plan of a query that uses a partition table to verify only the required child tables are scanned

# Partitioning and Constraint Exclusion

- Constraint exclusion is a query optimization technique that improves performance for partitioned tables defined in the fashion described above
- As an example:

```
=> SET constraint_exclusion = on;
=> SELECT count(*) FROM sales WHERE sale_date >= DATE '2017-10-01';
```
- Without constraint exclusion, the above query would scan each of the partitions of the sales table

# Caveats

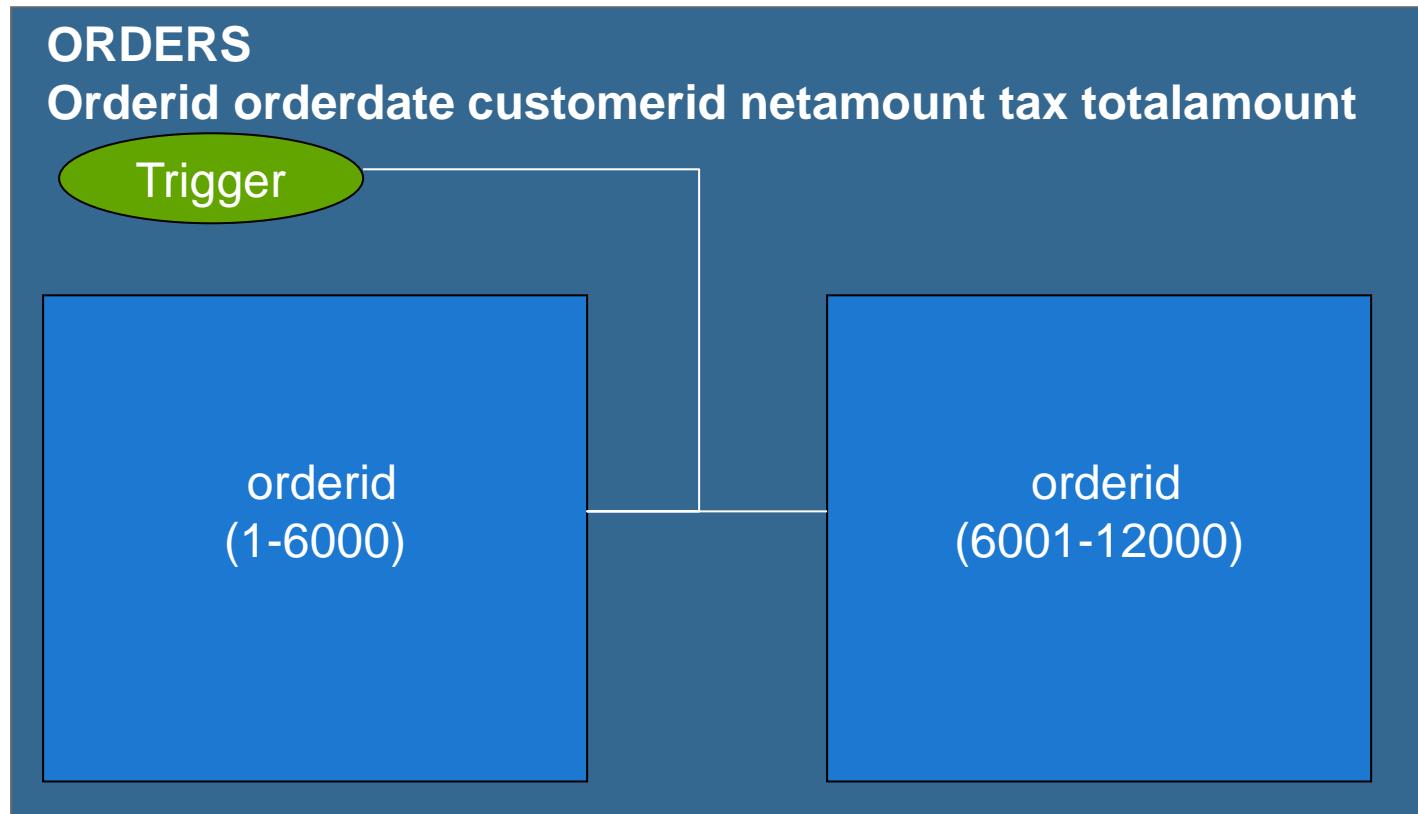
- There is no way to verify all of the CHECK constraints are mutually exclusive
- There is no way to specify that no rows should be inserted into the master table
- Each child table must be vacuumed separately if you are using manual vacuum
- In RULE based partitioning the COPY command cannot be used on a parent table
- A FOREIGN KEY cannot refer to any parent table column

# Module Summary

- Partitioning
- Partitioning Methods
- When to Partition
- Partitioning Setup
- Partitioning Example
- Partitioning and Constraint Exclusion
- Caveats

# Lab Exercise - 1

1. View the structure of the `orders` table in the `edbstore` database
2. Partition the existing `orders` table as shown in the following diagram:



# Lab Exercise - 2

1. Create a primary key on the `orderid` column of each partitioned table
2. View the execution plan of the following query:

```
SELECT * FROM orders WHERE orderid=3000;
```

3. Turn off the `constraint_exclusion` and view the execution plan of the following query:

```
SELECT * FROM orders WHERE orderid=3000;
```

# **Module 20**

# **Extensions**

# Module Objectives

- What are Extension modules?
- Installing Extension Modules
- Module Index

# What are Extension Modules?

- Modules in the “Extension” directory are additional features to PostgreSQL
- Generally, Extension modules aren’t included in the core database
- They may be still under development
- The modules in the Extension directory are tied to that particular version of Postgres, and the modules that are available change over time
- Once loaded in a database they can function just like features that are built in
- Also know as Additional Supplied Modules

# Installing Extension Modules

- In order to use an Extension module in a database you need to run the `CREATE EXTENSION` command to install the module's features into that database
- By default, Extension files are installed at `PREFIX/share/extension`

|                       |                           |                            |                         |
|-----------------------|---------------------------|----------------------------|-------------------------|
| adminpack.control     | hstore_plperl.control     | pageinspect.control        | pltcl.control           |
| autoinc.control       | hstore_plperlu.control    | pg_buffercache.control     | pltclu.control          |
| bloom.control         | hstore_plpython2u.control | pgcrypto.control           | postgres_fdw.control    |
| btree_gin.control     | hstore_plpython3u.control | pg_freespacemap.control    | refint.control          |
| btree_gist.control    | hstore_plpythonu.control  | pg_prewarm.control         | seg.control             |
| chkpass.control       | insert_username.control   | pgrowlocks.control         | sslinfo.control         |
| citext.control        | intagg.control            | pg_stat_statements.control | tablefunc.control       |
| cube.control          | intarray.control          | pgstattuple.control        | tcn.control             |
| dblink.control        | isn.control               | pg_trgm.control            | timetravel.control      |
| dict_int.control      | lo.control                | pg_visibility.control      | tsearch2.control        |
| dict_xsyn.control     | ltree.control             | pldbgapi.control           | tsm_system_rows.control |
| earthdistance.control | ltree_plpython2u.control  | plperl.control             | tsm_system_time.control |
| file_fdw.control      | ltree_plpython3u.control  | plperlu.control            | unaccent.control        |
| fuzzystrmatch.control | ltree_plpythonu.control   | plpgsql.control            | uuid-ossp.control       |
| hstore.control        | moddatetime.control       | plpython3u.control         | xml2.control            |

# Installing Extension Modules (continued)

- You need to register the new objects in the database system by running the `CREATE EXTENSION` command
- Alternatively, run it in database template1 so that the module will be copied into subsequently-created databases by default
- Syntax:  
`=# CREATE EXTENSION module_name;`
  - This command must be run by a database superuser

# Module Index

- **adminpack** - File and log manipulation routines, used by pgAdmin
- **btree\_gist** - Support for emulating B-Tree indexing in GiST
- **chkpass** - An auto-encrypted password datatype
- **cube** - Multidimensional-cube datatype (GiST indexing example)
- **dblink** - Allows remote query execution
- **earthdistance** - Operator for computing Earth distance for two points
- **fuzzystrmatch** - Levenshtein, metaphone, and soundex fuzzy string matching
- **hstore** - Module for storing (key,value) pairs
- **pg\_visibility** – To examine the visibility map and page-level visibility info
- **bloom** – Provides an index access method based on bloom filters

## Module Index (Continued)

- **intagg** - Integer aggregator
- **intarray** - Index support for arrays of int4, using GiST
- **isbn** - Postgres type extensions for ISBN, ISSN, ISMN, EAN13 product numbers
- **lo** - Large Object maintenance
- **ltree** – Adds ltree data type for storing hierarchical tree-like structure
- **pg\_buffercache** - Real time queries on the shared buffer cache
- **pg\_freespacemap** - Displays the contents of the free space map (FSM)

# Module Index (Continued)

- **pg\_trgm** - Functions for determining the similarity of text based on trigram matching
- **pgcrypto** - Cryptographic functions
- **pgrowlocks** - A function to return row locking information
- **pgstattuple** - A function to return statistics about "dead" tuples and free space within a table
- **seg** - Confidence-interval datatype (GiST indexing example)
- **spi** - Various trigger functions, examples for using SPI
- **sslinfo** - Functions to get information about SSL certificate
- **tsearch2** – Provide backward compatible text search
- **uuid-ossp** – Provide functions to generate UUIDs

# Example

```
postgres=# CREATE EXTENSION lo;
CREATE EXTENSION
postgres=# CREATE TABLE images (img_id NUMERIC, img LO);
CREATE TABLE
postgres=# INSERT INTO images VALUES(1,LO_IMPORT('/home/postgres/logo.png'));
INSERT 0 1
postgres=# SELECT * FROM images;
 img_id | img
-----+-----
 1 | 16415
(1 row)

postgres=# SELECT LO_EXPORT(img,'/home/postgres/logo1.png') FROM images WHERE img_id=1;
 lo_export

 1
(1 row)

postgres=#
```

# Module Summary

- What are Extension modules?
- Installing Extension Modules
- Module Index

# **Module 21**

# **Monitoring**

# Module Objectives

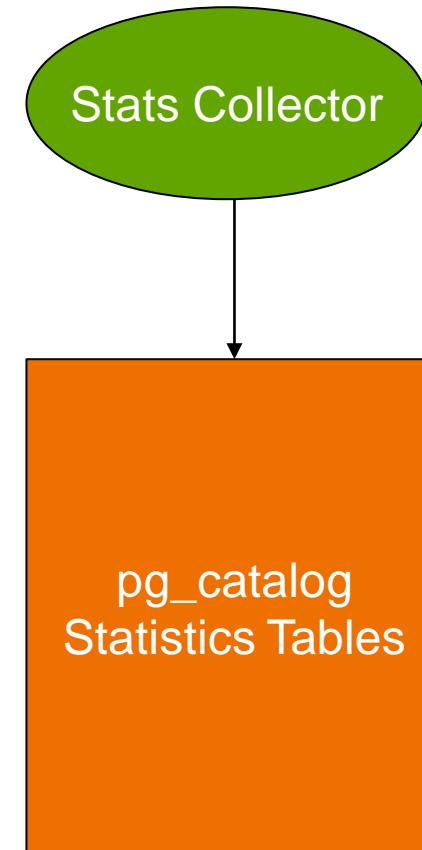
- Database Monitoring
- Database Statistics
- The Statistics Collector
- Database Statistic Tables
- Operating System Process Monitoring (Unix)
- Server Processes – Linux
- Current Sessions and Locks
- Logging Slow Running Queries
- Disk Usage
- Postgres Enterprise Manager (PEM)
  - PEM - Features
  - PEM - Architecture

# Database Monitoring

- Database monitoring consists of capturing and recording the database events
- This information helps DBAs to detect, identify and fix potential database performance issues
- Database monitoring statistics make it easy for DBAs to monitor the health and performance of PostgreSQL databases
- Several tools are available for monitoring database activity and analyzing performance

# Database Statistics

- Database statistics catalog tables store database activity information:
  - Current running sessions
  - Running SQL
  - Locks
  - DML counts
  - Row counts
  - Index usage



# The Statistics Collector

- Stats Collector is a process that collects and reports information about database activities
- Stats Collector adds some overhead to query execution
- Stats Parameters:
  - track\_counts - controls table and index statistics
  - track\_activities - enables monitoring of the current command
- The stats collector uses temp files stored in subdirectory pg\_stat\_tmp
- Permanent statistics are stored in the pg\_catalog schema in the global subdirectory
- **Performance Tip** - parameter stats\_temp\_directory can be pointed at a RAM-based file system

# Database Statistic Tables

- `pg_catalog` schema contains a set of tables, views and functions which store and report database statistics
- `pg_class` and `pg_stats` catalog tables store all statistics
- `pg_stat_database` view can be used to view stats information about a database and temporary file counts and sizes
- `pg_stat_bgwriter` shows background writer stats and checkpoint timing information
- `pg_stat_user_tables` shows information about activities on a table like inserts, updates, deletes, vacuum, autovacuum etc.
- `pg_stat_user_indexes` shows information about index usage for all user tables
- `pg_stat_progress_vacuum` shows one row for each backend including autovacuum worker processes that is currently vacuuming

# Operating System Process Monitoring (Unix)

- Unix
  - ps – Information about current processes
    - \$ ps waux | grep post (for Linux and OS/X)
    - \$ ps -ef | grep post (other Unix)
  - netstat – Information about current network connections
    - \$ netstat -an | grep LISTEN

# Server Processes - Linux

```
[postgres@localhost ~]$ ps -ef | grep postgres
root 5525 5486 0 15:50 pts/0 00:00:00 su - postgres
postgres 5532 5525 0 15:50 pts/0 00:00:00 -bash
postgres 5581 5532 0 15:51 pts/0 00:00:00 ps -ef
postgres 5582 5532 0 15:51 pts/0 00:00:00 grep --color=auto postgres
postgres 23310 1 0 00:46 ? 00:00:01 /opt/PostgreSQL/9.6/bin/postgres -D /opt/
PostgreSQL/9.6/data
postgres 23311 23310 0 00:46 ? 00:00:00 postgres: logger process
postgres 23313 23310 0 00:46 ? 00:00:00 postgres: checkpointer process
postgres 23314 23310 0 00:46 ? 00:00:00 postgres: writer process
postgres 23315 23310 0 00:46 ? 00:00:02 postgres: wal writer process
postgres 23316 23310 0 00:46 ? 00:00:01 postgres: autovacuum launcher process
postgres 23317 23310 0 00:46 ? 00:00:00 postgres: archiver process
postgres 23318 23310 0 00:46 ? 00:00:03 postgres: stats collector process
```

# Current Sessions and Locks

- pg\_locks - system table that stores information about outstanding locks in lock manager
- pg\_stat\_activity - shows the process id, database, user, query string and start time of currently executing query
  - Run the following SQL command:

```
SELECT * FROM pg_stat_activity;
```
- Terminate a session gracefully using the pg\_terminate\_backend function
- Rollback a transaction gracefully using function pg\_cancel\_backend

# Logging Slow Running Queries

- `log_min_duration_statement`
  - Sets a minimum statement execution time (in milliseconds) that causes a statement to be logged. All SQL statements that run for the time specified or longer will be logged with their duration.
  - Setting this to 0 will print all queries and their durations.
  - A setting of -1 (the default) disables the feature.
  - Enabling this option can be useful in tracking down non-optimized queries in your applications.

# Disk Usage

- The following stat functions can be used to view database size:
  - `pg_database_size(name)`
  - `pg_tablespace_size(name)`
- The disk usage command ‘du’ is also helpful in determining the growth of disk from time to time

# Tracking Execution Statistics

- Track query execution statistics using `pg_stat_statements` extension
- `pg_stat_statements` tracks statistics across all databases of a cluster
- `pg_stat_statements` extension add view, functions and configuration parameters
- View - `pg_stat_statements` to access the query statistics
- Functions - `pg_stat_statements_reset` to reset the statistics

# pg\_stat\_statements Setup

- Add `pg_stat_statements` to `shared_preload_libraries` parameter in `postgresql.conf`
- Configure available parameters:
  - `pg_stat_statements.max` - Maximum number of tracked statements, **default 5000**
  - `pg_stat_statements.track` – Which statements are counted, `top`, `all` or `none`. **Default is top**
  - `pg_stat_statements.track_utility` - Track commands other than `SELECT`, `INSERT`, `UPDATE` and `DELETE`. **Default is ON**
  - `pg_stat_statements.save` - Whether save statement statistics across server shutdowns. **Default is ON**
- Restart the Database Cluster
- Connect with a database and enable `pg_stat_statements` extension

# Postgres Enterprise Manager (PEM)

- Postgres Enterprise Manager (PEM) is an enterprise class software tool
- Assists DBAs in administering, monitoring, and tuning PostgreSQL and EDB Postgres Advanced database servers
- PEM is architected to manage and monitor multiple servers from a single console

# PEM - Features

- Global dashboards keep you up to date on the up/down/performance status of all your servers
- You can manage database servers regardless of the operating systems
- Database administration can be carried out via a graphical user interface
- Provides a full SQL integrated development environment (IDE)
- Lightweight and efficient agents monitor all aspects of each database server's operations

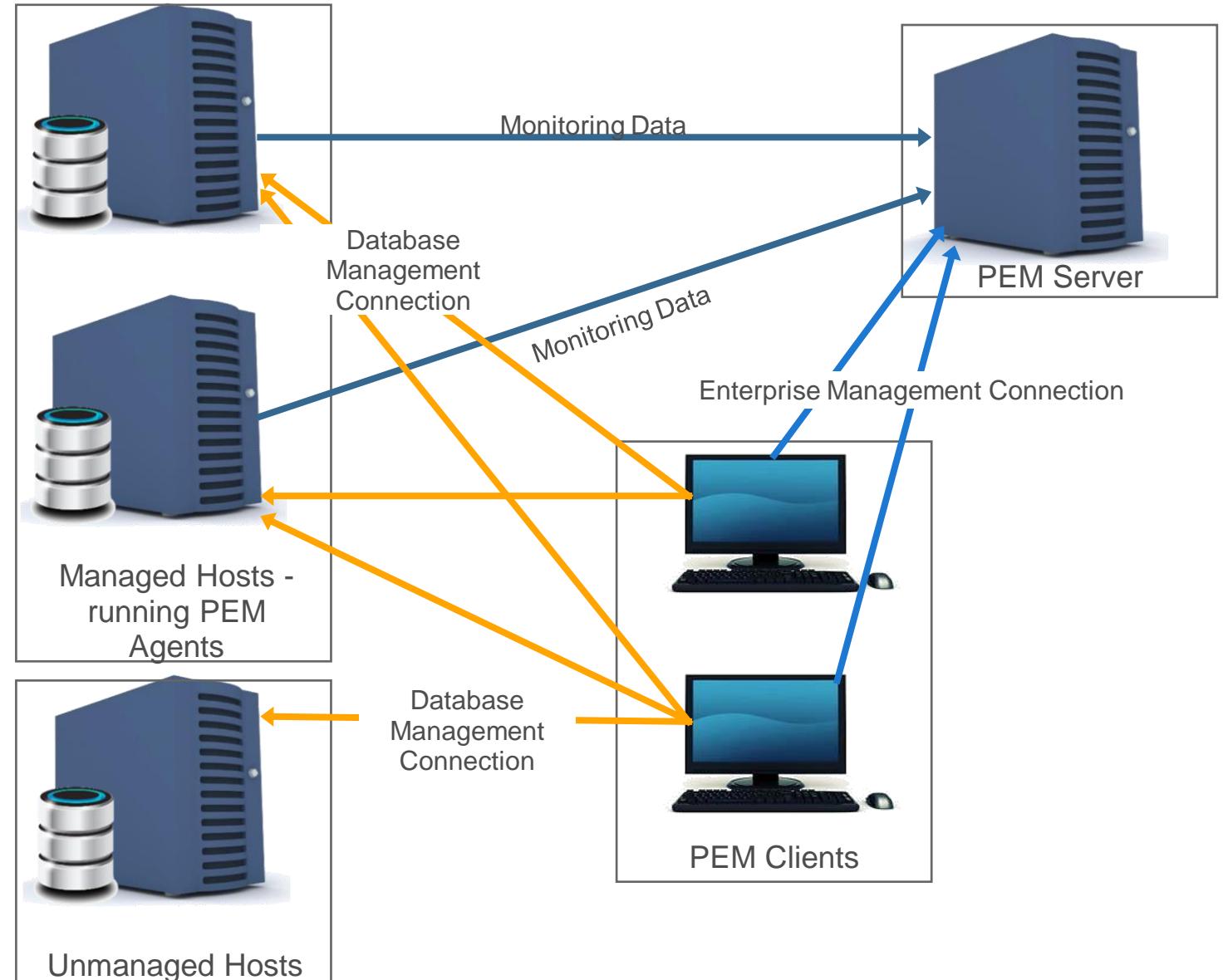
# PEM - Features

- Ability to create performance thresholds for each key metric e.g. memory, storage, etc.
- Any threshold violation results in an alert being sent to a centralized dashboard
  - Alerts can be sent via e-mail or SNMP trap
- All key performance-related statistics are automatically collected and retained
- Forecasts resource usage in the future
- Helps to identify and tune poorly running SQL statements
- Wide platform support

# PEM - Architecture

- PEM is composed of three primary components
  - The PEM Server
  - The PEM Agent
  - The PEM Client

Note - PEM Web Client



# Module Summary

- Database Monitoring
- Database Statistics
- The Statistics Collector
- Database Statistic Tables
- Operating System Process Monitoring (Unix)
- Server Processes – Linux
- Current Sessions and Locks
- Logging Slow Running Queries
- Disk Usage
- Postgres Enterprise Manager (PEM)
  - PEM - Features
  - PEM - Architecture

# Lab Exercise - 1

In this lab we will learn how to use PEM

1. Download and install Postgres Enterprise Manager Server
2. Download and install Postgres Enterprise Manager Client
3. Open PEM Client
4. Bind default cluster running on port 5432 with PEM Agent for collecting monitoring data
5. View Global dashboard for the PEM server
6. View current database activity dashboard for your default cluster
7. View Memory dashboard for your default cluster
8. View OS information for the server where default cluster is running

## Lab Exercise - 2

1. Write a query to view current running session on your default cluster
2. Write a query to view how many users are idle in session
3. Write a query to view how many users are connected with `edbstore` database
4. Open another terminal and connect with `edbstore` database using `psql`
  - Find the process id for the newly connected user using previous `psql` terminal
  - Write a command to rollback current running query
  - Write a command to terminate the newly created `psql` session

# Lab Exercise - 3

1. Open **postgresql.conf** file for your default cluster
2. Which parameter can be changed to log slow running queries?
3. Change this parameter so that all the queries running more than 5 secs are logged
4. Write a query to view current locks held in the lock manager
5. Write a query to find size of `edbstore` database

# **Module 22**

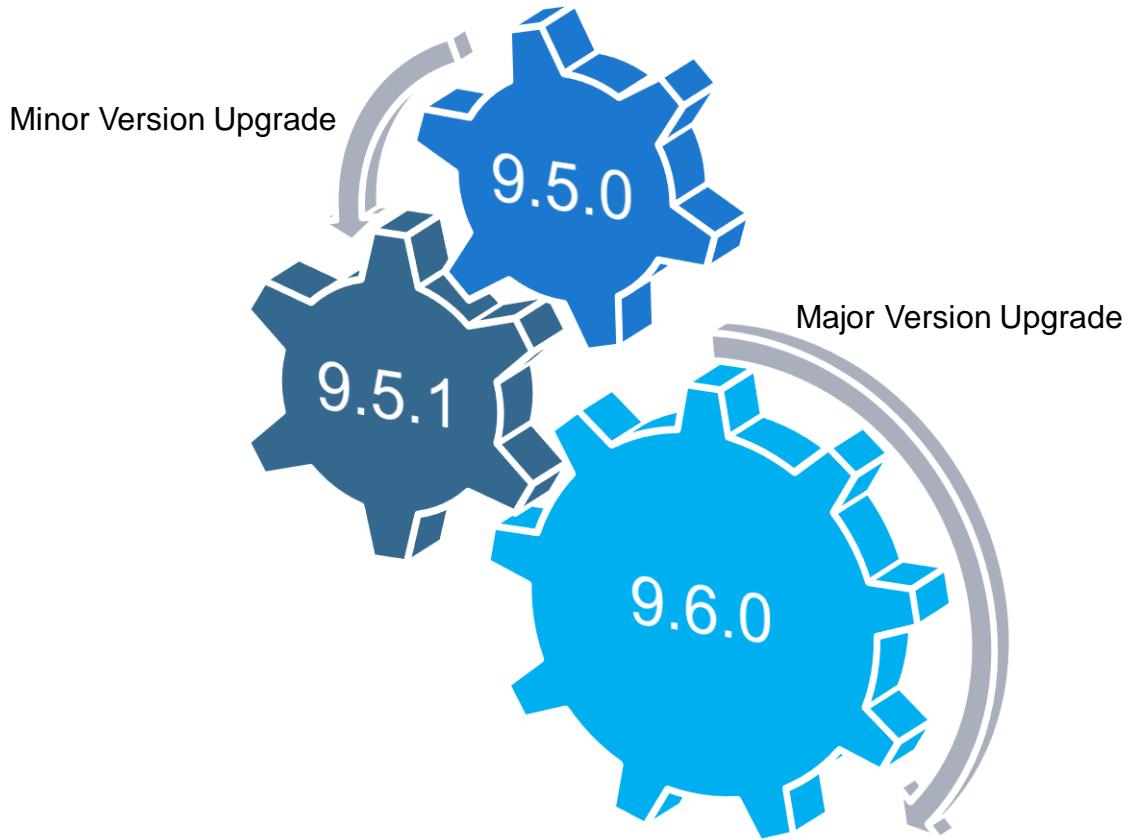
# **Upgrading Best Practices**

# Module Objectives

- Version Change and Upgrade
- Need to Upgrade
- Upgrade Plan
- Upgrade Using pg\_upgrade
- Upgrading Best Practices

# Version Change and Upgrade

- Minor Version Upgrade
  - Storage format not changed
  - Simply replace executable
  - Start server with upgraded executable and old data directory
- Major Version Upgrade
  - Storage format may change
  - Upgrade is complicated and can be performed using:
    - pg\_dumpall
    - pg\_upgrade
    - Replication

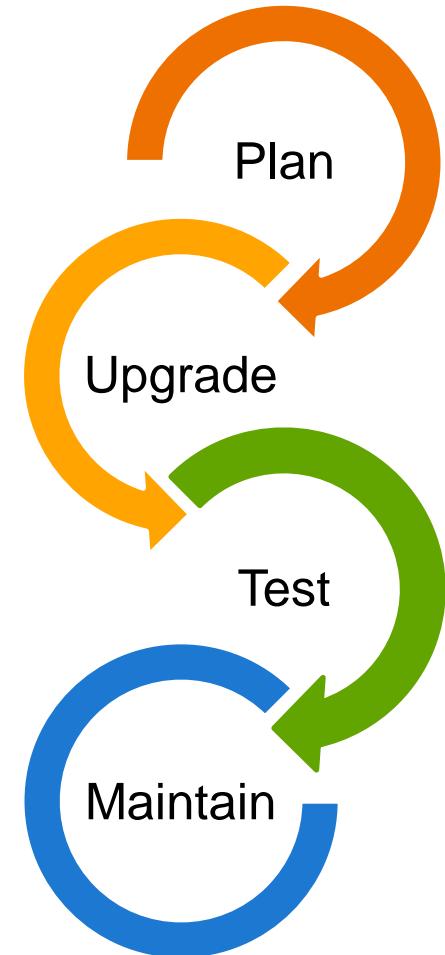


# Need to Upgrade

- PostgreSQL is an actively developed open source database
- Major releases include new features, capabilities and performance enhancements
- Minor releases include security and data corruption bug fixes
- PostgreSQL community recommends all users to run the latest available minor release
- Not upgrading is riskier than upgrading
- PostgreSQL community aims to fully support a major release for five years

# Upgrade Plan

- Read the Manual -  
<http://www.postgresql.org/docs/9.6/static/release-9-6.html>
- Check for user-visible incompatibilities
- Application must be tested on new version before complete switch
- It is recommended to setup concurrent installations of old and new Postgres versions
- Plan for the worst case – Going back to old version
- Decide the cutoff time
- Communicate the goal and timing to the technical support team



# Upgrade Using pg\_upgrade

## PostgreSQL

- Version 8.4 or later
- Data directory
- Old binaries

## pg\_upgrade

- Runs on new server
- Old and new cluster info
- Can be run in link mode
- Support for Parallel load

## PostgreSQL

- Latest Version
- New Features
- Better Performance

# Example - pg\_upgrade

- Scenario - Upgrade PostgreSQL 9.1 data cluster to PostgreSQL 9.6
- Solution:
  - Step 1 - Verify the source version and data directory

```
[postgres@localhost ~]$ psql -p 5430 postgres postgres
Password for user postgres:
psql.bin (9.1.24)
Type "help" for help.

postgres=# select version();
 version

PostgreSQL 9.1.24 on x86_64-unknown-linux-gnu, compiled by gcc
c (GCC) 4.1.2 20080704 (Red Hat 4.1.2-52), 64-bit
(1 row)

postgres=# show data_directory;
 data_directory

/opt/PostgreSQL/9.1/data
(1 row)
```

```
edbstore=> \dt
 List of relations
 Schema | Name | Type | Owner
-----+-----+-----+-----+
edbuser | categories | table | edbuser
edbuser | cust_hist | table | edbuser
edbuser | customers | table | edbuser
edbuser | dept | table | edbuser
edbuser | emp | table | edbuser
edbuser | inventory | table | edbuser
edbuser | job_grd | table | edbuser
edbuser | jobhist | table | edbuser
edbuser | locations | table | edbuser
edbuser | orderlines | table | edbuser
edbuser | orders | table | edbuser
edbuser | products | table | edbuser
edbuser | reorder | table | edbuser
(13 rows)
```

# Example - pg\_upgrade

- Scenario - Upgrade PostgreSQL 9.1 data cluster to PostgreSQL 9.6
- Solution:
  - Step 1 - Verify the source version and data directory

```
[postgres@localhost ~]$ psql -p 5430 postgres postgres
Password for user postgres:
psql.bin (9.1.24)
Type "help" for help.

postgres=# select version();
 version

PostgreSQL 9.1.24 on x86_64-unknown-linux-gnu, compiled by gcc
c (GCC) 4.1.2 20080704 (Red Hat 4.1.2-52), 64-bit
(1 row)

postgres=# show data_directory;
 data_directory

/opt/PostgreSQL/9.1/data
(1 row)
```

```
edbstore=> \dt
 List of relations
 Schema | Name | Type | Owner
-----+-----+-----+-----+
edbuser | categories | table | edbuser
edbuser | cust_hist | table | edbuser
edbuser | customers | table | edbuser
edbuser | dept | table | edbuser
edbuser | emp | table | edbuser
edbuser | inventory | table | edbuser
edbuser | job_grd | table | edbuser
edbuser | jobhist | table | edbuser
edbuser | locations | table | edbuser
edbuser | orderlines | table | edbuser
edbuser | orders | table | edbuser
edbuser | products | table | edbuser
edbuser | reorder | table | edbuser
(13 rows)
```

# Example - pg\_upgrade (Continued)

- Step 2 - Install PostgreSQL 9.6
- Step 3 - Initialize a new 9.6 data cluster

```
[postgres@localhost ~]$ initdb -V
initdb (PostgreSQL) 9.6.1
[postgres@localhost ~]$ mkdir pg96
[postgres@localhost ~]$ initdb -D pg96/data
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
```

The database cluster will be initialized with locale "en\_US.UTF-8".  
The default database encoding has accordingly been set to "UTF8".  
The default text search configuration will be set to "english".

Data page checksums are disabled.

```
creating directory pg96/data ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
```

# Example - pg\_upgrade (Continued)

- Step 4 - Stop source 9.1 data cluster

```
[postgres@localhost ~]$ pg_ctl -D /opt/PostgreSQL/9.1/data stop
waiting for server to shut down.... done
server stopped
[postgres@localhost ~]$
```

- Step 5 - Run pg\_upgrade

```
[postgres@localhost ~]$ pg_upgrade -V
pg_upgrade (PostgreSQL) 9.6.1
[postgres@localhost ~]$ pg_upgrade -d /opt/PostgreSQL/9.1/data -D /home/postgres/pg96/d
ata -b /opt/PostgreSQL/9.1/bin -B /opt/PostgreSQL/9.6/bin -v
Upgrade Complete

Optimizer statistics are not transferred by pg_upgrade so,
once you start the new server, consider running:
./analyze_new_cluster.sh

Running this script will delete the old cluster's data files:
./delete_old_cluster.sh
```

# Example - pg\_upgrade (Continued)

- Step 6 - Start newly upgraded 9.6 data cluster

```
[postgres@localhost ~]$ pg_ctl -D /home/postgres/pg96/data -l /home/postgres/pg96/startlog
start
server starting
[postgres@localhost ~]$ cat pg96/startlog
LOG: database system was shut down at 2016-12-14 16:14:23 IST
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
LOG: autovacuum launcher started
```

- Step 7 - Connect and verify upgrade

```
[postgres@localhost ~]$ psql --version
psql (PostgreSQL) 9.6.1
[postgres@localhost ~]$ psql -p 5432
psql.bin (9.6.1)
Type "help" for help.

postgres=# show data_directory;
 data_directory

/home/postgres/pg96/data
(1 row)
```

```
postgres=# \c edbstore edbuser
You are now connected to database "edbstore" as user "edbuser".
edbstore=> \dt
 List of relations
 Schema | Name | Type | Owner
-----+-----+-----+-----+
 edbuser | categories | table | edbuser
 edbuser | cust_hist | table | edbuser
 edbuser | customers | table | edbuser
 edbuser | dept | table | edbuser
 edbuser | emp | table | edbuser
 edbuser | inventory | table | edbuser
 edbuser | job_grd | table | edbuser
 edbuser | jobhist | table | edbuser
 edbuser | locations | table | edbuser
 edbuser | orderlines | table | edbuser
 edbuser | orders | table | edbuser
 edbuser | products | table | edbuser
edbuser | reorder | table | edbuser
(13 rows)
```

# Upgrading Best Practices

- PostgreSQL upgrade is not automatic
- Run `pg_upgrade` with `--check` option for any upgrade issues
- Take a full backup before and after backup
- If `--link` option is used with `pg_upgrade`, it's unsafe to start old cluster
- Check for failed rebuild, and reindex cases reported by `pg_upgrade`
- Watch for post-upgrade warnings generated by `pg_upgrade` and run the generated script on new database as Superuser
- Do not stop or abort the upgrade process once started
- In Streaming Replication environment standby server also needs upgrade
- Once satisfied, remove old cluster footprints

# Module Summary

- Version Change and Upgrade
- Need to Upgrade
- Upgrade Plan
- Upgrade Using pg\_upgrade
- Upgrading Best Practices

# Course Summary

- Introduction
- System Architecture
- Installation and Database Clusters
- Configuration
- Creating and Managing Databases
- User Tools - CUI and GUI
- Security
- SQL Primer
- Backup, Recovery and PITR
- Routine Maintenance Tasks
- Data Dictionary
- Loading and Moving Data
- SQL Tuning
- Performance Tuning
- Streaming Replication
- Connection Pooling using pgpool-II
- Table Partitioning
- Extensions
- Monitoring
- Upgrading Best Practices