POSTGRESQL DATABASE ADMINISTRATION

By : Siddheshwar Soni, DataXorg data Services.

PostgreSQL is an advanced, enterprise class open source **Object-Relational Database Management System (ORDBMS)**. It is a relational database that supports both SQL (relational) and JSON (non-relational) querying.

PostgreSQL is having robust feature-sets including
Multi-Version Concurrency Control (MVCC),
Point in time recovery,
Tablespaces,
Asynchronous replication,
Nested transactions,
Online/hot backups,
Refined query planner/optimizer, and
Write ahead logging. It supports international character sets, multi-byte character encodings, Unicode, and it is locale-aware for sorting, case-sensitivity, and formatting. PostgreSQL is highly scalable both in the quantity of data it can manage and in the number of concurrent users it can accommodate.

Topics covered are:

- PostgreSQL Process and Memory Architecture
- Installation of PostgreSQL v12 - Windows and Linux
- Setup PostgreSQL v12 Environment Variables - Windows and Linux
- PostgreSQL Page Layout
- Database Directory Layout
- PostgreSQL Configuration Files
- Cluster in PostgreSQL (Initdb,Start/Stop,Reload/Restart Cluster)
- Createdb/Dropdb, Createuser/Dropuser Utility
- Create Schema and Schema Search Path
- Psql Interface Commands
- Pg System Catalogs
- Working with PostgreSQL Objects

- CRUD Operations
- Table Inheritance
- Table Partitioning
- Tablespace in PostgreSQL
- Backup, Continuous Archiving and PITR (Point-in-Time-Recovery)
- Explain plan and Query Execution Cost
- Maintenance (Updating Planner Statistics, Vacuum, Routine Reindex & Cluster)
- Data Fragmentation
- Transaction ID wraparound, Vacuum freeze, Auto_Vacuum Demon
- Pgadmin 4
- Uninstall PostgreSQL v12 on Windows and Linux
- Bonus Module
- Journey into Postgresql v13
- Postgresql v13 - New Features

What you'll learn

- Understand PostgreSQL Process and Memory Architecture.
- Install PostgreSQL v12 on Windows and Linux.
- How to Setup Environment Variable on Windows and Linux.
- PostgreSQL Directory Structure.
- PostgreSQL Configuration Files.
- How to use PSQL command line interface to connect to PostgreSQL.
- How to Create Database/User/Schema from utility and Psql commands.
- Learn how to use pg system catalogs.
- CRUD operations on PostgreSQL.
- Table Inheritance in PostgreSQL.
- Partitioning in PostgreSQL using Table Inheritance.
- Copy table from/to PostgreSQL.
- How to Create/Move/Drop Tablespace.
- Logical (pg dump, pgdumpall), Physical backup (Offline, Online).

- Continuous Archiving and PITR.
- What is Explain plan and how to calculate cost of query.
- Maintenance (Updating planner statistics, Vacuum, Routine Reindex, Cluster).
- What is Transaction Id Wraparound Failure and Vacuum Freeze.
- How to use Pgadmin 4 Interface.
- Uninstall PostgreSQL on Windows and Linux.
- Bonus Module
- Course Extra Documents
- Journey into Postgresql v13
- Postgresql v13 - New Features

Are there any course requirements or prerequisites?

- Basic Knowledge of Database Concepts
- Internet access is required to download PostgreSQL Server
- Working Knowledge on Windows and Linux

Who this course is for:

- Database Administrators
- Accidental DBA's
- Developers
- Architects and Analyst
- Students
- Anyone who is curious about PostgreSQL

Introduction to PostgreSQL

What is PostgreSQL

PostgreSQL is a open-source free and Object Oriented relational database management system. It supports object oriented features i.e classes, objects , inheritance. Developers can use classes & inheritance in code.

PostgreSQL was developed as a research project in **1986 by Mr. Michel Stonebreker** in the university of callifornia in Berkeley.

PostgreSQL is Cross-Platform and runs on multiple operating systems i.e. Linux , windows , macos, FreeBSD etc.

PostgreSQL is complying with ACID Properties (Atomicity , Consistency , Isolation and Durability) that ensures the completion of transaction in all or nothing. For example , if you transferring money to your friend and the server is crashed , in this case database has to commit (money send to your friend account) or rollback(Money comes back to your account) as per the feasibility.

PostgreSQL manages concurrency control by Multi Version Concurrency control (MVCC), for example , if one user is trying to read the row and at the same time other user is trying to **update/modify the same row** than it might be possible that the reader may get inconsistent piece of data. To prevent this, **ISOLATION** is the property which ensure that both the user does not interfere with each other and both will get consistent copy of new one. MVCC takes care of this.

PosgreSQL Object Naming Conventions

Common Names	In PostgreSQL
Tables & Indexes	Relation
Row	Tuple
Column	Attribute
Data block	Page (On the Disk)
Page	Buffer (When block in the memory)

PostgreSQL Limits

Item	Upper Limit	Description
Database Size	Unlimited	
Number Of Databases	4,294,950,911	
Relations per database	1,431,650,303	
Relation Size	32 TB.	
Row per Table	Limited by the number of tuples that can fit onto 4,294,967,295 pages.	
Column per table	1600	

Field Size	1 GB
Identifier Length	63 bytes
Indexes Per table	Unlimited
Columns per indexes	32
Partition Keys	32

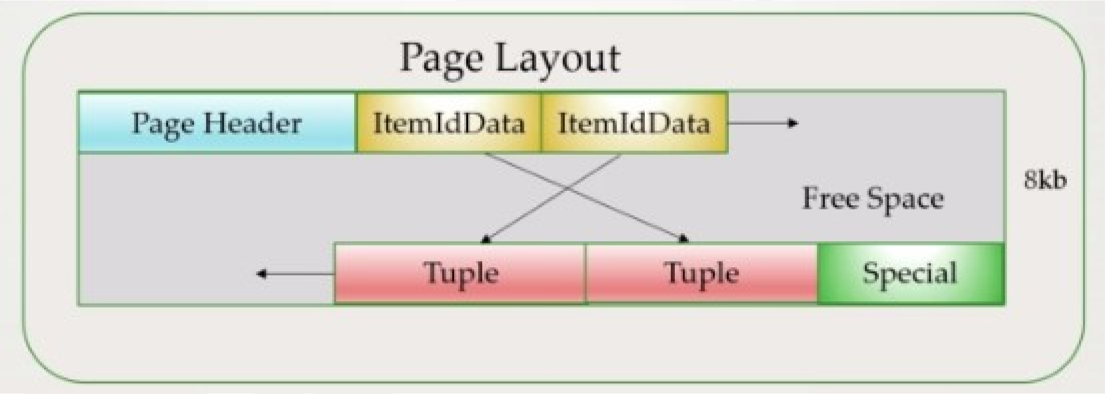
What is Page

- Page is an smallest unit of data storage.
- Every table and index is stored as an array of page (In Sequence) of fixed size.
- Default page size in PostgreSQL is **8 KB.**
- Different page size can be configured (not in same database) while compiling the server.
- All pages are logically equivalent and any row can be stored in any page.

Difference b/n Page and buffer

The **data stored on disk is called Page** , but while reading the data it is fetched back to memory , so the **data loaded back in the memory is called Buffer**.

Understanding PostgreSQL Page Layout



- **Page Header** (**24 byte** Long) – Keeps information about free space , page size and version.
  If insertion performed , database will check the availability of space in page header. Depending on the availability it stores the tuple or it will move to the next page.
- **Item Data ID** (**4 bytes** per item) – Array of pairs pointing to the actual item or tuples.
- **Free Space** – Un-allocated space. New item pointers are allocated from the start of this area and new item from the end.

- **Tuple** – Row Itself.
- **Special** – Holds Information about index access & any relevant information i.e. btree.

Installing PostgreSQL on Windows and Linux

Module Objective:

- Minimal System Requirement
- Installation of PostgreSQL 12 on Windows
- Setting Environment Variables on Windows
- Installation of PostgreSQL on Linux using Yum
- Setting Environment Variables on Linux

Hardware Requirements:

- 1 GHz processor
- 2 GB of RAM
- 512 MB of HDD

Software Requirements:

- User must have administrator privileges on windows system.
- Root or Super user access is required on Linux system.

PostgreSQL installation on Windows 64 Bit

Steps 1:

Navigate to **www.postgresql.org/downloads**

Step 2: Select operating system as Windows.

Step 3: Click on Download the installer

Step 4: Download Version 12 from available postgresql releases for Win x86_64

Step 5: Right click on the setup file and select “Run as Administrator”

Step 6: Navigate through the wizard by selecting appropriate options (initially All).

Step 7: Wait for the installation screen to finish installation.

Step 8: Search for psql interface from the start menu and try to connect to Postgresql.

(Stack bulider is a package manager that contains additional tools that are used for management, migration, replication, connectors, and other tools)

PostgreSQL Setting ENV Variable on Windows 64 Bit

Step 1: Click on Start -> Search for This PC/My PC -> Right click on This PC -> Select Properties -> Advanced -> Environment Variables.

Step 2: In System Variable window search for path.

Step 3: Select Path and Click Edit.

Step 4: Click new and copy the location of postgresql bin folder in the new line and click ok.

Step 5: Click option New in the system variable and enter variable name as “**PGDATA**” and variable value as location of the postgresql data directory.

Step 6: Select okay in all open windows to save & exit out of Environment variable.

PostgreSQL installation on Linux

Step 1: Navigate to www.postgresql.org/downloads

Step 2: Select operating system as Linux.

Step 3: Select appropriate Linux Distribution (In my case Red Hat).

Step 4: Select Postgresql YUM Repository link

Step 5: Select Version 12(or latest) from available postgresql releases.

Step 6: Select “RHEL/Centos 8- x86_64” .

Step 7 : Download “ pgdg-redhat-repo-42.0-11.noarch.rpm” from the list.

Step 8: Install the downloaded rpm using the following syntax on the linux box:

```
rpm -ivh pgdg-redhat-repo-42.0-11.noarch.rpm
```

Step 9 : Disable default postgresql on linux using the following syntax on the linux box: dnf -qy module disable postgresql.

Step 10: Type the following command to list all available postgresql version.

```
$ Yum list module postgresql*
```

Step 11: Look for postgresql v12 version.

```
$ Yum list module postgresql12*
```

Step 12: Install two packages from the postgresql12 list.

```
$Yum install postgresql12-server.x86_64 postgresql12-contrib.X86_64
```

Step 13: Check the installation has completed successfully.

PostgreSQL Setting ENV Variable on Linux

Step 1: Login as Postgres user on the linux box.

Step 2: Edit bash_profile of postgres user using the following command

```
Vi .bash_profile
```

Step 3: Add the below mentioned lines to the existing bash_profile.

```
PATH=$PATH:HOME/bin
export PATH
export PATH=/usr/pgsql-12/bin:$PATH  ## put relevant path as per OS
PG_DATA=/var/lib/pgsql/12/data
```

```
export PGDATA
```

Step 4: Save and quit the bash_profile file.

Note: On windows OS , after installation it will create and initiate the cluster automatically. But on linux it required to be created manually.

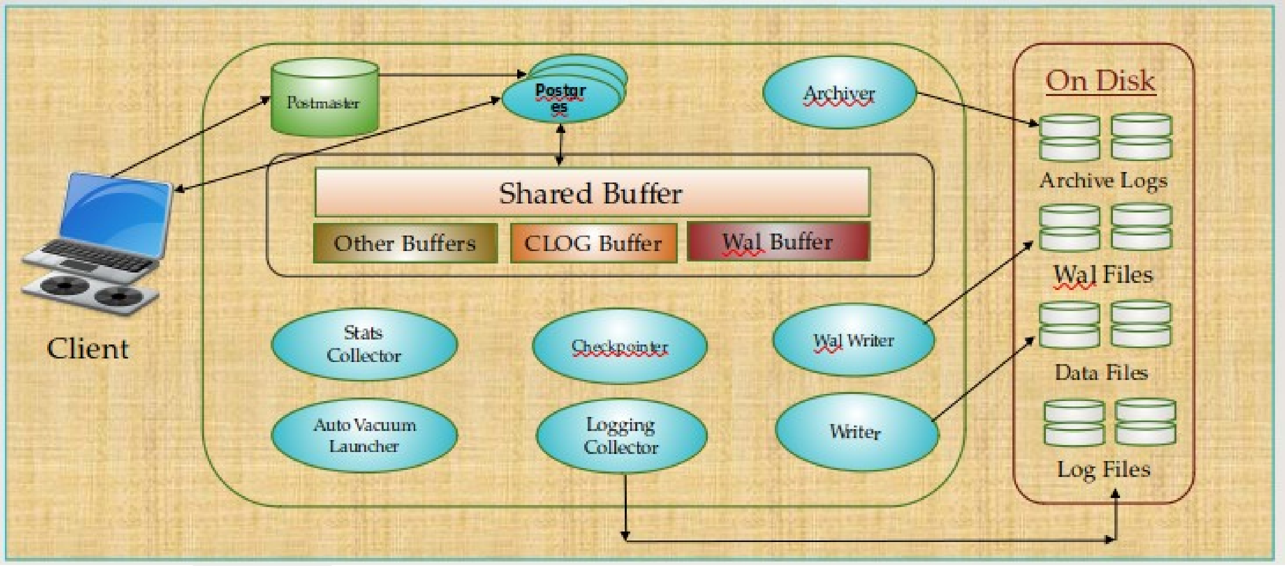
PostgreSQL Architecture

PostgreSQL Architecture Fundamentals

- PostgreSQL is a relational database management system with a **client-server** architecture. Many clients/users (All of us) requests/Receives services from host computer (Server) called client server model. We run a application on our PC and send the request to centralized server and a server responds to us.
- PostgreSQL uses "**process per-user**" client/server model. Each user is given a particular process on connection. Which will stay there **till the life of connection**.
- PostgreSQL's has a set of processes and memory structures which constitutes an instance. The instance is a combination of memory structures and processes. On a server there can be any number of instances and each instance will have its own memory structure and processes. Memory structure & processes can not be shared within different instances.

- Programs run by clients connect to the server instance and request read and write operations.
- **Default port of PostgreSQL is 5432**. Default port can be change while installation or by changing in configuration files after installation. Restart of service will be required for this operation.

Process and Memory Architecture



POSTMASTER is the first process to get started when we starts PostgreSQL. It is assigned with the shared memory which combine of Shared Buffer , WAL buffer, CLOG buffer and other. Postmaster will starts other processes I.e Stats Collector, Checkpoints, WAL Writer, BGWriter(or Writer) Logging Collector,Auto Vacume Launcher.

How the PG architecture Works : User (Client) Sends a request from workstation to connect to the PG Server. The request is picked up by postmaster to check validity of the credentials, IP address from configuration files. On success, postmaster will start a new process and handover the connection to a new process. Which in turn will connect to the shared buffer , results the user communicate to the database. Connection is kept alive till the user is logged in and user can read and write to the database.

Assume the scenario , user sends SELECT request , data loaded from the datafiles into the shared buffer and the results are sent back to the client by the postgres process.

INSERT, UPDATE and DELETE we make changes in data are called transactions. Any change request made by connected user the changes are made in shared buffer and the copy of the transaction is recorded in WAL buffer,when user commits changes WAL buffer will invoke the Wal Writer which will write all the changes in WAL files.

The changes made in shared buffer (Actual data) is marked as commited and stays in shared buffer.

CHECKPOINT is the process which invokes in **every 5 Min**.(Default, can change as per requirement) is signals to the **BGWRITER** (Writer) to write all the commited data buffers (dirty Buffers) into the data-files. CheckPoint ensure that all the data-files are in sync.

Generally in production system we enable the additional process i.e. **ARCHIVE**. It keeps checking the WAL files. If the WAL file gets full or switching between them, archive process copies the Wal files to Archive files. These archive files will help us in doing **PITR** (Point in time Recovery) and provide additional level of security. It should not be enables **in test or development environment** because it **consumes lots of space**.

CLOG Buffer : All commited transactions are marked in this buffer helps us to check all the commited transactions.

STATS COLLECTOR : collects information and report of server activities and update this information to database dictionary.

AUTO VACUUM PROCESS : when enabled its responsibility to run Vacuum demon on fragmented or bloated tables. Tables gets fragmented due to lots of update, insert or delete operations.

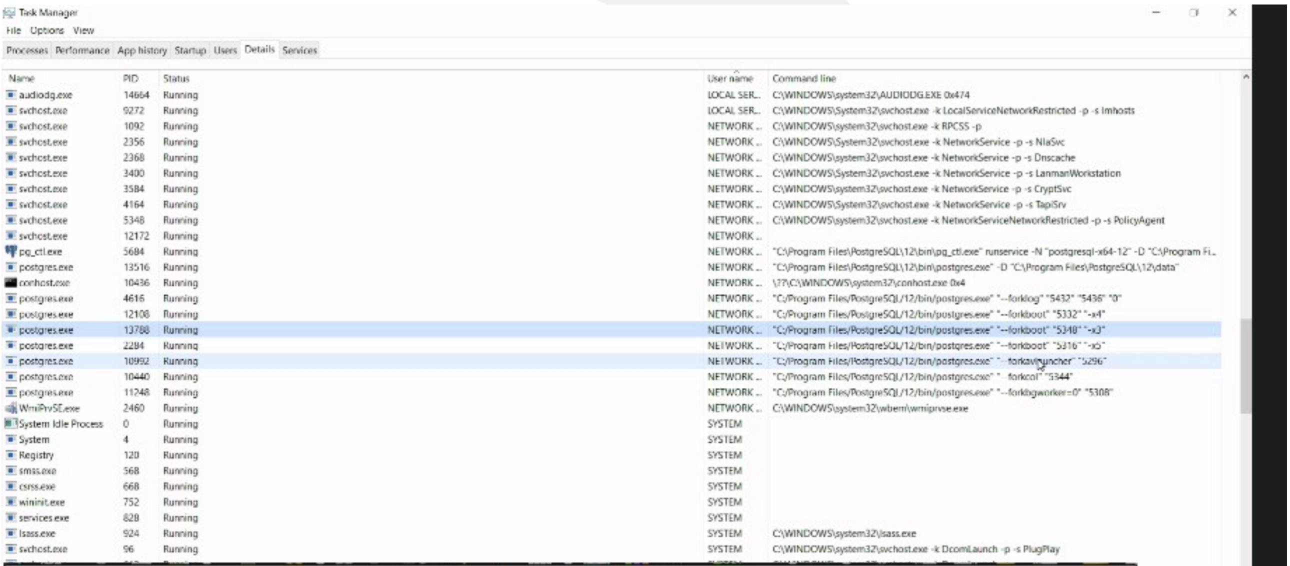
LOG COLLECTOR : Responsible to **collect the error logs** . For example I starts the Postgres service and it doesn't starts. To seek the cause or error I go to the error log files. Log collector writes all the log information to the log files.

Postmaster – Supervisor process

- Postmaster is the **first process** which gets started in PostgreSQL
- Postmaster acts as supervisor process, whose job is to monitor, start, **restart some processes if they die**.
- Postmaster acts a listener and receive new connection request from the client. It listens the new connection request from users and **create new background process for newly received connection request**.
- Postmaster is **responsible for Authentication and Authorization** of all incoming request. When the new connection request received it checks the validity, IP address by checking the configuration files. Then it checks the database grant which user wants to access.
- Postmaster spawns a new process called Postgres for each new connection.

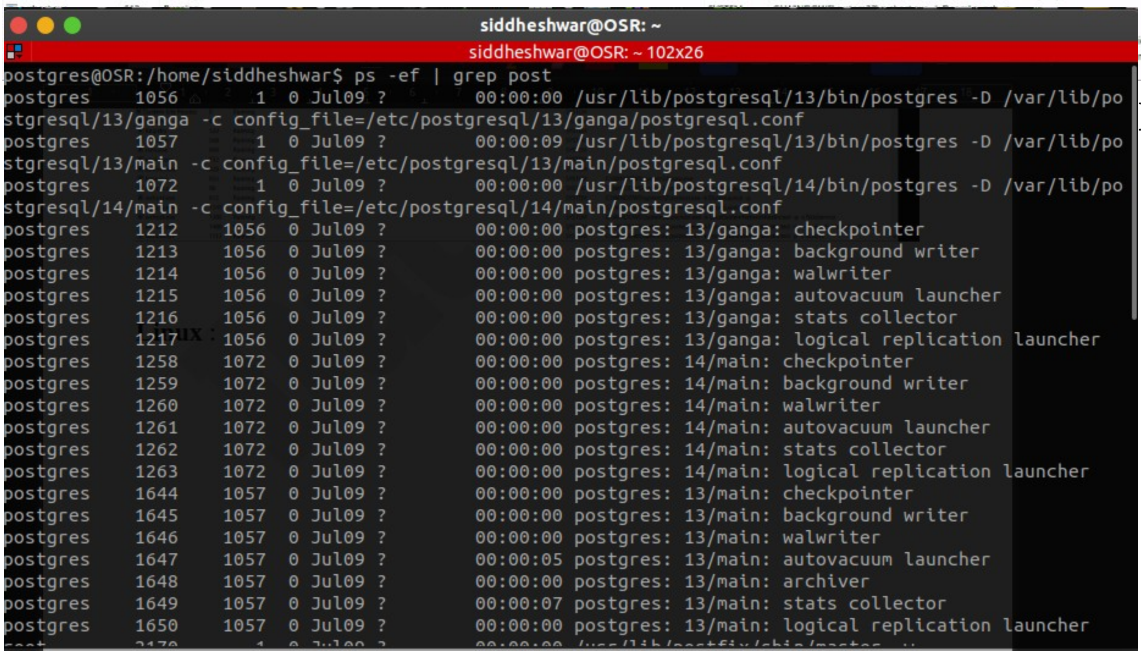
Checking the PostgreSQL – postmaster and other process status

Windows : Goto taskbar > Right Click >> Open Task Manager , All OS running processes will be displayed . Write click on status and add column “Command Line”. You can see the all processes associated with postgres. If one process is killed postmaster will restart the killed process.



Name	PID	Status	User name	Command line
audiodg.exe	14664	Running	LOCAL SER...	C:\WINDOWS\system32\AUDIODG.EXE 0x474
svchost.exe	9272	Running	LOCAL SER...	C:\WINDOWS\System32\svchost.exe -k LocalServiceNetworkRestricted -p -s lmhosts
svchost.exe	1092	Running	NETWORK ...	C:\WINDOWS\system32\svchost.exe -k RPCSS -p
svchost.exe	2356	Running	NETWORK ...	C:\WINDOWS\System32\svchost.exe -k NetworkService -p -s NlaSvc
svchost.exe	2368	Running	NETWORK ...	C:\WINDOWS\system32\svchost.exe -k NetworkService -p -s Dnscache
svchost.exe	3400	Running	NETWORK ...	C:\WINDOWS\System32\svchost.exe -k NetworkService -p -s LanmanWorkstation
svchost.exe	3584	Running	NETWORK ...	C:\WINDOWS\system32\svchost.exe -k NetworkService -p -s CryptSvc
svchost.exe	4164	Running	NETWORK ...	C:\WINDOWS\System32\svchost.exe -k NetworkService -p -s TapiSrv
svchost.exe	5348	Running	NETWORK ...	C:\WINDOWS\system32\svchost.exe -k NetworkServiceNetworkRestricted -p -s PolicyAgent
svchost.exe	12172	Running	NETWORK ...	
pg_ctl.exe	5684	Running	NETWORK ...	"C:\Program Files\PostgreSQL\12\bin\pg_ctl.exe" runservice -N "postgresql-x64-12" -D "C:\Program Fi...
postgres.exe	13516	Running	NETWORK ...	"C:\Program Files\PostgreSQL\12\bin\postgres.exe" -D "C:\Program Files\PostgreSQL\12\data"
conhost.exe	10436	Running	NETWORK ...	\??\C:\WINDOWS\system32\conhost.exe 0x4
postgres.exe	4616	Running	NETWORK ...	"C:/Program Files/PostgreSQL/12/bin/postgres.exe" "--forklog" "5432" "5436" "0"
postgres.exe	12108	Running	NETWORK ...	"C:/Program Files/PostgreSQL/12/bin/postgres.exe" "--forkboot" "5332" "-x4"
postgres.exe	13788	Running	NETWORK ...	"C:/Program Files/PostgreSQL/12/bin/postgres.exe" "--forkboot" "5348" "-x3"
postgres.exe	2284	Running	NETWORK ...	"C:/Program Files/PostgreSQL/12/bin/postgres.exe" "--forkboot" "5316" "-x5"
postgres.exe	10992	Running	NETWORK ...	"C:/Program Files/PostgreSQL/12/bin/postgres.exe" "--forkavlauncher" "5296"
postgres.exe	10440	Running	NETWORK ...	"C:/Program Files/PostgreSQL/12/bin/postgres.exe" "--forkcol" "5344"
postgres.exe	11248	Running	NETWORK ...	"C:/Program Files/PostgreSQL/12/bin/postgres.exe" "--forkbgworker=0" "5308"
WmiPrvSE.exe	2460	Running	NETWORK ...	C:\WINDOWS\system32\wbem\wmiprvse.exe
System Idle Process	0	Running	SYSTEM	
System	4	Running	SYSTEM	
Registry	120	Running	SYSTEM	
smss.exe	568	Running	SYSTEM	
csrss.exe	668	Running	SYSTEM	
wininit.exe	752	Running	SYSTEM	
services.exe	828	Running	SYSTEM	
lsass.exe	924	Running	SYSTEM	C:\WINDOWS\system32\lsass.exe
svchost.exe	96	Running	SYSTEM	C:\WINDOWS\system32\svchost.exe -k DcomLaunch -p -s PlugPlay

Linux :



```
postgres@OSR:/home/siddheshwar$ ps -ef | grep post
postgres    1056       1  0 Jul09 ?        00:00:00 /usr/lib/postgresql/13/bin/postgres -D /var/lib/po
stgresql/13/ganga -c config_file=/etc/postgresql/13/ganga/postgresql.conf
postgres    1057       1  0 Jul09 ?        00:00:09 /usr/lib/postgresql/13/bin/postgres -D /var/lib/po
stgresql/13/main -c config_file=/etc/postgresql/13/main/postgresql.conf
postgres    1072       1  0 Jul09 ?        00:00:00 /usr/lib/postgresql/14/bin/postgres -D /var/lib/po
stgresql/14/main -c config_file=/etc/postgresql/14/main/postgresql.conf
postgres    1212    1056  0 Jul09 ?        00:00:00 postgres: 13/ganga: checkpointer
postgres    1213    1056  0 Jul09 ?        00:00:00 postgres: 13/ganga: background writer
postgres    1214    1056  0 Jul09 ?        00:00:00 postgres: 13/ganga: walwriter
postgres    1215    1056  0 Jul09 ?        00:00:00 postgres: 13/ganga: autovacuum launcher
postgres    1216    1056  0 Jul09 ?        00:00:00 postgres: 13/ganga: stats collector
postgres    1217    1056  0 Jul09 ?        00:00:00 postgres: 13/ganga: logical replication launcher
postgres    1258    1072  0 Jul09 ?        00:00:00 postgres: 14/main: checkpointer
postgres    1259    1072  0 Jul09 ?        00:00:00 postgres: 14/main: background writer
postgres    1260    1072  0 Jul09 ?        00:00:00 postgres: 14/main: walwriter
postgres    1261    1072  0 Jul09 ?        00:00:00 postgres: 14/main: autovacuum launcher
postgres    1262    1072  0 Jul09 ?        00:00:00 postgres: 14/main: stats collector
postgres    1263    1072  0 Jul09 ?        00:00:00 postgres: 14/main: logical replication launcher
postgres    1644    1057  0 Jul09 ?        00:00:00 postgres: 13/main: checkpointer
postgres    1645    1057  0 Jul09 ?        00:00:00 postgres: 13/main: background writer
postgres    1646    1057  0 Jul09 ?        00:00:00 postgres: 13/main: walwriter
postgres    1647    1057  0 Jul09 ?        00:00:05 postgres: 13/main: autovacuum launcher
postgres    1648    1057  0 Jul09 ?        00:00:00 postgres: 13/main: archiver
postgres    1649    1057  0 Jul09 ?        00:00:07 postgres: 13/main: stats collector
postgres    1650    1057  0 Jul09 ?        00:00:00 postgres: 13/main: logical replication launcher
```

Utility Processes

- **Bgwriter\Writer** : Periodically writes the dirty buffer to a data file.

- **Wal Writer** : Write the WAL buffer to the WAL file when the **commit** request received.

- **Checkpointer** : Checkpoint is invoked every 5 minute(default) or when **max_wal_size** (PostgreSQL.conf) value is exceeded. The check pointer signals to **BGWriter** to sync all the commited buffers from the shared buffer area to the data files.

- **Auto vacuum** : Responsible to carry vacuum operations on bloated tables.(If Enabled).

- **Statscollector**: Responsible for collection and reporting of information about server activity then update the information to optimizer dictionary((pg_catalog) .

- **Logwriter\Logger**: Write the error message to the log file.

- **Archiver (Optional)**:When in Archive.log mode, copy the WAL file to the specified directory.

Memory Segments of PostgreSQL

- Shared Buffers
- Wal Buffers
- Clog Buffers
- Work Memory
- Maintenance Work Memory
- Temp Buffers

Shared Buffer :

- User cannot access the datafile directly to read or write any data.
- Any select, insert, update or delete to the data is done via shared buffer area.
- The data that is written or modified in this location is called “Dirty data“.
- Dirty data is written to the data files located in physical disk through background writer process.

  Shared Buffers are controlled by parameter named: shared_buffer (Default , **128MB**) located in postgresql.conf file. We can change size as per need.

Wal Buffer:

- Write ahead logs buffer is also called as “Transaction log Buffers”.

- WAL data is the metadata information about changes to the actual data, and is sufficient to reconstruct actual data during database recovery operations.
- WAL data is written to a set of physical files in persistent location called "WAL segments" or "checkpoint segments".
- Wal buffers are flushed from the buffer area to wal segments by wal writer.
- Wal buffers **memory allocation** is controlled by the **wal_buffers default -1** parameter in postgresql.conf.

Clog and other buffers:

- **CLOG** stands for "commit log", and the CLOG buffers is an area in operating system RAM dedicated to hold commit log pages.

- The commit logs have commit status of all transactions and indicate whether or not a transaction has been completed (committed).

- **Work Memory** is a memory reserved for either a single sort or hash table (Parameter : **work_mem - default 4MB** in PostgreSQL.conf) suppose we runs a query with order by clause, distinct , such sorting & merge joins operations are performed in Work Memory.

- **Maintenance Work Memory** is allocated for Maintenance work i.e. Vaccuming , index rebuild, (Parameter : **maintenance_work_mem default -1**).

- **Temp Buffers** are used for access to temporary tables in a user session during large sort and hash table. (Parameter : **temp_buffers** , **Default 8mb**).

Physical Files:

- **Data Files**: It is a file which is use to store data. It does not contain any instructions or code to be executed.
- **Wal Files** : Write ahead log file, where all committed transactions are written first here before writing to datafile.
- **Log Files**: All server messages, including stderr, csvlog and syslog are logged in log files.
- **Archive Logs(Optional):** Data from wal segments are written on to archive log files to be used for recovery purpose.

Database Clusters

Database Cluster:

- Database cluster is a collection of databases that is managed by a single instance on a server.
- **Initdb** creates a new PostgreSQL database cluster. It makes new directories set for the cluster.
- Creating a database cluster consists of creating the directories in which the data is store. We call this the ”**data directory**”.
- We have to first initialize the storage area on the disk before we begin any operation on the database.
- Location of Data Directory :
  Linux : /var/lib/pgsql/data (or according to version)
  Windows : C:\Program Files\PostgreSQL\12\data ()

Initdb Syntax :

- We have to be logged in as PostgreSQL user to execute the below commands.
- There are two way to initialize database

Syntax:

initdb -D /usr/local/pgsql/data

pg_ctl -D /usr/local/pgsql/data initdb

-D = refers to the data directory location.

-W = we can use this option to force the super user to provide password before initialize db

Start\Stop Cluster:

- Start Cluster Syntax :
  Linux : *systemctl start postgresql-<Ver>*
  Windows : *pg_ctl –D "C:\Program Files\PostgreSQL\12\data" start*
- Stop Cluster Syntax :
  Linux : *systemctl stop postgresql-<Ver>*
  Windows : : *pg_ctl stop -D "C:\Program Files\PostgreSQL\12\data" –m shutdown mode*

Types of Shutdown:

Smart (SIGTERM): the server disallows new connections, but lets existing sessions end their work normally. It shuts down only after all of the sessions are terminated.

Fast :(**SIGINT** Default) : The server disallows new connections and abort their current transactions and exits gracefully.

Immediate (**SIGQUIT**): Quits/aborts without proper shutdown which lead to recovery from the WAL files on next startup .

Difference between Reload and Restart:

- When we make changes to server parameters, we need to reload the configuration for them to take effect.
- Reload will just reload the new configurations, without restarting the service.
- Few configuration changes in server parameters, Do not get reflected until we restart the service.
- Restart gracefully shutdown all activity, relinquishes the resource, close all open files and start again with new configuration.

Reload\Restart Cluster:

Syntax for Reload of Cluster:

On linux : *systemctl reload posgresql-12*

On windows: *pg_ctl reload*

Syntax for Restart of Cluster:

On linux : *systemctl restart postgresql-12*

On Windows : *pg_ctl restart*

Psql Command line:

SQL : SELECT pg_reload_conf(); (Irrespective of Env)

Pg_Controldata

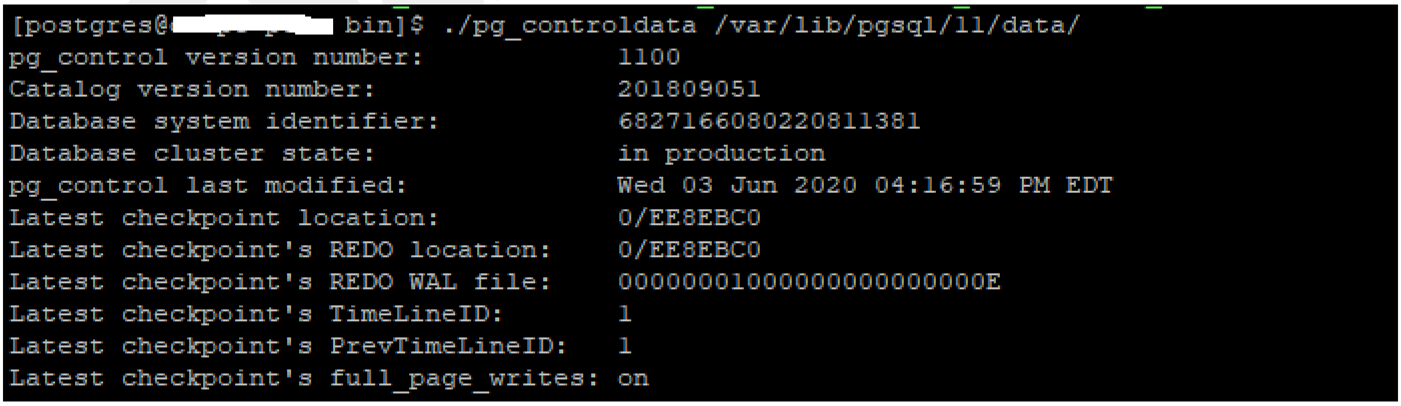
Pg_controldata – Information about cluster.

Syntax : ./pg_controldata /var/lib/pgsql/11/data/

configure path if not working

PATH="/usr/lib/postgresql/13/bin/:$PATH

Syntax : pg_controldata /var/lib/postgresql/13/main



```
[postgres@[redacted] bin]$ ./pg_controldata /var/lib/pgsql/11/data/
pg_control version number:            1100
Catalog version number:               201809051
Database system identifier:           6827166080220811381
Database cluster state:               in production
pg_control last modified:             Wed 03 Jun 2020 04:16:59 PM EDT
Latest checkpoint location:           0/EE8EBC0
Latest checkpoint's REDO location:    0/EE8EBC0
Latest checkpoint's REDO WAL file:    00000001000000000000000E
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
```

FULL Output:

```
pg_control version number:           1300
Catalog version number:              202007201
Database system identifier:          7002519706457325072
Database cluster state:              in production
pg_control last modified:            Saturday 28 May 2022 02:38:38 PM
Latest checkpoint location:          4A/6E000110
Latest checkpoint's REDO location:    4A/6E0000D8
Latest checkpoint's REDO WAL file:    000000020000004A0000006E
Latest checkpoint's TimeLineID:       2
Latest checkpoint's PrevTimeLineID:   2
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:94813
Latest checkpoint's NextOID:          35281
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        478
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  94813
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:            Saturday 28 May 2022 02:38:37 PM
Fake LSN counter for unlogged rels:   0/3E8
Minimum recovery ending location:     0/0
Min recovery ending loc's timeline:   0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
wal_level setting:                    replica
```

```
wal_log_hints setting:              off
max_connections setting:             100
max_worker_processes setting:          8
max_wal_senders setting:              10
max_prepared_xacts setting:            0
max_locks_per_xact setting:           64
track_commit_timestamp setting:       off
Maximum data alignment:                8
Database block size:                8192
Blocks per segment of large relation: 131072
WAL block size:                     8192
Bytes per WAL segment:              16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:       1996
Size of a large-object chunk:        2048
Date/time type storage:             64-bit integers
Float8 argument passing:            by value
Data page checksum version:            0
Mock authentication nonce:
b700f489ed82c1424c2f16ecfd38a56a2b2c48656d92b034100e78bbefc7afd
```

Database Directory Layout

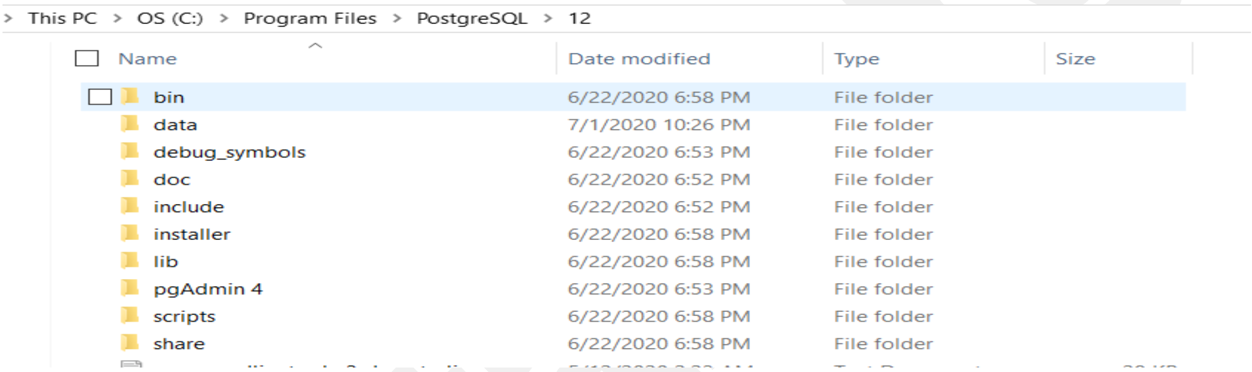
Module Objective:

- Installation Directory Layout
- Overview of Installation Directory & Demo
- Database Directory Layout
- Overview of Database Directory & Demo

- Base Directory
- Overview of Base Directory & Demo

Installation Directory Layout:

- PostgreSQL is typically installed to /usr/local/pgsql or /var/lib/pgsql on linux.
- C:\Program Files\PostgreSQL\<version number> on windows.



> This PC > OS (C:) > Program Files > PostgreSQL > 12

Name	Date modified	Type	Size
bin	6/22/2020 6:58 PM	File folder	
data	7/1/2020 10:26 PM	File folder	
debug_symbols	6/22/2020 6:53 PM	File folder	
doc	6/22/2020 6:52 PM	File folder	
include	6/22/2020 6:52 PM	File folder	
installer	6/22/2020 6:58 PM	File folder	
lib	6/22/2020 6:58 PM	File folder	
pgAdmin 4	6/22/2020 6:53 PM	File folder	
scripts	6/22/2020 6:58 PM	File folder	
share	6/22/2020 6:58 PM	File folder	

- bin-> programs(createdb, initdb,createuser,etc)
- data -> Data Directory
- Doc --> Documentation
- Include --> Header Files
- Installer -> Installer files
- Scripts --> scripts like runpsql, serverctl vbscript files
- Share -> Sample configuration files
- pgadmin - pgadmin files.

Database (Data) Directory Layout:

Directory Name	Description
Base	Subdirectory containing per-database subdirectories

Current_logfiles	File recording the log file(s) currently written to by the logging collector
Global	Subdirectory containing cluster-wide tables, such as pg_database,pg_tablespace,pg_index etc
pg_commit_ts	Subdirectory containing transaction commit timestamp data= 9.5 and later, track_commit_timestamp
pg_dynshmem	Subdirectory containing files used by the dynamic shared memory subsystem
pg_logical	Subdirectory containing status data for logical decoding
pg_multixact	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify	Subdirectory containing LISTEN/NOTIFY status data
pg_replslot	Subdirectory containing replication slot data
pg_serial	Subdirectory containing information about committed serializable transactions
Log	All error logs kept in this directory.
pg_snapshots	Subdirectory containing exported snapshots
pg_stat	Subdirectory containing permanent files for the statistics subsystem
pg_stat_tmp	Subdirectory containing temporary files for the statistics subsystem
pg_subtrans	Subdirectory containing subtransaction status data
pg_tblspc	Subdirectory containing symbolic links to tablespaces

pg_twophase	Subdirectory containing state files for prepared transactions
pg_wal	Subdirectory containing WAL (Write Ahead Log) files
pg_xact	Subdirectory containing transaction commit status data, transaction metadata logs
Pg_ident.conf	User name maps are defined in the ident map file.user name map can be applied to map the operating system user name to a database user.
postgresql.auto.conf	A file used for storing configuration parameters that are set by ALTER SYSTEM
postmaster.opts	A file recording the command-line options the server was last started.
postmaster.pid	A lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (empty on Windows), first valid listen_address (IP address or *, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown)
PG_VERSION	A file containing the major version number of PostgreSQL

Base_Directory:

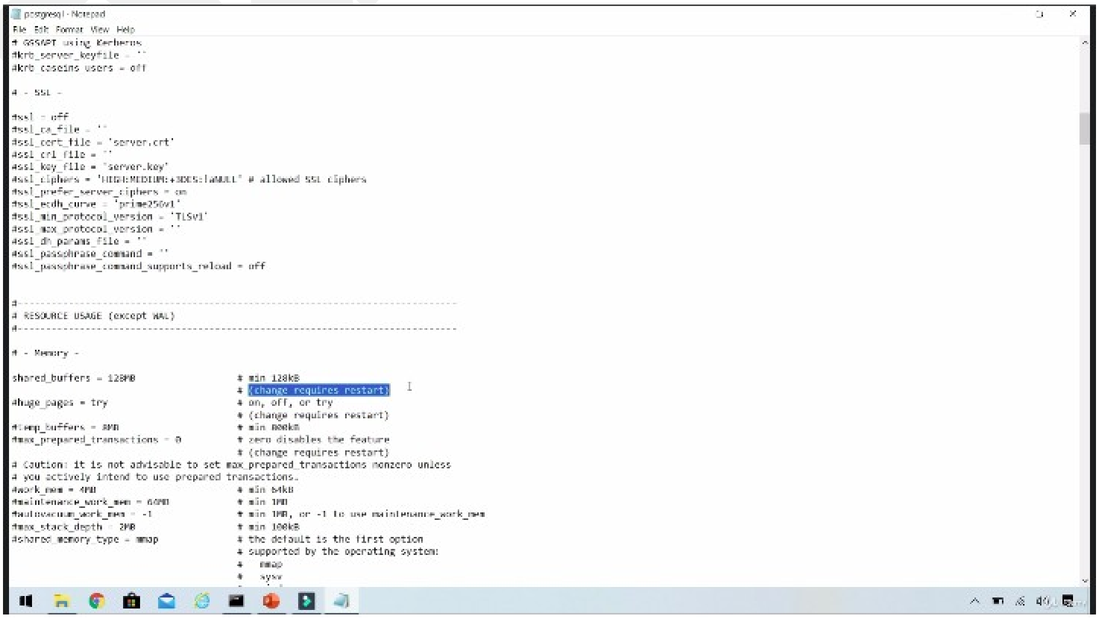
- Contains databases, that represented as directories named after their object identifier (OID) not by the database name. Database name and OID mapping can be found on pg_database table.

- Template 1 always has oid 1.
- Syntax to find oid of database:
  Syntax: select datname,oid from pg_database;

PostgreSQL Configuration Files

Postgresql.conf File

- Postgresql.conf file contains parameters to help configure and manage performance of the database server.
- Initdb installs a default copy of postgresql.conf and is usually located in data directory. (in 13 it is in /etc)
- The file follows one parameter per line format.
- Parameters which requires restart are clearly marked in the file.
- Many parameter needs a server restart to take effect.



```
# GSSAPI using Kerberos
#krb_server_keyfile = ''
#krb_caseins_users = off

# - SSL -

#ssl = off
#ssl_ca_file = ''
#ssl_cert_file = 'server.crt'
#ssl_crl_file = ''
#ssl_key_file = 'server.key'
#ssl_ciphers = 'HIGH:MEDIUM:+3DES:!aNULL' # allowed SSL ciphers
#ssl_prefer_server_ciphers = on
#ssl_ecdh_curve = 'prime256v1'
#ssl_min_protocol_version = 'TLSv1'
#ssl_max_protocol_version = ''
#ssl_dh_params_file = ''
#ssl_passphrase_command = ''
#ssl_passphrase_command_supports_reload = off


#------------------------------------------------------------------------------
# RESOURCE USAGE (except WAL)
#------------------------------------------------------------------------------

# - Memory -

shared_buffers = 128MB                  # min 128kB
                                        # (change requires restart)
#huge_pages = try                       # on, off, or try
                                        # (change requires restart)
#temp_buffers = 8MB                     # min 800kB
#max_prepared_transactions = 0          # zero disables the feature
                                        # (change requires restart)
# Caution: it is not advisable to set max_prepared_transactions nonzero unless
# you actively intend to use prepared transactions.
#work_mem = 4MB                         # min 64kB
#maintenance_work_mem = 64MB            # min 1MB
#autovacuum_work_mem = -1               # min 1MB, or -1 to use maintenance_work_mem
#max_stack_depth = 2MB                  # min 100kB
#shared_memory_type = mmap              # the default is the first option
                                        # supported by the operating system:
                                        #   mmap
                                        #   sysv
```

The path of config file is data directory till Postgresql version 12. in version 13 config files has been moved to *etc/*postgresql/<version> folder in linux/ubuntu.

We can query the path by below command.

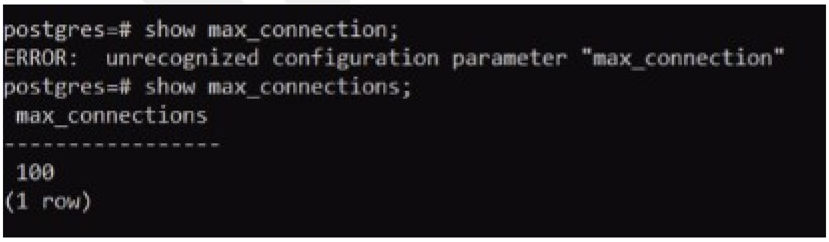
psql -U postgres -c 'SHOW config_file'

or from ubuntu user :

sudo -u postgres psql -c 'SHOW config_file'

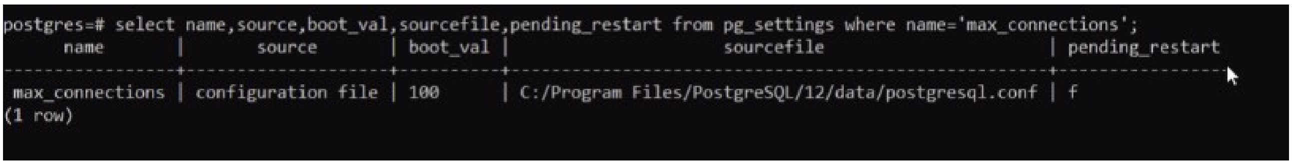
The parameters in postgresql.conf file can be checked by SQL without opening the file.

i.e. : SHOW max_connections;



```
postgres=# show max_connection;
ERROR:  unrecognized configuration parameter "max_connection"
postgres=# show max_connections;
 max_connections
-----------------
 100
(1 row)
```

Or we can check in more detail by the query below
select name,source_boot_val, sourcefile,pending_restart from ***pg_setting*** *where name=’max_connections’;*

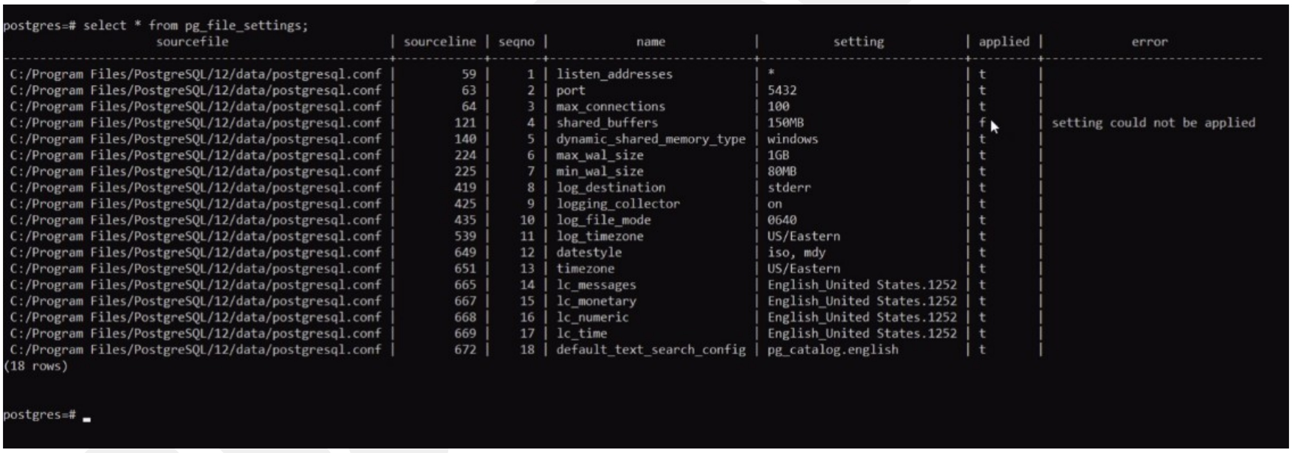


```
postgres=# select name,source,boot_val,sourcefile,pending_restart from pg_settings where name='max_connections';
      name       |       source       | boot_val |                      sourcefile                      | pending_restart
-----------------+--------------------+----------+------------------------------------------------------+-----------------
 max_connections | configuration file | 100      | C:/Program Files/PostgreSQL/12/data/postgresql.conf | f
(1 row)
```

Pending_restart (t-true, f-false) means parameter was changed but restart is not done so old value is applicable currently. Activated by reloading the config I.e *select pg_reload_conf();*

pg_settings table can be described by command *\d pg_settings.*

Another catalog table is **pg_file_Settings** gives us the summary to plan the parameter change.



```
postgres=# select * from pg_file_settings;
                     sourcefile                      | sourceline | seqno |           name            |          setting           | applied |            error
-----------------------------------------------------+------------+-------+---------------------------+----------------------------+---------+------------------------------
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |         59 |     1 | listen_addresses          | *                          | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |         63 |     2 | port                      | 5432                       | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |         64 |     3 | max_connections           | 100                        | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        121 |     4 | shared_buffers            | 150MB                      | f       | setting could not be applied
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        140 |     5 | dynamic_shared_memory_type | windows                   | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        224 |     6 | max_wal_size              | 1GB                        | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        225 |     7 | min_wal_size              | 80MB                       | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        419 |     8 | log_destination           | stderr                     | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        425 |     9 | logging_collector         | on                         | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        435 |    10 | log_file_mode             | 0640                       | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        539 |    11 | log_timezone              | US/Eastern                 | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        649 |    12 | datestyle                 | iso, mdy                   | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        651 |    13 | timezone                  | US/Eastern                 | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        665 |    14 | lc_messages               | English_United States.1252 | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        667 |    15 | lc_monetary               | English_United States.1252 | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        668 |    16 | lc_numeric                | English_United States.1252 | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        669 |    17 | lc_time                   | English_United States.1252 | t       |
 C:/Program Files/PostgreSQL/12/data/postgresql.conf |        672 |    18 | default_text_search_config | pg_catalog.english        | t       |
(18 rows)


postgres=# _
```

PG_setting TABLE:

- Pg_settings table provides access to run-time parameters of the server.
- It is a alternate interface to SHOW command.
- Pg_file_settings provides a summary of the contents of the server's configuration file.

- This view is helpful for checking whether planned changes in the configuration files will work
- Each “name = value” entry appearing in the files has a corresponding applied column.

PostgreSQL.auto.conf file:

- This file hold settings provided through **ALTER SYSTEM** command.
- Settings in postgresql.auto.conf overrides the settings in postgresql.conf.
- ”Alter system” command provides a SQL-accessible means of changing global defaults.
- Syntax : ALTER SYSTEM SET configuration_parameter = 'value'
- Syntax to reset : ALTER SYSTEM RESET configuration_parameter;
- Syntax to reset all : ALTER SYSTEM RESET ALL;
- It does not make any change in main config file postgresql.conf but it changes postgresql.auto.conf. When we restart or reload the conf , the configuration applied as in postgresql.conf if same parameter is not avilable in postgresql.auto.conf.

pg_ident.conf file:

- Configuration to indicate which map to use for each individual connection.
- User name maps are defined in the ident map file.
- Pg_ident.conf file is read on start-up and any changes needs pg_ctl reload

- Operating system user that initiated the connection might not be the same as the database user.
- User name map can be applied to map the operating system user name to a database user.
- **pg_ident.conf** is used in conjunction with pg_hba.conf.

Sample :

```
# MAP                IDENT      POSTGRESQL_USERNAME

sales                rmartin    sales

sales                jpenny     sales

audit                auditor    sales

audit                auditor    postgres
```

The file shown in allows either of the system users **rmartin** or **jpenny** to connect as the PostgreSQL **sales user**, and allows the system user named **auditor** to connect to PostgreSQL as either **sales**, or **postgres**.

pg_hba.conf File :

- Enables client authentication between the PostgreSQL server and the client application.
- HBA means host based authentication. Postmaster checks the authorization in pg_hba.conf configuration file while connecting to new connection.
- When postgreSQL receives a connection request it will check the ”pg_hba.conf” file to verify that the machine from which the

application is requesting a connection has rights to connect to the specified database.

- PostgreSQL rejects a connection if an entry is not found in pg_hba.conf file.

Sample Of the pg_hba.conf File :

```
TYPE         DATABASE         USER     ADDRESS
     METHOD

# IPv4 local connections:

 host       all              all      127.0.0.1/32     md5
                                           (/32 is a network mask)
# IPv6 local connections:

 host       all              all      ::1/128          trust

# Allow replication connections from localhost, by a user with the

# replication privilege.

Host        replication      all      127.0.0.1/32     trust

host        replication      all      ::1/128          trust
```

TYPE

Host: is used to specify remote hosts that are allowed to connect to the PostgreSQL server. PostgreSQL's postmaster backend must be running with the -i option (TCP/IP) in order for a host entry to work correctly.

Local : is semantically the same as a host entry. However, you do not need to specify a host that is allowed to connect. The local entry is used for client connections that are initiated from the same machine that the PostgreSQL server is operating on.

Hostssl: is user to specify hosts (remote or local) that are allowed to connect to the PostgreSQL server using SSL.

DATABASE

This is the database name that the specified host is allowed to connect to. The database keyword has three possible values:

All : keyword specifies that the client connecting can connect to any database the PostgreSQL server is hosting.

Same user : keyword specifies that the client can only connect to a database that matches the clients authenticated user name.

Name : Client can only connect to the database as specified by name.

IP ADDR/NET MASK

The ip_addr and netmask fields specify either a specific IP address, or range of IP addresses, that are allowed to connect to the PostgreSQL server.

Range can by specified by describing an IP network with an associated netmask.

For single IP address the netmask field should be set to 255.255.255.255.

AUTHENTICATION

The Authentication method specifies the type of authentication the server should use for a user trying to connect to PostgreSQL.

Trust :This method allows any user from the defined host to connect to a PostgreSQL database without the use of a password, as any PostgreSQL user. You are trusting the host-based authentication with the use of this method, and any user on the specified host. This is a dangerous condition if the specified host is not a secure machine, or provides access to users unknown to you.

Reject : This method automatically denies access to PostgreSQL for that host or user. This can be a prudent setting for sites that you know are never allowed to connect to your database server.

Password :This method specifies that a password must exist for a connecting user. The use of this method will require the connecting user to supply a password that matches the password found in the database.

Crypt : This method is similar to the password method. When using crypt, the password is not sent in clear text, but through a simple form of encryption. The use of this method is not very secure, but is better than using the clear text password method.

Krb4, krb5 : This methods are used to specify Version 4 or 5 of the Kerberos authentication system.

Ident : This method specifies that an ident map should be used when a host is requesting connections from a valid IP address listed in the pg_hba.conf file. This method requires one option.

The required option may be either the special term sameuser, or a named map that is defined within the pg_ident.conf file.

pg_hba modification

- Stop postgresql on the source machine.
- Edit pg_hba.conf file and add the entry of client.
- Change the authentication method to Trust or md5(depending on requirement)
- Edit/ensure parameter in postgresql.conf to listen_addresses = '*‘ or ip address.
- Start postgres on the source machine.
- Connection psql -U postgres –h hostname from client.
- Depending on the authentication method chosen the client may or may not prompt for password.

DATABASE - CREATE / DROP

Module Objective:

- Create database – Psql / createdb utility
- Drop database – Psql/ dropdb utility
- Create user – Psql/ createuser utility/ Interactive
- Drop user - Psql/ dropuser utility
- Privileges in PostgreSQL
- Grants and Revoke Access
- What is schema and its benefits?
- Create/ drop schema
- Schema Search Path

Create database Psql / createdb utility:

- Database is an organized collection of structured information, or data, typically stored and accessed electronically from a computer system.
- Syntax from psql : Create database databasename owner ownername;
- If we requires to create clone (i.e. SNAPSHOT) of database, we can add WITH TEMPLATE <dbname_to_be_cloned> in the above syntax.
- Syntax from commandline : Createdb <dbname>.

- Syntax for createdb help : createdb –help

* CreateDB utility is found in bin folder

Example :

from psql prompt: *create database siddhu owner postgres;*
With createdb utility on OS command prompt :
$ *createdb siddhu -O postgres*

Verifying newly created db's OID on disk:
psql : select name,oid from pg_database;
check the OID number in data directory folder.

Drop database Psql /dropdb utility:

We can't drop the database which we are connected.

Example :

scott=# *drop database scott*;

ERROR: cannot drop the currently open database

Syntax from psql : *Drop database siddhu*;

Syntax from command line : *dropdb -u username siddhu*.

Syntax for dropdb help : *dropdb –help*
psql command to check current connection information: \conninfo

User in PostgreSQL:

- *Db users and Operating users are completely separate.*
- *Users name should be unique and should not start with pg_.*
- *Postgres super user is created by default on installation of postgresql*
- *Postgres user has all the privileges with grant option.*
- *Only super users or users with create role privilege can create a user.*
- *Database users are global across the cluster.*

Create user – Psql/ createuser utility/ Interactive

Syntax from psql : create user scott login superuser password 'welcome';

Syntax from command line : createuser <username>

Syntax for interactive user creation from command line :

Example (create user) :

```
createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) y
```

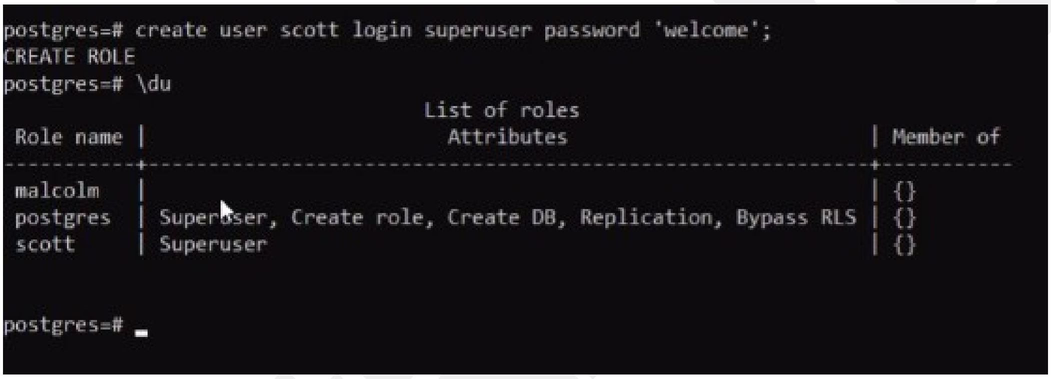
Syntax for createuser help : createuser –help

“--Interactive option will ask the required parameter while creating user.”

(Note : postgres os user does not exist in windows so we have to mention username and password with create db command. But in linux we can directly run the createuser command due to already logged in to postgres user.)

postgres# create user siddhu login superuser password ‘outcome’;

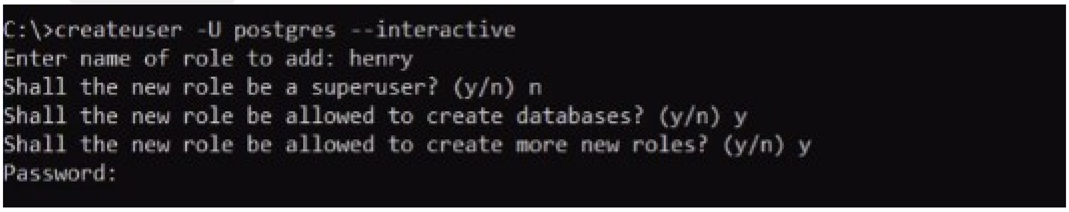
This will create the super user. We can check users details by meta command
\du.



```
postgres=# create user scott login superuser password 'welcome';
CREATE ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 malcolm   |                                                            | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 scott     | Superuser                                                  | {}


postgres=# _
```

User creation with –interactive switch in command. By this we can assign roles while user creation.

$ *Createuser –-interactive*



```
C:\>createuser -U postgres --interactive
Enter name of role to add: henry
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) y
Password:
```

Restrict user to connect particular DB :

postgres# *revoke connect on database mydb from public.*

Drop user - Psql/ dropuser utility:

Syntax from psql : drop user <username>

Syntax from command line : dropuser <username>

Dropping a user with objects or privileges will return an error.

Example :

```
postgres=# drop user test1;
ERROR:  role "test1" cannot be dropped because some objects depend on it
```

Assign the user privileges to another user before dropping the user.

Example :

```
REASSIGN OWNED BY user to postgres;
Drop role username;
```

Privileges in PostgreSQL:

- Privilege is a right to execute a particular type of SQL statement, or a right to access another user's object.
- There are two types of privileges – Cluster level and Object level
- **Cluster Level Privileges are granted by super user. It can be granted during create user or by altering an existing user.**
- **Object Level Privileges are granted by super user or the owner of the object or someone with grant privileges.**

- Privileges allow a user to perform particular actions on a database object, such as tables, view or sequence.

Grant Examples:

- Grant CONNECT to the database:

```
GRANT CONNECT ON DATABASE database_name TO username;
```

- Grant USAGE on schema:

```
GRANT USAGE ON SCHEMA schema_name TO username;
```

- Grant on all tables for DML statements: SELECT, INSERT, UPDATE, DELETE

```
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES
IN SCHEMA schema_name TO username;
```

- Grant all privileges on all tables in the schema:

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA
schema_name TO username;
```

- Grant all privileges on all sequences in the schema:

```
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA
schema_name TO username;
```

- Grant permission to create database: cluster level

```
ALTER USER username CREATEDB;
```

- Make a user superuser:

```
ALTER USER myuser WITH SUPERUSER;
```

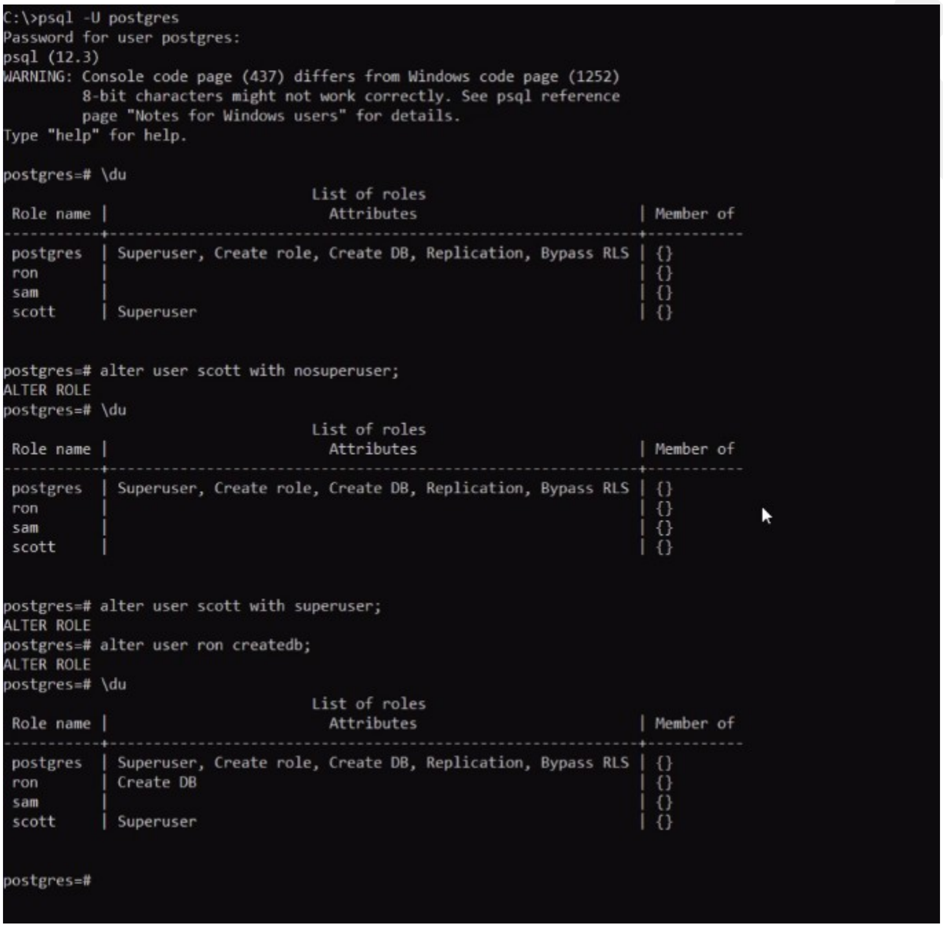
- Remove superuser status:

ALTER USER username WITH NOSUPERUSER;

- Column Level access.:

GRANT SELECT (col1), UPDATE (col1) ON mytable TO user;

Cluster Level Grant:



```
C:\>psql -U postgres
Password for user postgres:
psql (12.3)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 ron       |                                                            | {}
 sam       |                                                            | {}
 scott     | Superuser                                                  | {}


postgres=# alter user scott with nosuperuser;
ALTER ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 ron       |                                                            | {}
 sam       |                                                            | {}
 scott     |                                                            | {}


postgres=# alter user scott with superuser;
ALTER ROLE
postgres=# alter user ron createdb;
ALTER ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 ron       | Create DB                                                  | {}
 sam       |                                                            | {}
 scott     | Superuser                                                  | {}


postgres=#
```

Revoke Examples:

- Revoke Delete/update privilege on table from user

REVOKE DELETE, UPDATE ON products FROM user;

- Revoke all privilege on table from user

REVOKE ALL ON products FROM user;

- Revoke select privilege on table from all users(Public)

REVOKE SELECT ON products FROM PUBLIC;

Schema & its Benefits

- Schema is a name space that contains named objects (tables, data types, functions, and operators).
- One database can have multiple schemas.
- Schemas helps us in separation of data between different applications.
- Organize database objects into logical groups to make them more manageable.
- Applications can be put into separate schemas so that they cannot collide with the names of other objects.
- By this , One Database can be used by multiple users without interfering with each other.

Create & Drop Schema:

- Create Schema

CREATE schema <schema_name>;

- Create Schema for a user, the schema will also be named as the user

Create schema authorization <username>;

- Create Schema named John, that will be owned by brett

```
CREATE schema  IF NOT EXISTS john AUTHORIZATION brett;
```

- Drop a Schema

```
Drop schema <schema_name>;
```

NOTE:: We cannot drop schema if there are any object associate with it.

Schema Search Path :

Search path is used to determine what order PostgreSQL should search to find the objects.

Show searchpath can be used to find the current search path.

Example :

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

Default "$user" is a special option that says if there is a schema that matches the **current user** (i.e SELECT SESSION_USER;), then search within that schema.

The session_user is normally the user who initiated the current database connection; but superusers can change this setting with SET SESSION AUTHORIZATION. Both are same.

Search path can be set at session level , user level, database level and cluster level

Example :

```
  Test1=# SET search_path TO test1,public;

   Test1=# \dt

  List of relations

  Schema |  Name   | Type  |  Owner

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

    test1  | abc     | table | test1

 (1 rows)
```

Database level setting : alter database mydb set search_path="$user", public;

User Level Setting : alter role johnny set search_path = "$user", public;

Retrive the search_path Settings:

```
SELECT r.rolname, d.datname, rs.setconfig
FROM   pg_db_role_setting rs
LEFT   JOIN pg_roles      r ON r.oid = rs.setrole
LEFT   JOIN pg_database   d ON d.oid = rs.setdatabase
WHERE  r.rolname = 'myrole' OR d.datname = 'mydb';
```

PostgreSQL tablespace size

```
SELECT    pg_size_pretty ( pg_tablespace_size ('pg_default') );
```

Query to Find all active sessions and queries:

```
SELECT  pid, datname, usename, application_name, client_hostname
 ,client_port, backend_start, query_start, query, state FROM
pg_stat_activity WHERE state = 'active';
```

Queries to Find slow, long-running, and Blocked Queries

```
SELECT  pid,  user,  pg_stat_activity.query_start,  now() -
pg_stat_activity.query_start AS query_time,  query,  state,
wait_event_type,  wait_event FROM pg_stat_activity WHERE (now() -
pg_stat_activity.query_start) > interval '10 minutes';
```

Query to find blocking session

```
SELECT activity.pid, activity.usename, activity.query, blocking.pid
AS blocking_id,    blocking.query AS blocking_query FROM
pg_stat_activity AS activity JOIN pg_stat_activity AS blocking ON
blocking.pid = ANY(pg_blocking_pids(activity.pid));
```

Viewing locks with table names and queries

```
select  relname as relation_name, query, pg_locks.* from pg_locks join
pg_class on pg_locks.relation = pg_class.oid join pg_stat_activity on
pg_locks.pid = pg_stat_activity.pid;
```

Killing/cancelling a long running Postgres query:

You can find the pid using pg_stat_activity

Option #1 (graceful):

```
SELECT pg_cancel_backend(<PID>);
```

Option #2 (forceful):

```
SELECT pg_terminate_backend(<PID>);
```

Terminate all queries

If you want to terminate all running queries, the following statement can be executed:

```
SELECT pg_cancel_backend(pid) FROM pg_stat_activity WHERE state
= 'active' and pid <> pg_backend_pid();
```

Query to find high cpu usage and which query is causing it.

```
select   ((total_plan_time + total_exec_time) / 1000 / 3600) as total_hours,
((total_plan_time + total_exec_time) / 1000) as total_seconds,
((total_plan_time + total_exec_time) / calls) as avg_millis,   calls
num_calls,  query from pg_stat_statements order by 1 desc limit 10;
```

You can also use **pg_activity** utility to monitor the cpu usage.

Monitoring CPU and memory usage from Postgres

We'll create two database tables that will allow you to query CPU and memory usage from within the database connection. This way your applications can monitor the health of the servers without needing to worry about another connection or another protocol.

You can run these commands on the master database and they will propagate to all the slave databases as well.
First, load the file foreign data wrapper and create the foreign data server:

```
CREATE EXTENSION file_fdw;
CREATE SERVER fileserver FOREIGN DATA WRAPPER file_fdw;
```

Then we'll create the table that loads CPU loadavg from the /proc/loadavg file:

```
CREATE FOREIGN TABLE loadavg
(one text, five text, fifteen text, scheduled text, pid text)
SERVER fileserver
OPTIONS (filename '/proc/loadavg', format 'text', delimiter ' ');
```

Creating the table that will let you query memory info is similar:

```
CREATE FOREIGN TABLE meminfo
(stat text, value text)
SERVER fileserver
OPTIONS (filename '/proc/meminfo', format 'csv', delimiter ':');
```

Now you can run SELECT queries to see the info!

```
postgres=# SELECT * FROM loadavg;
```

you can also query

```
SELECT * FROM meminfo;
```

List all database users.

```
select * from pg_user;
```

All database and their size, with/without indexes

```
select datname, pg_size_pretty(pg_database_size(datname))
from pg_database
order by pg_database_size(datname) desc;
```

Cache hit rates (ratio should not be less than 0.99)

```
SELECT sum(heap_blks_read) as heap_read, sum(heap_blks_hit)  as
heap_hit, (sum(heap_blks_hit) - sum(heap_blks_read)) /
sum(heap_blks_hit) as ratio
FROM pg_statio_user_tables;
```

Table index usage rates (should not be less than 0.99)

```
SELECT relname, 100 * idx_scan / (seq_scan + idx_scan)
percent_of_times_index_used, n_live_tup rows_in_table
FROM pg_stat_user_tables
ORDER BY n_live_tup DESC; – division by 0
```

How many indexes are in cache

```
SELECT sum(idx_blks_read) as idx_read, sum(idx_blks_hit)  as idx_hit,
(sum(idx_blks_hit) - sum(idx_blks_read)) / sum(idx_blks_hit) as ratio
FROM pg_statio_user_indexes;
```

Show unused indexes:

```
SELECT relname AS table_name, indexrelname AS index_name,
idx_scan, idx_tup_read, idx_tup_fetch,
pg_size_pretty(pg_relation_size(indexrelname::regclass))
FROM pg_stat_all_indexes
WHERE schemaname = 'public'
AND idx_scan = 0
AND idx_tup_read = 0
AND idx_tup_fetch = 0
ORDER BY pg_relation_size(indexrelname::regclass) DESC;
```

Find cardinality of index:

```
SELECT relname, relkind, reltuples as cardinality, relpages
FROM pg_class
WHERE relname LIKE 'tableprefix%';
```

or

Find cardinality of index

```
SELECT schema_name,
       object_name,
       object_type,
       cardinality,
       pages
FROM (
       SELECT pg_catalog.pg_namespace.nspname AS schema_name,
              relname                       as object_name,
              relkind                      as object_type,
              reltuples                    as cardinality,
              relpages                      as pages
       FROM pg_catalog.pg_class
              JOIN pg_catalog.pg_namespace ON relnamespace =
pg_catalog.pg_namespace.oid
     ) t
WHERE schema_name NOT LIKE 'pg_%'
  and schema_name <> 'information_schema'
  --and schema_name = '$schema_name'
  --and object_name = '$object_name'
ORDER BY pages DESC, schema_name, object_name;
```

Cardinality is an indicator that refers to the uniqueness of all values in a column. Low cardinality means a lot of duplicate values in that column. For example, a column that stores the gender values has low cardinality. In contrast, high cardinality means that there are many distinct values.

Show Table Bloats

```
with foo as (
  SELECT
    schemaname, tablename, hdr, ma, bs,
    SUM((1-null_frac)*avg_width) AS datawidth,
    MAX(null_frac) AS maxfracsum,
    hdr+(
      SELECT 1+COUNT(*)/8
      FROM pg_stats s2
      WHERE null_frac<>0 AND s2.schemaname = s.schemaname AND
s2.tablename = s.tablename
```

```
    ) AS nullhdr
  FROM pg_stats s, (
    SELECT
      (SELECT current_setting('block_size')::NUMERIC) AS bs,
      CASE WHEN SUBSTRING(v,12,3) IN ('8.0','8.1','8.2') THEN 27
ELSE 23 END AS hdr,
      CASE WHEN v ~ 'mingw32' THEN 8 ELSE 4 END AS ma
    FROM (SELECT version() AS v) AS foo
  ) AS constants
  GROUP BY 1,2,3,4,5
), rs as (
  SELECT
    ma,bs,schemaname,tablename,
    (datawidth+(hdr+ma-(CASE WHEN hdr%ma=0 THEN ma ELSE hdr
%ma END)))::NUMERIC AS datahdr,
    (maxfracsum*(nullhdr+ma-(CASE WHEN nullhdr%ma=0 THEN ma
ELSE nullhdr%ma END))) AS nullhdr2
  FROM foo
), sml as (
  SELECT
    schemaname, tablename, cc.reltuples, cc.relpages, bs,
    CEIL((cc.reltuples*((datahdr+ma-
      (CASE WHEN datahdr%ma=0 THEN ma ELSE datahdr%ma END))
+nullhdr2+4))/(bs-20::FLOAT)) AS otta,
    COALESCE(c2.relname,'?') AS iname, COALESCE(c2.reltuples,0) AS
ituples, COALESCE(c2.relpages,0) AS ipages,
    COALESCE(CEIL((c2.reltuples*(datahdr-12))/(bs-20::FLOAT)),0) AS
iotta -- very rough approximation, assumes all cols
  FROM rs
  JOIN pg_class cc ON cc.relname = rs.tablename
  JOIN pg_namespace nn ON cc.relnamespace = nn.oid AND nn.nspname
= rs.schemaname AND nn.nspname <> 'information_schema'
  LEFT JOIN pg_index i ON indrelid = cc.oid
  LEFT JOIN pg_class c2 ON c2.oid = i.indexrelid
)
```

```
SELECT
  current_database(), schemaname, tablename, /*reltuples::bigint,
relpages::bigint, otta,*/
  ROUND((CASE WHEN otta=0 THEN 0.0 ELSE
sml.relpages::FLOAT/otta END)::NUMERIC,1) AS tbloat,
  CASE WHEN relpages < otta THEN 0 ELSE bs*(sml.relpages-
otta)::BIGINT END AS wastedbytes,
  iname, /*ituples::bigint, ipages::bigint, iotta,*/
  ROUND((CASE WHEN iotta=0 OR ipages=0 THEN 0.0 ELSE
ipages::FLOAT/iotta END)::NUMERIC,1) AS ibloat,
  CASE WHEN ipages < iotta THEN 0 ELSE bs*(ipages-iotta) END AS
wastedibytes
FROM sml

ORDER BY wastedbytes DESC
```

PSQL Commands:

Connect to Psql

Psql is a terminal-based front-end to PostgreSQL.

It enables the users to query postgreSQL interactively and see the query results.

Connect to Specific Database with user and password

Syntax: psql -d database -U user –W (-d =Database,-U = User, -W = Password)

Connect to Database on a different host/machine.

Syntax : psql -h host -d database -U user –W

Connect using SSL Mode

Syntax : psql -U user -h host "dbname=db sslmode=require"

Psql Commands:

Switch connection to a new database

```
   postgres=# \c test1
  You are now connected to database "test1" as user "postgres".
```

List available databases

```
   postgres=# \l
```

List available tables

```
   postgres=# \ dt
```

Describe a table

```
    postgres=# \d table_name
```

List available schema (+ to get more info)

```
    postgres=# \dn
```

List available functions(+ to get more info)

```
   postgres=# \df
```

\

List available views(+ to get more info)

```
   postgres=# \dv
```

List users and their roles(+ to get more info)

```
    postgres=# \du
```

List available sequence(+ to get more info)

```
   postgres=# \ds
```

Execute the previous command

```
postgres=# \g
```

Command history

```
postgres=# \s
```

Save Command History to file:

```
postgres=# \s filename
```

Get help on psql commands

```
postgres=# \?
```

Turn on\off query execution time

```
postgres=# \timing
```

Edit statements in editor

```
postgres=# \e
```

Edit Functions in editor

```
postgres=# \ef
```

set output from non-aligned to aligned column output.

```
postgres=# \a
```

Formats output to HTML format.

```
postgres=# \H
```

Connection Information

```
postgres=# \conninfo
```

Quit psql

```
postgres=# \q
```

Run sql statements from a file.

```
psql -d test1 -U test1 -f test1.sql ( command line)
```

Send the output to a file.

```
postgres=# \o <filename>
```

Save query buffer to filename.

```
postgres=# \w filename
```

Turn off auto commit on session level

```
\set AUTOCOMMIT off
```

PostgreSQL SYSTEM_CATALOGS

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables.

Name	Description
pg_database	Stores general database info
pg_stat_database	Contains stats information of database
pg_tablespace	Contains Tablespace information
pg_operator	Contains all operator information
pg_available_extensions	List all available extensions

pg_shadow	List of all database users. pg_user is a publicly readable view on pg_shadow that **blanks out the password field**.
pg_stats	Planner stats
pg_timezone_names	Time Zone names
pg_locks	Currently held locks
pg_tables	All tables in the database
pg_settings	Parameter Settings
pg_user_mappings	All user mappings
pg_indexes	All indexes in the database
pg_views	All views in the database.

- **Find Current Schema**

  ```
  postgres=# select current_Schema();
  ```

- **Current User**

  ```
  postgres=# select current_user;
  ```

- **Current Database**

  ```
  postgres=# select current_database();
  ```

- **Current setting of any parameter in PostgreSQL**

  ```
  postgres=# select current_setting('max_parallel_workers');
  ```

- **Current User process id**

  ```
  postgres=# select pg_backend_pid();
  ```

- **Find Postmaster start time**

  ```
  postgres=# select pg_postmaster_start_time();
  ```

- **PostgreSQL Version**

```
    postgres=# select version ();
```

- **Backup is running or not**

```
    postgres=# select pg_is_in_backup();
```

- **Date & Time in PostgreSQL with time zone:**

```
    postgres=# select now () as current;
```

- **Current Date and Time without Timezone**

```
     postgres=# SELECT NOW ()::timestamp;
```

- **Add 1 hour to existing date and time**

```
     postgres=# SELECT (NOW () + interval '1 hour') AS
an_hour_later;
```

- **To Find next day date and time**

```
     postgres=# SELECT (NOW () + interval '1 day') AS
this_time_tomorrow;
```

- **To deduct 2 hours and 30 minutes from current time**

```
     postgres=# SELECT now() - interval '2 hours 30 minutes' AS
two_hour_30_min_go;
```

TABLE INHERITANCE & PARTITIONING

Module Objectives:

- Table Inheritance
- Table Partitioning
- Copy Table

Table Inheritance:

- **Table inheritance allows child table to inherit all the columns of the parent master table.**
- A child table can have extra fields of its own in addition to the inherited columns.
- Query references all rows of that master table plus all of its children tables.
- "**Only**" keyword can be used to indicate that the query should apply only to a particular table and not any tables.
- Any update or delete on parents table without only affects the records in child table.

Example:

<u>Create Table:</u>

```
create table orders(orderno serial, flightname varchar(100),boarding
varchar(100),status varchar(100),source varchar(100));

create table online_booking (price int) inherits(orders);
create table agent_booking (commision int) inherits(orders);
```

<u>Insert Records:</u>

```
insert into orders(flightname,boarding,status,source)
values('aircanada','xyz','ontime','employees');

insert into online_booking(flightname,boarding,status,source,price)
values('nippon','chn','ontime','website',5000);
```

```
insert into online_booking(flightname,boarding,status,source,price)
values('luftansa','chn','ontime','app',3000);

 insert into agent_booking(flightname,boarding,status,source,commision)
values('etihad','aud','ontime','agent001',1000);

insert into agent_booking(flightname,boarding,status,source,commision)
values('emirates','dxb','ontime','agent007',1300);
```

Select Parent Table:

```
nano=# \set AUTOCOMMIT off

nano=# select * from orders;

 orderno | flightname | boarding | status |  source
---------+------------+----------+--------+-----------
       1 | aircanada  | xyz      | ontime | employees
       2 | nippon     | chn      | ontime | website
       3 | luftansa   | chn      | ontime | app
       5 | etihad     | aud      | ontime | agent001
       6 | emirates   | dxb      | ontime | agent007
(5 rows)
```

Update Parent:

```
nano=# update orders set status='Cancelled';
UPDATE 5
nano=# select * from orders;
 orderno | flightname | boarding |  status   |  source
---------+------------+----------+-----------+-----------
       1 | aircanada  | xyz      | Cancelled | employees
       2 | nippon     | chn      | Cancelled | website
       3 | luftansa   | chn      | Cancelled | app
       5 | etihad     | aud      | Cancelled | agent001
       6 | emirates   | dxb      | Cancelled | agent007
(5 rows)

nano=# rollback;
ROLLBACK
nano=# select * from orders;
 orderno | flightname | boarding | status |  source
---------+------------+----------+--------+-----------
       1 | aircanada  | xyz      | ontime | employees
       2 | nippon     | chn      | ontime | website
       3 | luftansa   | chn      | ontime | app
       5 | etihad     | aud      | ontime | agent001
       6 | emirates   | dxb      | ontime | agent007
(5 rows)

nano=# update only orders set status='Cancelled';
UPDATE 1
nano=# select * from orders;
 orderno | flightname | boarding |  status   |  source
---------+------------+----------+-----------+-----------
       1 | aircanada  | xyz      | Cancelled | employees
       2 | nippon     | chn      | ontime    | website
       3 | luftansa   | chn      | ontime    | app
       5 | etihad     | aud      | ontime    | agent001
       6 | emirates   | dxb      | ontime    | agent007
(5 rows)
```

Delete:

```
nano=# delete from orders;
DELETE 5
nano=# select * from orders;
 orderno | flightname | boarding | status | source
---------+------------+----------+--------+--------
(0 rows)
```

```
nano=# rollback;
WARNING:  there is no transaction in progress
ROLLBACK
nano=# select * from orders;
 orderno | flightname | boarding | status | source
---------+------------+----------+--------+--------
(0 rows)
```

Drop Table:

```
nano=# drop table orders;
ERROR:  cannot drop table orders because other objects depend on it
DETAIL:  table online_booking depends on table orders
table agent_booking depends on table orders
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
nano=# drop table orders cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table online_booking
drop cascades to table agent_booking
DROP TABLE
```

Table Partitioning :

- **Table Partitioning means splitting a table into smaller pieces.**
- Table Partitioning holds many performance benefits for tables that hold large amount of data.
- PostgreSQL allows table partitioning via table inheritance.
- Each Partition is created as a child table of a single parent table.
- PostgreSQL implements range and list partitioning methods.

Example:

Create Table:

```
create table bookings(flightno varchar(200),flightname
varchar(200),booking_date timestamp);

create table jan_bookings(check(booking_date >= date '2020-01-01' and
booking_date <= '2020-01-31')) inherits(bookings);

create table feb_bookings(check(booking_date >= date '2020-02-01' and
booking_date <= '2020-02-29')) inherits(bookings);

nano=# \d+ bookings
                                          Table "public.bookings"
   Column    |            Type             | Collation | Nullable | Default | Storage  |
Stats target | Description
--------------+-----------------------------+-----------+----------+---------
+----------+--------------+-------------
 flightno     | character varying(200)      |           |          |         | extended |
|
 flightname   | character varying(200)      |           |          |         | extended |
|
 booking_date | timestamp without time zone |           |          |         | plain    |
|
Child tables: feb_booking,
              jan_booking
Access method: heap
```

Create Index:

```
nano=# create index booking_jan_idx on jan_booking using
btree(booking_date);
CREATE INDEX

nano=# create index booking_feb_idx on feb_booking using
btree(booking_date);
CREATE INDEX
```

Create Function:

```
create or replace function on_insert() returns trigger as $$
begin
  if(new.booking_date >= date '2020-01-01' and new.booking_date <=date
'2020-01-31') then
     insert into jan_booking values(new.*);
    elseif (new.booking_date >= date '2020-02-01' and new.booking_date
<=date '2020-02-29') then
     insert into feb_booking values(new.*);
    else
     raise exception 'Enter valid booking date';
    end if;
     return null;
end;
$$ LANGUAGE plpgsql;
```

<u>Create Trigger:</u>

```
create trigger booking_entry before insert on bookings for each row
execute procedure on_insert();
```

```
CREATE TRIGGER  booking_entry
  BEFORE INSERT
  ON  bookings
  FOR EACH ROW
  EXECUTE PROCEDURE on_insert();
```

<u>Insert Records:</u>

```
nano=# insert into bookings values('dxb102','emirates','2020-02-09');
INSERT 0 0
nano=# insert into bookings values('dxb103','emirates','2020-02-15');
INSERT 0 0
nano=# insert into bookings values('auh345','etihad','2020-01-10');
INSERT 0 0
nano=# select * from bookings;
```

```
 flightno | flightname |    booking_date
----------+------------+---------------------
 dxb101   | etihad     | 2020-01-25 00:00:00
 auh345   | etihad     | 2020-01-10 00:00:00
 dxb102   | emirates   | 2020-02-09 00:00:00
 dxb103   | emirates   | 2020-02-15 00:00:00
(4 rows)

nano=# select * from only bookings;
 flightno | flightname | booking_date
----------+------------+--------------
(0 rows)
```

```
nano=# select * from jan_booking;
 flightno | flightname |    booking_date
----------+------------+---------------------
 dxb101   | etihad     | 2020-01-25 00:00:00
 auh345   | etihad     | 2020-01-10 00:00:00
(2 rows)

nano=# select * from feb_booking;
flightno | flightname |    booking_date
----------+------------+---------------------
 dxb102   | emirates   | 2020-02-09 00:00:00
 dxb103   | emirates   | 2020-02-15 00:00:00
(2 rows)
```

<u>Validate Function with wrong entry:</u>

```
nano=# insert into bookings values('auh234','etihad','2020-03-12');
ERROR:  Enter valid booking date
        CONTEXT:  PL/pgSQL function on_insert() line 8 at RAISE
```

COPY TABLE With or Without Data:

Copy Table is used to copy the structure of a table along with data.

Unlike Inheritance table , copy table does not have any relationship with the base table.

Syntax with data:

CREATE TABLE new_table AS TABLE existing_table;

Syntax without data:

CREATE TABLE new_table AS TABLE existing_table ***WITH NO DATA;***

Example:

Create Table:

```
nano=# create table train_bookings(trainno serial,trainname
varchar(200),destination varchar(100));
```

Insert Records:

```
nano=# insert into train_bookings(trainname,destination)
values('express','toronto');
INSERT 0 1
nano=# insert into train_bookings(trainname,destination)
values('semiexpress','montreal');
INSERT 0 1
nano=# insert into train_bookings(trainname,destination)
values('goods','calgary');
INSERT 0 1
nano=# select * from train_bookings;
 trainno |  trainname  | destination
---------+-------------+-------------
       1 | express     | toronto
       2 | semiexpress | montreal
       3 | goods       | calgary
(3 rows)
```

Create Table AS:

```
nano=# create table train_dest as table train_bookings;
SELECT 3
nano=# select * from train_dest;
 trainno |  trainname  | destination
---------+-------------+-------------
       1 | express     | toronto
       2 | semiexpress | montreal
       3 | goods       | calgary
(3 rows)


nano=# \d train_bookings
                                        Table "public.train_bookings"
   Column    |          Type          | Collation | Nullable |                     Default
-------------+------------------------+-----------+----------
+---------------------------------------------
 trainno     | integer                |           | not null |
nextval('train_bookings_trainno_seq'::regclass)
 trainname   | character varying(200) |           |          |
 destination | character varying(100) |           |          |


nano=# \d train_dest
                       Table "public.train_dest"
   Column    |          Type          | Collation | Nullable | Default
-------------+------------------------+-----------+----------+---------
 trainno     | integer                |           |          |
 trainname   | character varying(200) |           |          |
 destination | character varying(100) |           |          |
```

Insert into Destination:

```
nano=# insert into train_dest (trainname,destination)
values('passenger','yukon');
INSERT 0 1
nano=# select * from train_bookings;
 trainno |  trainname  | destination
---------+-------------+-------------
       1 | express     | toronto
       2 | semiexpress | montreal
       3 | goods       | calgary
(3 rows)
nano=# select * from train_dest;
 trainno |  trainname  | destination
---------+-------------+-------------
       1 | express     | toronto
       2 | semiexpress | montreal
       3 | goods       | calgary
         | passenger   | yukon
(4 rows)
```

Drop Table:

```
nano=# drop table train_dest;
DROP TABLE
nano=# create table train_dest as table train_bookings with no data;
CREATE TABLE AS
nano=# select * from train_dest;
 trainno | trainname | destination
---------+-----------+-------------
(0 rows)
```

TABLE SPACES IN PostgreSQL

Module Objective:

- Tablespace & its advantages

- PostgreSQL default tablespaces
- Create tablespaces
- Move table from one tablespace to another
- Drop tablespaces
- Temporary tablespaces

Tablespace & its advantages

- PostgreSQL stores data logically in tablespaces and physically in datafiles.

  “Tablespace does not holds any data. Data is still located in the disk. Tablespace just map a logical name for the physical location of the data on the disk. It just a pointer to the location of the data. Tablespace can be used to segregate the data i.e. data in drive a and indexes in drive d. ”
- PostgreSQL uses a tablespace to map a logical name to a physical location on disk.
- Tablespace allows the user to control the disk layout of PostgreSQL.
- Statistics of database objects usage to optimize the performance of databases.
- Allocate data storage across devices to improve performance .
- WAL files object on fast media and archive data on slow media.
- In Iinux , postgres user must have the permission to read write on the folder.

DEFAULT TABLESPACE:

- Default comes with two out of the box tablespaces namely pg_default and pg_global
- pg_default tablespace stores all user data.
- pg_global tablespace stores all global data.
- pg_default tablespace is the default tablespace of the template1 and template0 databases.
- All newly created database uses pg_default tablespace, unless overridden by a TABLESPACE clause while CREATING DATABASE.
- Location of Default Tablespaces is data directory.

DROP TABLESPACE:

Dropping a tablespace all the reference from the system automatically.

We cannot drop a tablespace which is not empty.

Find objects associate with the tablespace

Syntax : select * from pg_tables where tablespace = 'hrd';

Drop tablespace

Syntax : drop tablespace hrd;

Query pg system catalog view to check the tablespace is dropped.

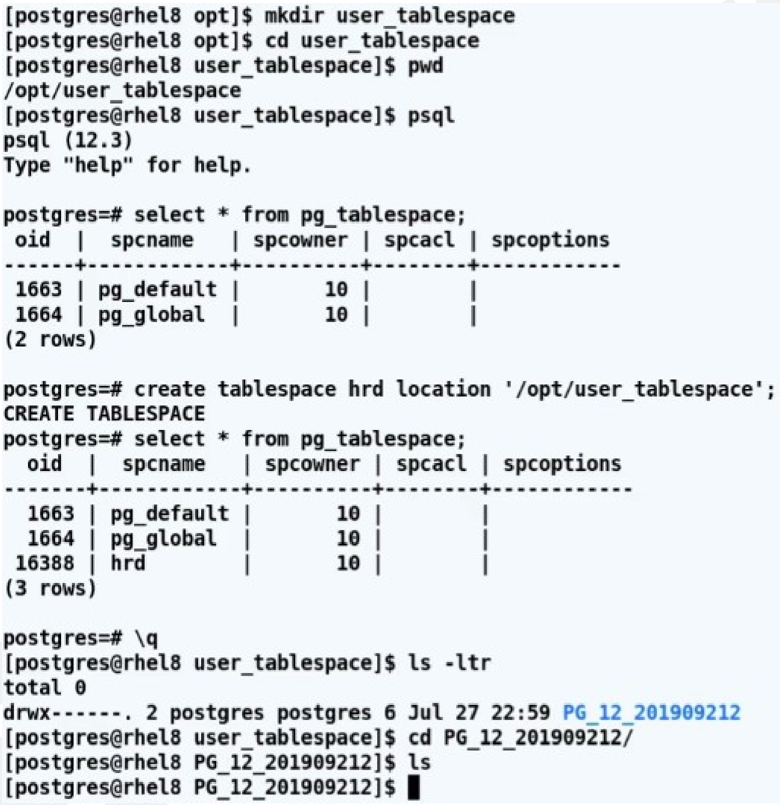
Syntax : select * from pg_tablespace;

Examples:

- Syntax for creating tablespace: (ensure the location exist)
  create tablespace hrd location '/opt/app/hrd/';

- Syntax for creating a table on a newly created tablespace
  create table test1(studid int,stuname varchar(50)) tablespace hrd;
- Query to find which tablespace the table belong to

  select * from pg_tables where tablespace='hrd';
  or
  select * from pg_tables where tablename=‘test1';



```
[postgres@rhel8 opt]$ mkdir user_tablespace
[postgres@rhel8 opt]$ cd user_tablespace
[postgres@rhel8 user_tablespace]$ pwd
/opt/user_tablespace
[postgres@rhel8 user_tablespace]$ psql
psql (12.3)
Type "help" for help.

postgres=# select * from pg_tablespace;
 oid  |  spcname   | spcowner | spcacl | spcoptions
------+------------+----------+--------+------------
 1663 | pg_default |       10 |        |
 1664 | pg_global  |       10 |        |
(2 rows)

postgres=# create tablespace hrd location '/opt/user_tablespace';
CREATE TABLESPACE
postgres=# select * from pg_tablespace;
  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16388 | hrd        |       10 |        |
(3 rows)

postgres=# \q
[postgres@rhel8 user_tablespace]$ ls -ltr
total 0
drwx------. 2 postgres postgres 6 Jul 27 22:59 PG_12_201909212
[postgres@rhel8 user_tablespace]$ cd PG_12_201909212/
[postgres@rhel8 PG_12_201909212]$ ls
[postgres@rhel8 PG_12_201909212]$ 
```

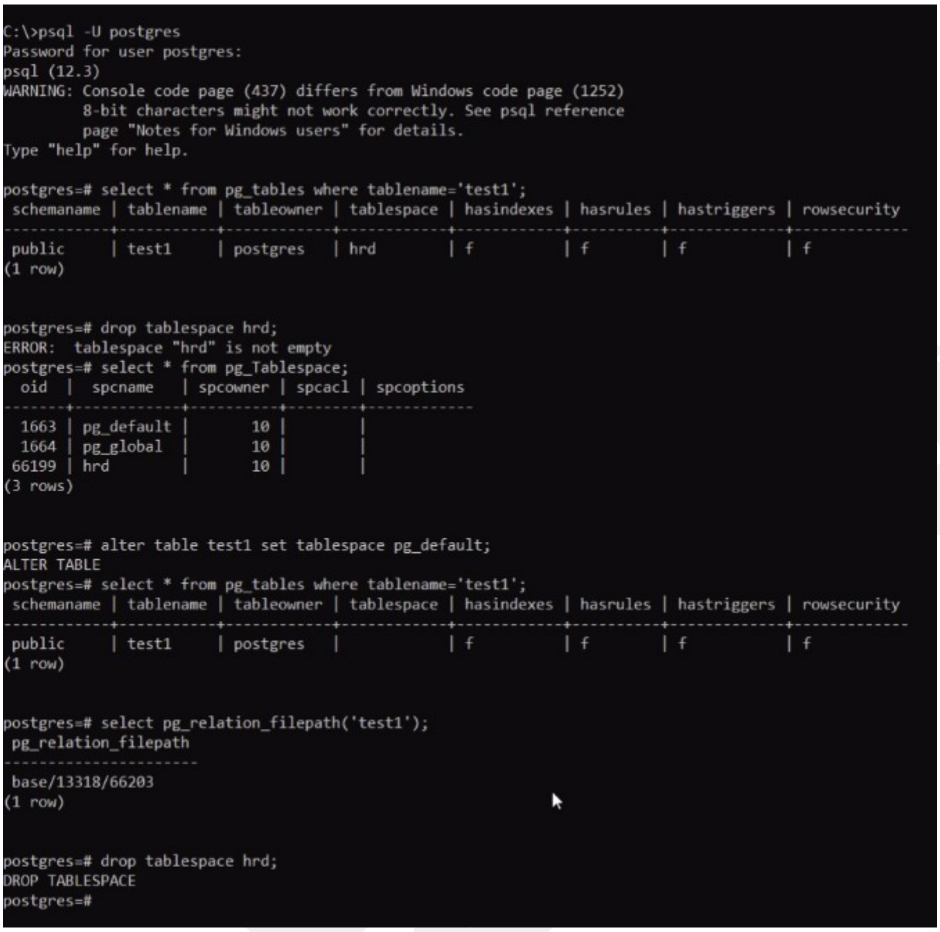
MOVE TABLE BETWEEN TABLE SPACES:

- Move tables from one tablespace to another
  Syntax: alter table test1 set tablespace pg_default

- Check whether the table is moved successfully to another tablespace
  Syntax: select * from pg_tables where tablename=‘test1’

- Find physical location of the table
  Syntax: select pg_relation_filepath(‘test1');

- Find physical location of the tablespace
  Syntax: postgres#\dt



```
C:\>psql -U postgres
Password for user postgres:
psql (12.3)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# select * from pg_tables where tablename='test1';
 schemaname | tablename | tableowner | tablespace | hasindexes | hasrules | hastriggers | rowsecurity
------------+-----------+------------+------------+------------+----------+-------------+-------------
 public     | test1     | postgres   | hrd        | f          | f        | f           | f
(1 row)


postgres=# drop tablespace hrd;
ERROR:  tablespace "hrd" is not empty
postgres=# select * from pg_Tablespace;
  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 66199 | hrd        |       10 |        |
(3 rows)


postgres=# alter table test1 set tablespace pg_default;
ALTER TABLE
postgres=# select * from pg_tables where tablename='test1';
 schemaname | tablename | tableowner | tablespace | hasindexes | hasrules | hastriggers | rowsecurity
------------+-----------+------------+------------+------------+----------+-------------+-------------
 public     | test1     | postgres   |            | f          | f        | f           | f
(1 row)


postgres=# select pg_relation_filepath('test1');
 pg_relation_filepath
----------------------
 base/13318/66203
(1 row)


postgres=# drop tablespace hrd;
DROP TABLESPACE
postgres=#
```

Temporary tablespace:

- **Temporary tables and indexes are created by PostgreSQL when it needs to hold large datasets temporarily for completing a query. EX: Sorting**
- Temporary tablespace does not store any data and their no persistent file left when we shutdown database.

  How to create temporary tablespace

- **Syntax** : *CREATE TABLESPACE temp01 OWNER ownername LOCATION '\opt\app\hrd\'*

- Set temp_tablespaces=temp01 in postgresql.conf and reloaded configuration.
- PG will automatically create a subfolder in the above location when a temp table is created.
- When we shutdown the database the temp files will be delete automatically.

BACKUPS & RESTORES

- Backup is a copy of data taken from the database and can be used to reconstruct in case of a failure.
- Backups can be divided into **Logical backups** and **Physical backups**.
- **Logical Backups** are simple and the **textual representation of the data** in the databases.
- These text statements can be used to recreate postgres cluster, database or table.

- **Physical backups** are backups of the physical files used in storing and recovering of database, such as **datafiles**, **wal** files and **archive files**.
- **Physical backups** are further divided as **online** backup and **offline** backup.

Logical Backup:

- Logical Backups are simple and the textual representation of the data in the databases.
- It supports various output forms like plain text(default),tar and custom binary format.
- Sql dumps creates a consistent copy of database as of the time of execution.
- **Small database are perfect candidates for logical backups.**
- **pg_dump (For one db)** and **pg_dumpall (All Dbs in cluster)** utilities are used to perform logical dumps.
- **pg_dump --help** displays the options which can be used to customize of dumps.

Pg_dump :

Backup single database from postgres instance

Syntax: *pg_dump test1 > /var/lib/pgsql/12/backups/test1backup*

Pg_dumpall -- extract a PostgreSQL database **cluster into a script file**

Syntax: *pg_dumpall -U postgres > /var/lib/pgsql/12/backups/clusterall.sql*

We can use any standard editor to view the extracted file (Vi or notepad)

pg_dumpall --help displays the options which can be used to customize of dumps.

Compressed & Split Dumps:

Dumps grows exponentially when dealing with large databases

We can use any standard compression utility to compress the dump like gz (**Linux only**)

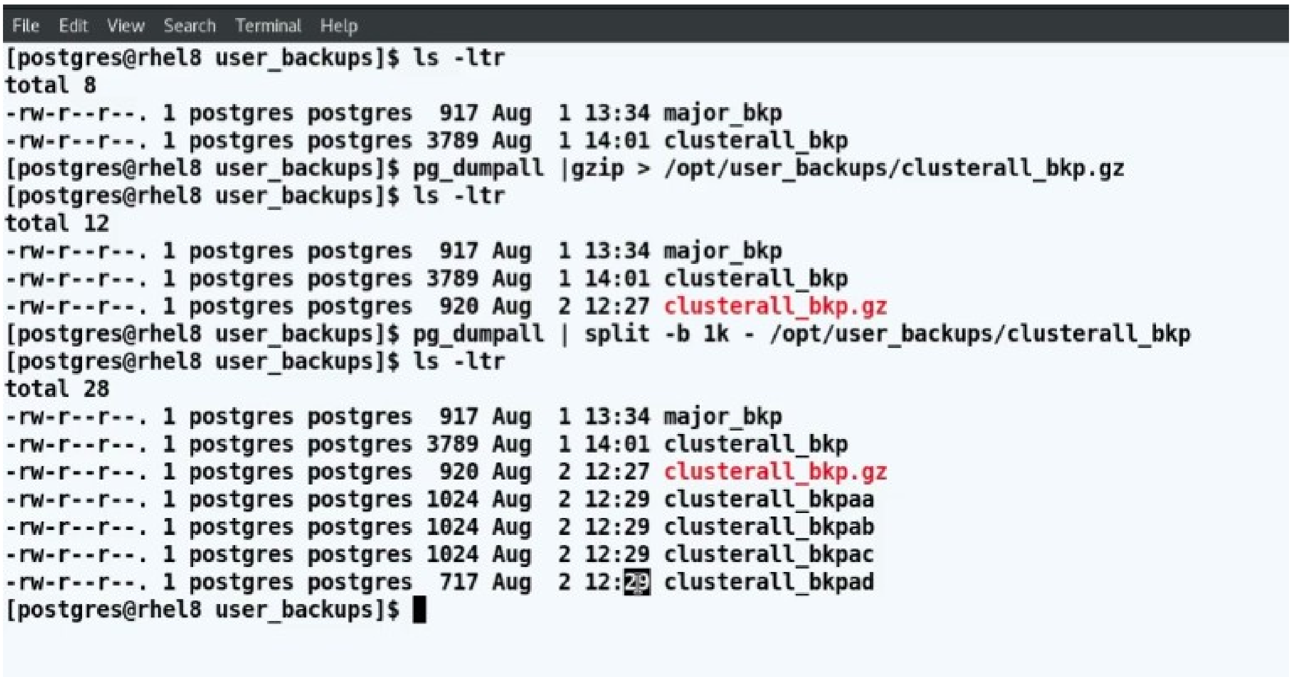
Syntax : *pg_dump test1 | gzip >/var/lib/pgsql/11/backups/test1backup.gz*

We can split the dumps into smaller chunks of desirable size for easy maintenance.

Syntax :
pg_dump test1 | split -b 1k - /var/lib/pgsql/11/backups/test1backup

Compressed & split backup

```
$ pg_dump -h localhost -U postgres -W -d mydb | gzip | split -b 100m –
mydb.sql.gz
```



```
[postgres@rhel8 user_backups]$ ls -ltr
total 8
-rw-r--r--. 1 postgres postgres  917 Aug  1 13:34 major_bkp
-rw-r--r--. 1 postgres postgres 3789 Aug  1 14:01 clusterall_bkp
[postgres@rhel8 user_backups]$ pg_dumpall |gzip > /opt/user_backups/clusterall_bkp.gz
[postgres@rhel8 user_backups]$ ls -ltr
total 12
-rw-r--r--. 1 postgres postgres  917 Aug  1 13:34 major_bkp
-rw-r--r--. 1 postgres postgres 3789 Aug  1 14:01 clusterall_bkp
-rw-r--r--. 1 postgres postgres  920 Aug  2 12:27 clusterall_bkp.gz
[postgres@rhel8 user_backups]$ pg_dumpall | split -b 1k - /opt/user_backups/clusterall_bkp
[postgres@rhel8 user_backups]$ ls -ltr
total 28
-rw-r--r--. 1 postgres postgres  917 Aug  1 13:34 major_bkp
-rw-r--r--. 1 postgres postgres 3789 Aug  1 14:01 clusterall_bkp
-rw-r--r--. 1 postgres postgres  920 Aug  2 12:27 clusterall_bkp.gz
-rw-r--r--. 1 postgres postgres 1024 Aug  2 12:29 clusterall_bkpaa
-rw-r--r--. 1 postgres postgres 1024 Aug  2 12:29 clusterall_bkpab
-rw-r--r--. 1 postgres postgres 1024 Aug  2 12:29 clusterall_bkpac
-rw-r--r--. 1 postgres postgres  717 Aug  2 12:29 clusterall_bkpad
[postgres@rhel8 user_backups]$ 
```

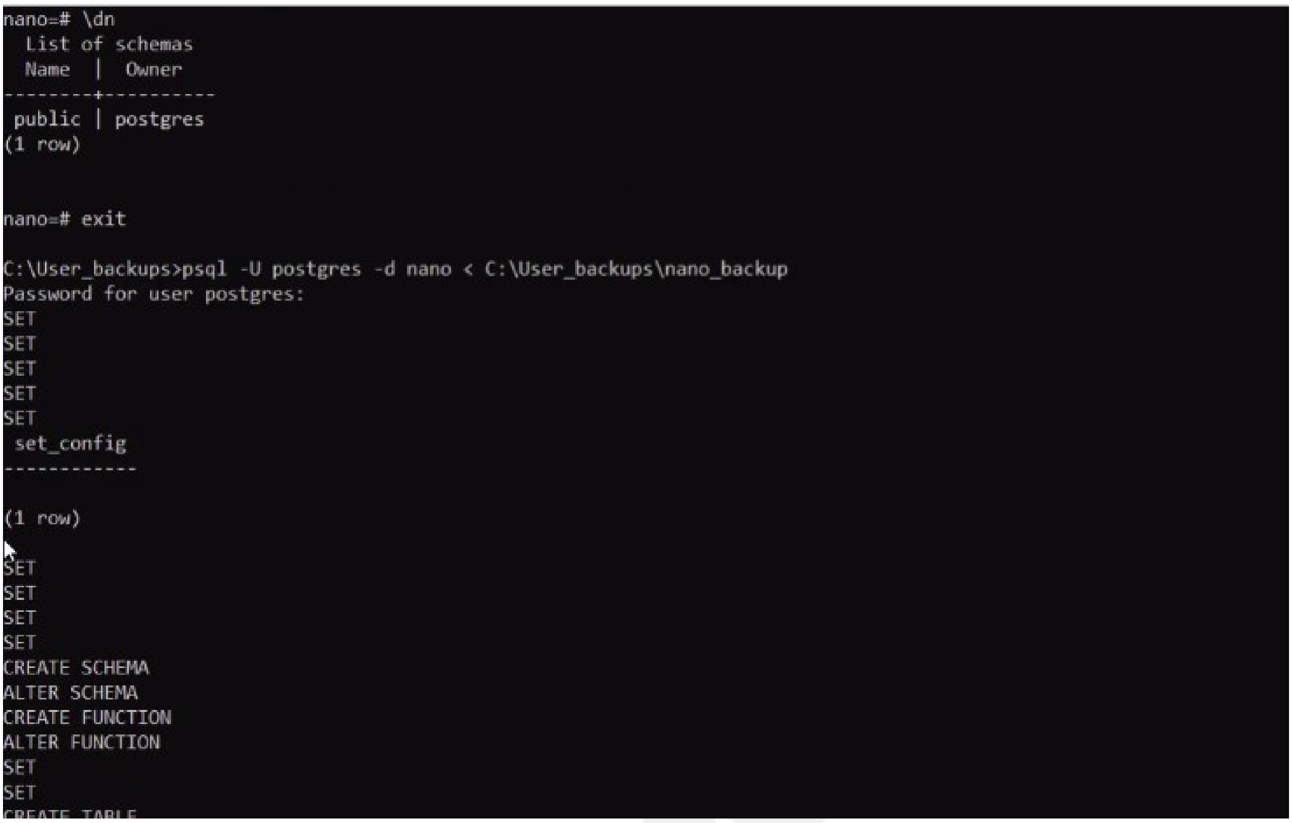
Restore database using psql interface:

- Data restore is the process of copying backup data from secondary storage and restoring it to its original location or a new location.

- Restore is performed to return data that has been lost, stolen or damaged to its original condition or to move data to a new location.

- Restoring database using pg_dump backup
  Syntax : *psql -U test1 -d test1 </var/lib/pgsql/11/backups/test1backup*

***NOTE** : We have to create empty database with same name before restore.



```
nano=# \dn
  List of schemas
  Name  |  Owner
--------+----------
 public | postgres
(1 row)


nano=# exit

C:\User_backups>psql -U postgres -d nano < C:\User_backups\nano_backup
Password for user postgres:
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
SET
CREATE SCHEMA
ALTER SCHEMA
CREATE FUNCTION
ALTER FUNCTION
SET
SET
CREATE TABLE
```

Pg_Restore

Pg_restore is a utility for restoring a PostgreSQL database from an archive created by pg_dump in one of the non-plain-text formats.

Pg_restore is a useful utility use to **restore a database or a single table.**

Syntax to take Custom Format Archive file for pg_restore:

```
# pg_dump -Fc mydb > db.dump
```

Syntax for pg_restore:

```
#  pg_restore –d test1 db.dump
```

We can restore a single table from a full pg_dump file without restoring the entire database.

This scenario is really helpful when we lose a particular table accidentally or due to user mistake.

Syntax:

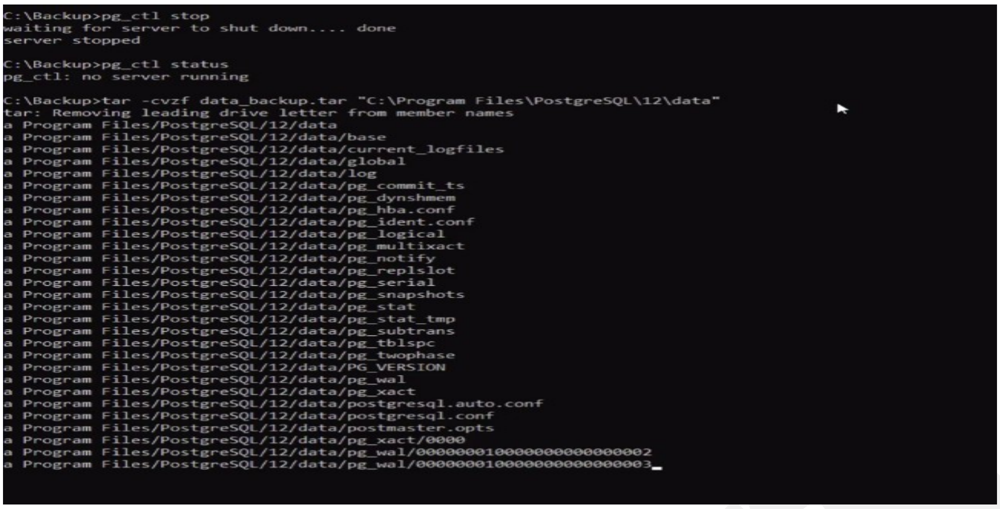
```
$ pg_restore -t <tablename> -d <DBNAME> pathtodump.dump
```

List content / objects in dump

```
$ pg_restore -l <dump file name> -- non plain text
```

File System backup – Offline mode :

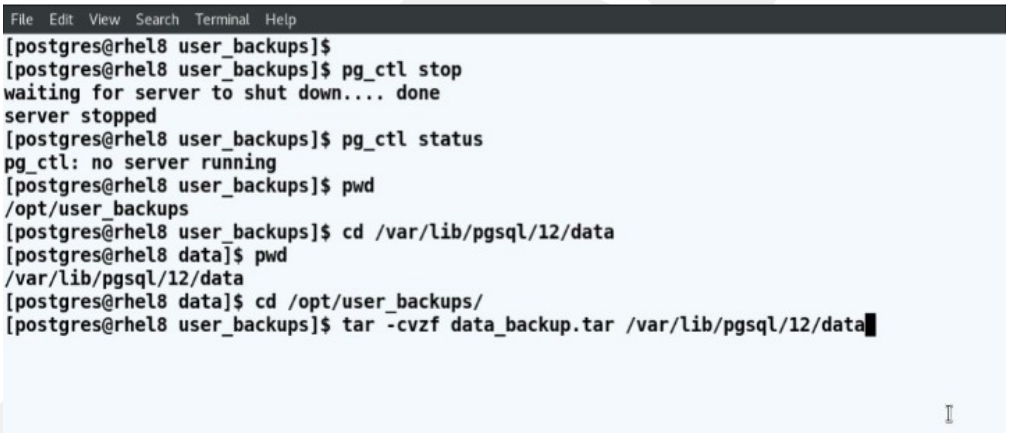
- The database server must be shut down in order to get a usable backup.
- The database server must be shutdown before restoring the data.
- Partial restore or Single table restore not possible.
- This approach is suitable only for complete backup or complete restoration of the entire database cluster.
- “Consistent snapshot" of the data directory is considered a better approach than file system level backup.
- Syntax : tar –cvzf backup.tar /usr/local/pgsql/data



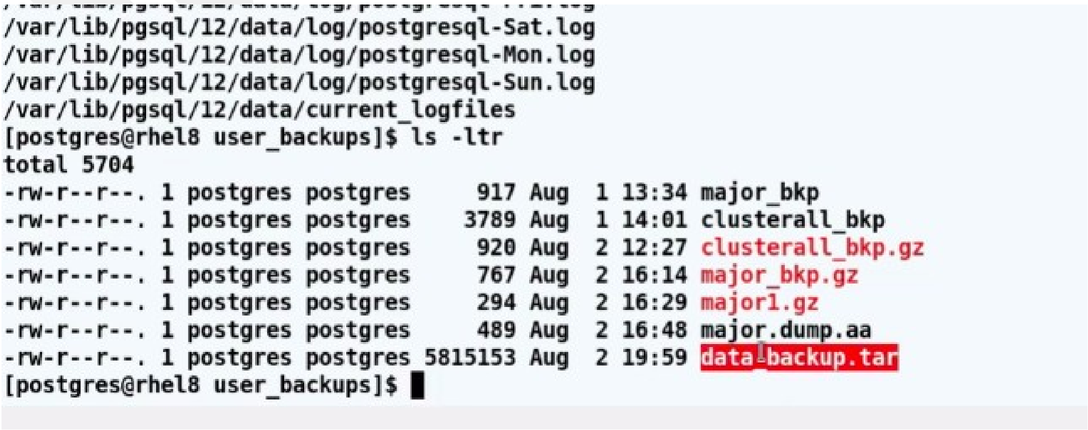
```
C:\Backup>pg_ctl stop
waiting for server to shut down.... done
server stopped

C:\Backup>pg_ctl status
pg_ctl: no server running

C:\Backup>tar -cvzf data_backup.tar "C:\Program Files\PostgreSQL\12\data"
tar: Removing leading drive letter from member names
a Program Files/PostgreSQL/12/data
a Program Files/PostgreSQL/12/data/base
a Program Files/PostgreSQL/12/data/current_logfiles
a Program Files/PostgreSQL/12/data/global
a Program Files/PostgreSQL/12/data/log
a Program Files/PostgreSQL/12/data/pg_commit_ts
a Program Files/PostgreSQL/12/data/pg_dynshmem
a Program Files/PostgreSQL/12/data/pg_hba.conf
a Program Files/PostgreSQL/12/data/pg_ident.conf
a Program Files/PostgreSQL/12/data/pg_logical
a Program Files/PostgreSQL/12/data/pg_multixact
a Program Files/PostgreSQL/12/data/pg_notify
a Program Files/PostgreSQL/12/data/pg_replslot
a Program Files/PostgreSQL/12/data/pg_serial
a Program Files/PostgreSQL/12/data/pg_snapshots
a Program Files/PostgreSQL/12/data/pg_stat
a Program Files/PostgreSQL/12/data/pg_stat_tmp
a Program Files/PostgreSQL/12/data/pg_subtrans
a Program Files/PostgreSQL/12/data/pg_tblspc
a Program Files/PostgreSQL/12/data/pg_twophase
a Program Files/PostgreSQL/12/data/PG_VERSION
a Program Files/PostgreSQL/12/data/pg_wal
a Program Files/PostgreSQL/12/data/pg_xact
a Program Files/PostgreSQL/12/data/postgresql.auto.conf
a Program Files/PostgreSQL/12/data/postgresql.conf
a Program Files/PostgreSQL/12/data/postmaster.opts
a Program Files/PostgreSQL/12/data/pg_xact/0000
a Program Files/PostgreSQL/12/data/pg_wal/000000010000000000000002
a Program Files/PostgreSQL/12/data/pg_wal/000000010000000000000003
```

Windows:

Linux :



```
[postgres@rhel8 user_backups]$
[postgres@rhel8 user_backups]$ pg_ctl stop
waiting for server to shut down.... done
server stopped
[postgres@rhel8 user_backups]$ pg_ctl status
pg_ctl: no server running
[postgres@rhel8 user_backups]$ pwd
/opt/user_backups
[postgres@rhel8 user_backups]$ cd /var/lib/pgsql/12/data
[postgres@rhel8 data]$ pwd
/var/lib/pgsql/12/data
[postgres@rhel8 data]$ cd /opt/user_backups/
[postgres@rhel8 user_backups]$ tar -cvzf data_backup.tar /var/lib/pgsql/12/data
```



```
/var/lib/pgsql/12/data/log/postgresql-Sat.log
/var/lib/pgsql/12/data/log/postgresql-Mon.log
/var/lib/pgsql/12/data/log/postgresql-Sun.log
/var/lib/pgsql/12/data/current_logfiles
[postgres@rhel8 user_backups]$ ls -ltr
total 5704
-rw-r--r--. 1 postgres postgres     917 Aug  1 13:34 major_bkp
-rw-r--r--. 1 postgres postgres    3789 Aug  1 14:01 clusterall_bkp
-rw-r--r--. 1 postgres postgres     920 Aug  2 12:27 clusterall_bkp.gz
-rw-r--r--. 1 postgres postgres     767 Aug  2 16:14 major_bkp.gz
-rw-r--r--. 1 postgres postgres     294 Aug  2 16:29 major1.gz
-rw-r--r--. 1 postgres postgres     489 Aug  2 16:48 major.dump.aa
-rw-r--r--. 1 postgres postgres 5815153 Aug  2 19:59 data_backup.tar
[postgres@rhel8 user_backups]$
```

Restore Splited backup files :

```
$ cat clusterallbkp* | psql -h localhost -U postgres -W -d mydb
```

Compressed splited files restored

```
$ cat mydb.sql.gz* | gunzip | psql -h localhost -U postgres -W -d mydb
```

Convert binary dump to plain text

```
$ pg_Restore binbkp.dump -f plainbkp.sql
```

Continuous Archiving and PITR :

- Continuous Archiving is the process of archiving Write Ahead Log (WAL) files.
- Point-in-Time-Recovery(PITR) refers to PostgreSQL’s ability to start from the restore of a full backup and apply archived WAL files up to a specified timestamp.

Steps to setup up Continuous Archiving and PITR

Setting Up WAL Archiving

Making a Base Backup

Making a Base Backup Using the Low Level API

Recovering Using a Continuous Archive Backup

First we check the status that archive in ON or Off. Below command will be used in psql.

```
#show archive_mode
```

returns On or Off.

If this is off , we need to make it on by editing the postgresql.conf. Before doing this, need to ensure that postgres cluster is shutdown (#pg_ctl stop).

Check the cluster status by psql# pg_ctl status.

If the cluster is stopped , we can set the below parameter in postgresql.conf file.

1. wal_level – replica or archive

2. archive_mode = on

3. archive_command = ‘copy “%p” “c:\\archive_location\\%f”’ ##for Windows

3. archive_command= ‘cp -i %p /opt/archivedir/%f‘ ## for linux

archive_command is the most imp parameter which will copy the wal files from one location to another safe location ie. Archive directory.

After the parameter is set , start the cluster psql # pg_ctl start.

Check the destination location if any files are there, if not we can do a test by firing the below command in psql

# select pg_start_backup(‘test1’); -- we are telling postgresql that backup is started.

check the archive folder.

#select pg_stop_backup();

check the archive folder the files copied.

To stop archiving , comment # the parameter in postgresql.conf.

Online backup using low level API :

- Ensure that WAL archiving is enabled and working.
- Select either Exclusive or Non-Exclusive low level API backup.

- As super user or user with Execute priviliges issue command “select pg_start_backup(‘label’,false, false)”
- pg_start_backup performs a checkpoint right away and it takes sometime to finish.
- If we want the control to return faster use “select pg_start_backup(’label’,true); “ it performs checkpoint faster.
- Use file system utility to backup the data directory.
- Run “select pg_stop_backup();” to stop the backup mode.
- All wal segments would have been archived to form a complete set of backup files.

However, archive files alone will not help us in restoring a database in case of a failure.The core of any backup and restore strategy is full backup or the base backup.Archive files work in conjunction with full backup in order to restore a database until a certain time.

For example.
I have taken a full backup or a base backup on Sunday and my system crashed somewhere on Monday 10 am.

Now, in order to recover my system until 10 a.m. on Monday, I have to restore my full backup along with all the archive files till Monday 10 am. If I don't have a full back up, then all the archive files are meaningless.

I cannot restore my system.
so the base back up is very critical and important.

In this section, we are going to learn how to take a full online backup.

or, full online basebackup. This is a preferred way of backing up any production system or any important system. In this type of backup, the system is online and available for use, the users are connected and they are doing their transactions while the backup is happening in the background.

The reason this type of backup is preferred because in real time, we cannot afford to have a down time, the system availability is a huge criteria and the expectation from the business is that the system should be available 24 by seven.

We cannot shut down a system for us to take a backup.
There are two ways how we can take online backup or online backup.

One is through a low level API, which is what we are going to discuss now.

Another one is using a tool called pg_basebackup.

How to take Low level API backup :

```
psql # select pg_start_backup(‘backup2’,false,false);
```

backup2 – backup label.

False - That I'll give the parameter false, which is to inform PG that it can take its time for doing the IO and not return the control back to the user immediately.

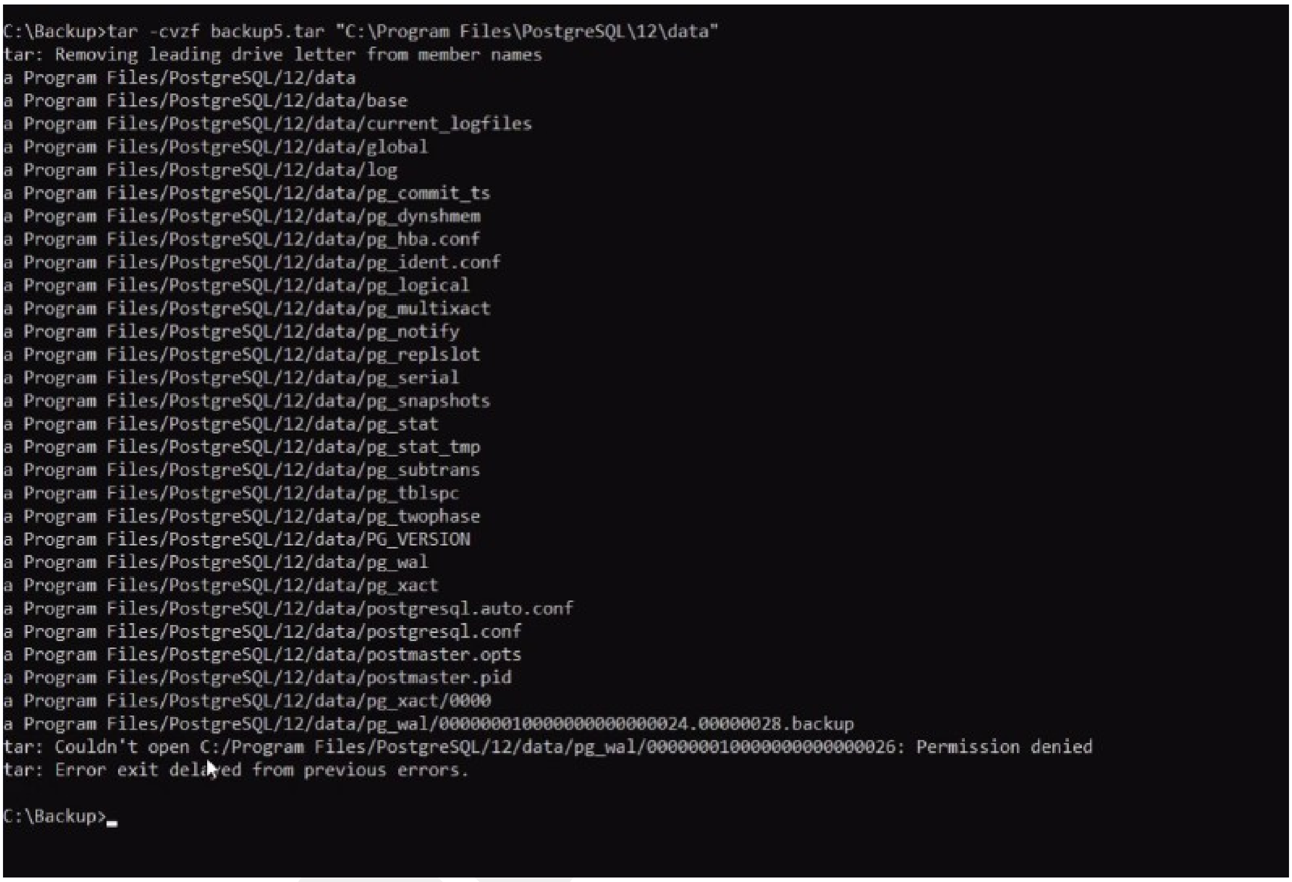
False – it informs to PG that I want to do a non exclusive backup.

Now run the command

```
psql # \! tar cvzf backup2.tar “c:\program files\postgresql\12\data”        -
windows
```

psql # \! tar cvzf backup2.tar /var/lib/pgsql/12/data - Linux

it will take the online backup in backup2.tar file.

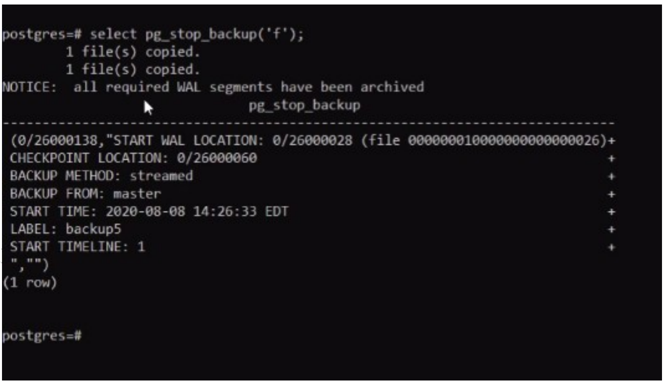


```
C:\Backup>tar -cvzf backup5.tar "C:\Program Files\PostgreSQL\12\data"
tar: Removing leading drive letter from member names
a Program Files/PostgreSQL/12/data
a Program Files/PostgreSQL/12/data/base
a Program Files/PostgreSQL/12/data/current_logfiles
a Program Files/PostgreSQL/12/data/global
a Program Files/PostgreSQL/12/data/log
a Program Files/PostgreSQL/12/data/pg_commit_ts
a Program Files/PostgreSQL/12/data/pg_dynshmem
a Program Files/PostgreSQL/12/data/pg_hba.conf
a Program Files/PostgreSQL/12/data/pg_ident.conf
a Program Files/PostgreSQL/12/data/pg_logical
a Program Files/PostgreSQL/12/data/pg_multixact
a Program Files/PostgreSQL/12/data/pg_notify
a Program Files/PostgreSQL/12/data/pg_replslot
a Program Files/PostgreSQL/12/data/pg_serial
a Program Files/PostgreSQL/12/data/pg_snapshots
a Program Files/PostgreSQL/12/data/pg_stat
a Program Files/PostgreSQL/12/data/pg_stat_tmp
a Program Files/PostgreSQL/12/data/pg_subtrans
a Program Files/PostgreSQL/12/data/pg_tblspc
a Program Files/PostgreSQL/12/data/pg_twophase
a Program Files/PostgreSQL/12/data/PG_VERSION
a Program Files/PostgreSQL/12/data/pg_wal
a Program Files/PostgreSQL/12/data/pg_xact
a Program Files/PostgreSQL/12/data/postgresql.auto.conf
a Program Files/PostgreSQL/12/data/postgresql.conf
a Program Files/PostgreSQL/12/data/postmaster.opts
a Program Files/PostgreSQL/12/data/postmaster.pid
a Program Files/PostgreSQL/12/data/pg_xact/0000
a Program Files/PostgreSQL/12/data/pg_wal/000000010000000000000024.00000028.backup
tar: Couldn't open C:/Program Files/PostgreSQL/12/data/pg_wal/000000010000000000000026: Permission denied
tar: Error exit delayed from previous errors.

C:\Backup>
```

In above image we can see the permission denied error, actually this log file is in use. When we stop the backup it will automatically copy to archive folder.

Stop the backup:

psql # select pg_stop_backup('f'); -- f means non exclusive backup , other wise t.



```
postgres=# select pg_stop_backup('f');
        1 file(s) copied.
        1 file(s) copied.
NOTICE:  all required WAL segments have been archived
                           pg_stop_backup
---------------------------------------------------------------------------
 (0/26000138,"START WAL LOCATION: 0/26000028 (file 000000010000000000000026)+
 CHECKPOINT LOCATION: 0/26000060                                            +
 BACKUP METHOD: streamed                                                    +
 BACKUP FROM: master                                                        +
 START TIME: 2020-08-08 14:26:33 EDT                                        +
 LABEL: backup5                                                             +
 START TIMELINE: 1                                                          +
 ","")
(1 row)


postgres=#
```

You can use this information when you are doing a point in time recovery.

You can specify the stop back up time and say that I want to recover my database.

Now we have the complete online backup with the last wal files. If we want to restore it we can untar the backup2.tar copy the wal file in folder and start cluster.

Pg_basebackup:

- pg_basebackup is used to take base backups of a running PostgreSQL database cluster.(Online).
- This backup can be used for PITR or Replication.
- It automatically puts and take out the database from backup mode.
- Backups are always taken of the entire database cluster and cannot be used for single database or objects.

Syntax: pg_basebackup –D <backup directory location>

There are no direct command for incremental or differential backup in postgreSQL. But it can be managed. If the databases are small, we ideally prefer taking full backup, which is pg_basebackup backup fully on a daily basis.

But for a larger database. This is not possible because the backup me takes hours to get completed.

So the other scenario is. We get a full backup on sunday. Then we have all our Wal files which are created and archived.

On Monday, by the end of the day, end of business by five or five thirty, I would go ahead and take a backup of all the archive files, which are generated from my last backup to end up in this Monday.

repeat the same on Tuesday, Wednesday, Thursday, Friday, Saturday. Then all this backup, the full backup plus all the archive files , I take an entire backup on tape for that particular week, so I have a full backup and all archive for that particularly week.

then I flush everything out and Sunday we start the routine again.

Below are pg_basebackup command.

C:/> pg_basebackup -h localhost -U posrgres -p 5432 -D “C:\\mybasebackup\” -z -P -Ft -Xs

-h : host

-U: user

-p : port

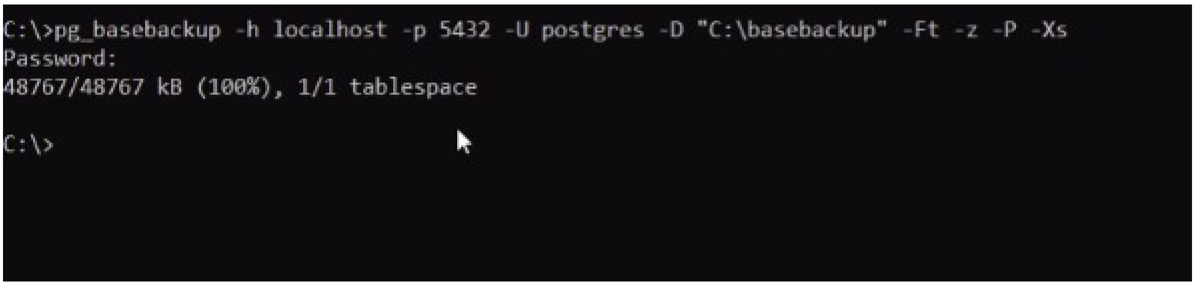
-D : destination folder

-z : zip

-P : show progress

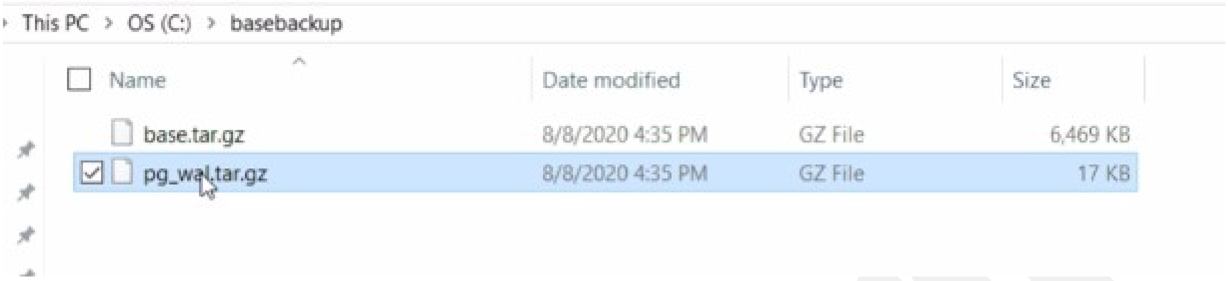
-Ft : format tar.

-Xs: **also archive the transaction which is happening during the backup** itself.



```
C:\>pg_basebackup -h localhost -p 5432 -U postgres -D "C:\basebackup" -Ft -z -P -Xs
Password:
48767/48767 kB (100%), 1/1 tablespace

C:\>
```

See the basebackup and wal files during backup running.



This PC › OS (C:) › basebackup

☐ Name	Date modified	Type	Size
base.tar.gz	8/8/2020 4:35 PM	GZ File	6,469 KB
☑ pg_wal.tar.gz	8/8/2020 4:35 PM	GZ File	17 KB

For non zip format :

```
#pg_basebackup  -U postgres -D “C:\\mybasebackup\”  -P -Fp -Xs
      – windows

#pg_basebackup   -D /opt/mybasebackup/  -P -Fp -Xs                    – linux
```

PITR and Recovery Configuration:

- Point-in-Time-Recovery(PITR) refers to PostgreSQL’s ability to restore and recover a postgresql cluster until a specified point.

Now, for example, I assume there is disk failure or got corrupted in data or drop the database accidentally. It becomes a job of a DBA to recover and restore a database until the time of the failure. So in order to do that, I need to have strong and reliable backup strategy, which will help me to restore my database.

- Restore Command : This command is used to retrieve archive segment of the WAL Files.

Example (postgresql.conf**):**

restore_command = 'cp /mnt/server/archivedir/%f "%p“’ # **Linux** restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"' # **Windows**

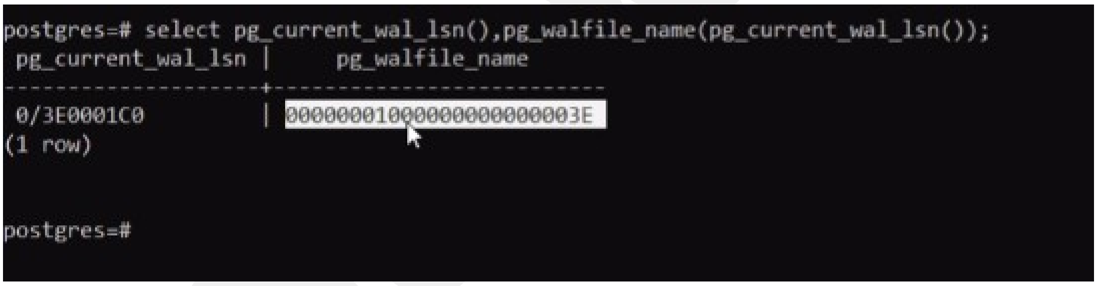
Recovery_Target = immediate (This parameter specifies the recovery should end as soon as a consistent state is reached).

Recover_Target_Lsn – This parameter specifies the LSN of the write-ahead log location up to which the recovery will proceed. It is log Sequence number (LSN) we can instruct PG to restore until the specified **log sequence number** generated while log generation.

To see the current WAL LSN number below is the command.

Psql # *select pg_current_wal_lsn(),pg_wal_file_name(pg_current_wal_lsn());*

#output



```
postgres=# select pg_current_wal_lsn(),pg_walfile_name(pg_current_wal_lsn());
 pg_current_wal_lsn |     pg_walfile_name
--------------------+--------------------------
 0/3E0001C0         | 00000001000000000000003E
(1 row)


postgres=#
```

- Recovery_Target_Name = This parameter specifies the named restore point(create with pg_create_restore_point)) to which recovery will proceed.
- Recovery_Target_Time = This parameter specifies the time stamp up to which recovery will proceed. (in 95% cases it uses)
- Recovery_Target_Xid = The parameter specifies the transaction ID upto which recovery will proceed.

- Recovery_Target_Inclusive = Specifies whether to stop the recovery just after the target is reached(on) or just before the recovery target(off). Default is On. Note : Lsn, name, time, xid use only one at the time we cant use all of them.

Steps to perform PITR

- Stop the server, if it's running. (#pg_ctl stop)
- copy the whole old cluster data directory and any tablespaces to a temporary location.
- Remove all directories and files from cluster data directory.
- Restore the database files from your file system backup or base backup. (don't move)
- Remove old files present in pg_wal directory in the file system backup.
- If there are any unarchived WAL segment files recovered from crashed cluster, copy them to pg_wal directory.
- Set recovery configuration settings in postgresql.conf and create a file recovery.signal in the cluster data directory. Before version 12 we used to have a file recovery.conf. We used to set all the recovery parameters in recovery.conf file. we use to create it and save it. whenever PG recovers, starts the recovery , It will first look for this recovery.conf file whether it is available or not. If it is available, it will read the parameter and start the recovery.This is how it was before 12 version. From version 12 these all parameters included in postgresql.conf. Only empty file recovery.signal need to be in data directory. When PG starts the recovery, it will check data folder that there is any file recovery signal. If it finds a file called recovery.signal, it will go ahead and start the recovery.

- Temporarily modify pg_hba.conf to prevent ordinary users from connecting. Stop / block all the connections.
- Start the server (pg_ctl start) . The server will go into recovery mode and proceed to read through the archived WAL files it needs.
- Upon completion of the recovery process, the server will remove recovery.signal.
- Inspect the contents of the database to ensure the recovered database is in desired state.
- Modify pg_hba.conf to allow users to connect to the database.

How to recover database using pg_basebackup & how to perform PITR using pg_basebackup

Prerequisite:

Ensure Archive Mode is turned on and archiving is happening.

How to recover database using online pg_basebackup:

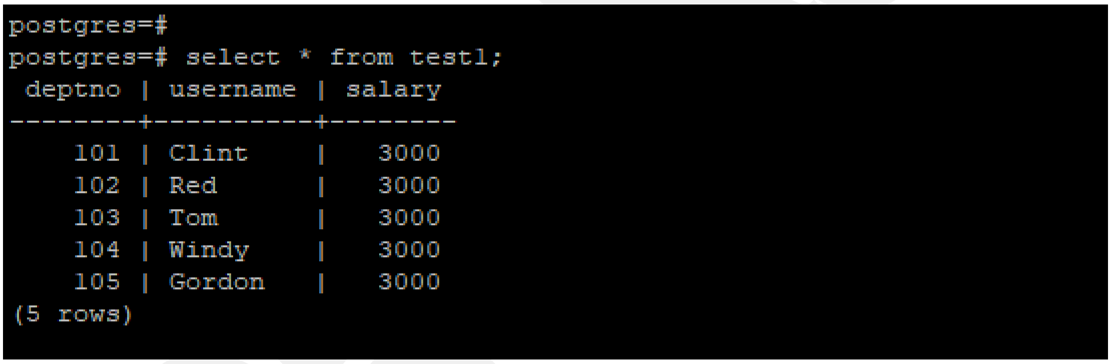
Example:

Step 1:

Create a table and insert few records, update or delete (perform some transaction which can be archived)

I have created a table named test1 and did few operations on it like insert and update.

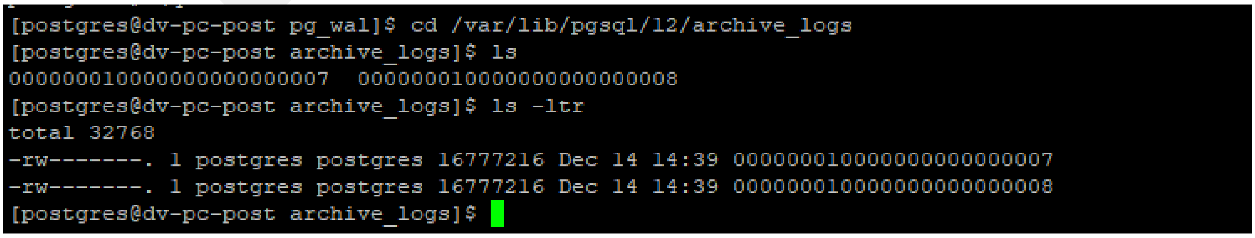
Currently my table has 5 rows.



```
postgres=#
postgres=# select * from test1;
 deptno | username | salary
--------+----------+--------
    101 | Clint    |   3000
    102 | Red      |   3000
    103 | Tom      |   3000
    104 | Windy    |   3000
    105 | Gordon   |   3000
(5 rows)
```

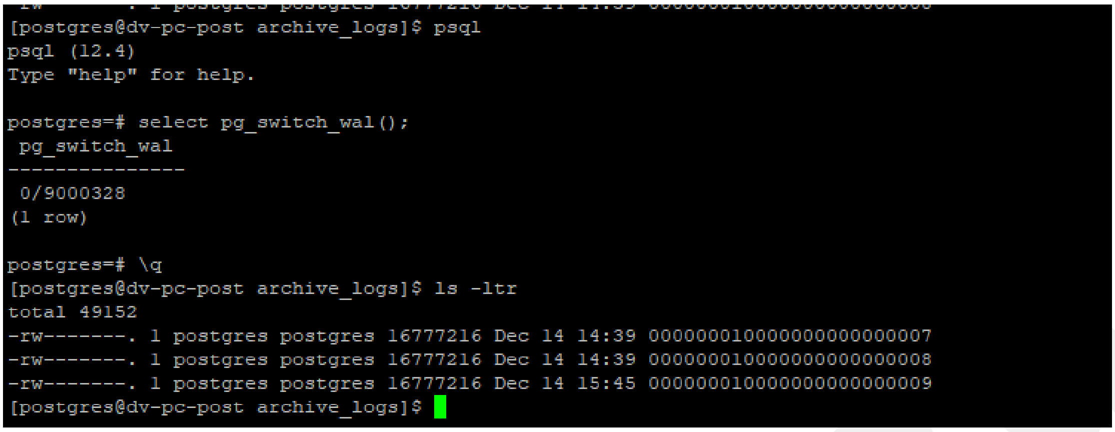
Step 2:

Verify archive log folder whether all wal files are archived.



```
[postgres@dv-pc-post pg_wal]$ cd /var/lib/pgsql/12/archive_logs
[postgres@dv-pc-post archive_logs]$ ls
000000010000000000000007  000000010000000000000008
[postgres@dv-pc-post archive_logs]$ ls -ltr
total 32768
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000007
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000008
[postgres@dv-pc-post archive_logs]$
```

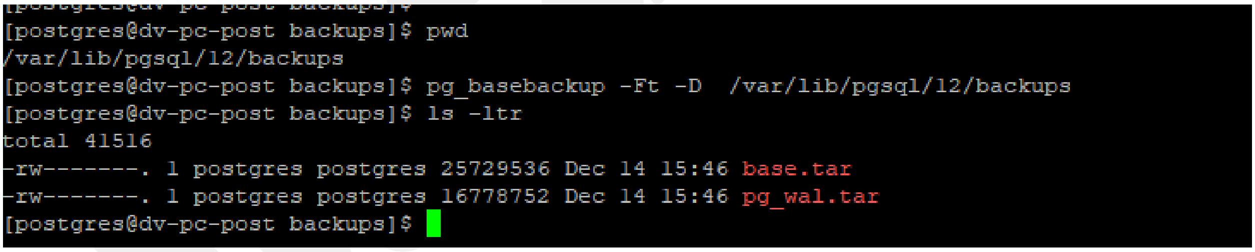
Step 3:
Perform a log switch to archive the current log



```
[postgres@dv-pc-post archive_logs]$ psql
psql (12.4)
Type "help" for help.

postgres=# select pg_switch_wal();
 pg_switch_wal
---------------
 0/9000328
(1 row)

postgres=# \q
[postgres@dv-pc-post archive_logs]$ ls -ltr
total 49152
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000007
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000008
-rw-------. 1 postgres postgres 16777216 Dec 14 15:45 000000010000000000000009
[postgres@dv-pc-post archive_logs]$
```

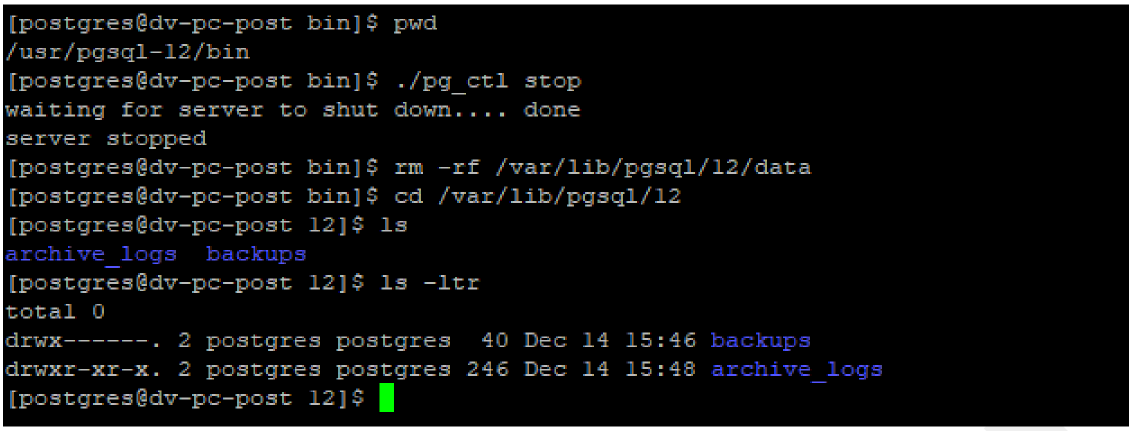
As you can see the current log is archived as well.

Step 4:
Take a pg_basebackup of the entire cluster. Using the below command.
This will generate 2 tar files
One is of the entire data and other one is wal files(pg_wal.tar)
$ pg_basebackup -Ft -D *var*/pgtmp – example syntax



```
[postgres@dv-pc-post backups]$ pwd
/var/lib/pgsql/12/backups
[postgres@dv-pc-post backups]$ pg_basebackup -Ft -D  /var/lib/pgsql/12/backups
[postgres@dv-pc-post backups]$ ls -ltr
total 41516
-rw-------. 1 postgres postgres 25729536 Dec 14 15:46 base.tar
-rw-------. 1 postgres postgres 16778752 Dec 14 15:46 pg_wal.tar
[postgres@dv-pc-post backups]$
```

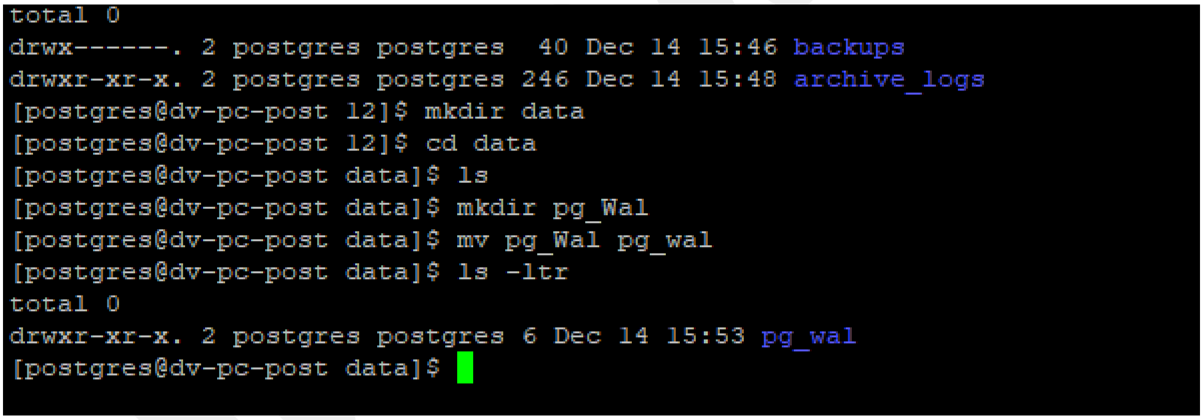
Step 5:
Stop the cluster and delete the data folder(we are going to mimic a crash here were we lost our data folder).



```
[postgres@dv-pc-post bin]$ pwd
/usr/pgsql-12/bin
[postgres@dv-pc-post bin]$ ./pg_ctl stop
waiting for server to shut down.... done
server stopped
[postgres@dv-pc-post bin]$ rm -rf /var/lib/pgsql/12/data
[postgres@dv-pc-post bin]$ cd /var/lib/pgsql/12
[postgres@dv-pc-post 12]$ ls
archive_logs  backups
[postgres@dv-pc-post 12]$ ls -ltr
total 0
drwx------. 2 postgres postgres  40 Dec 14 15:46 backups
drwxr-xr-x. 2 postgres postgres 246 Dec 14 15:48 archive_logs
[postgres@dv-pc-post 12]$
```

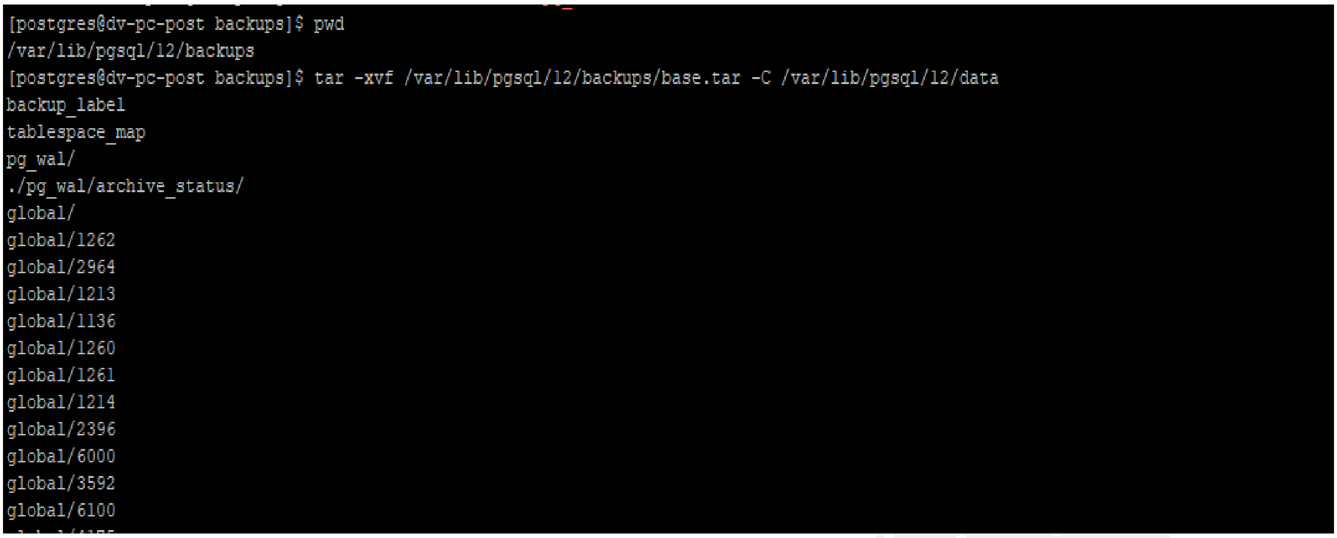
I removed the data folder using rm –rf and you can see that there is no data folder in the location.

Step 6: Now we are going to restore the data folder using the backup which we took.
First create data folder in the same location where we removed.
Second move in to the data folder and create pg_wal folder.



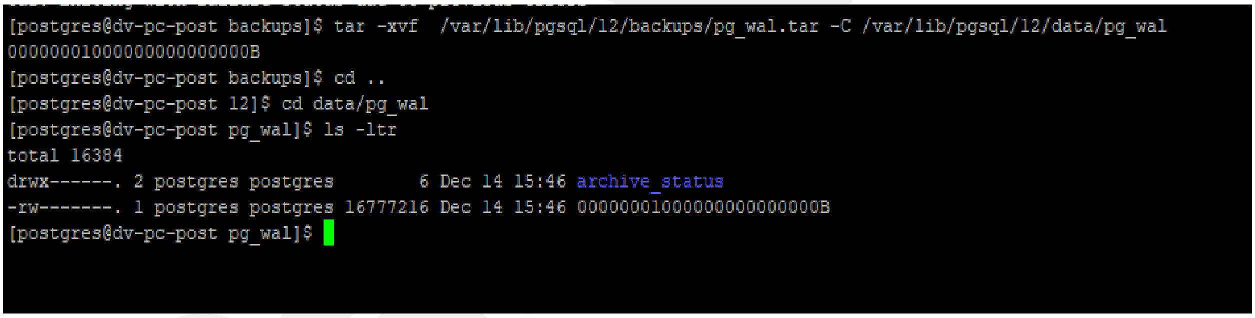
```
total 0
drwx------. 2 postgres postgres  40 Dec 14 15:46 backups
drwxr-xr-x. 2 postgres postgres 246 Dec 14 15:48 archive_logs
[postgres@dv-pc-post 12]$ mkdir data
[postgres@dv-pc-post 12]$ cd data
[postgres@dv-pc-post data]$ ls
[postgres@dv-pc-post data]$ mkdir pg_Wal
[postgres@dv-pc-post data]$ mv pg_Wal pg_wal
[postgres@dv-pc-post data]$ ls -ltr
total 0
drwxr-xr-x. 2 postgres postgres 6 Dec 14 15:53 pg_wal
[postgres@dv-pc-post data]$
```

Let us start with the restore of data folder and wal files using the pg_basebackup which we took.



```
[postgres@dv-pc-post backups]$ pwd
/var/lib/pgsql/12/backups
[postgres@dv-pc-post backups]$ tar -xvf /var/lib/pgsql/12/backups/base.tar -C /var/lib/pgsql/12/data
backup_label
tablespace_map
pg_wal/
./pg_wal/archive_status/
global/
global/1262
global/2964
global/1213
global/1136
global/1260
global/1261
global/1214
global/2396
global/6000
global/3592
global/6100
```

This will create all the data inside the data folder. Log in to data folder and check whether all the folder and files are there.
Now let us restore the wal files from wal backup.

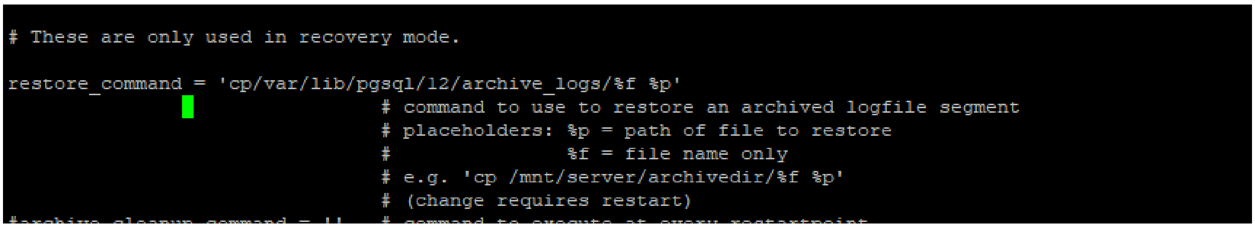


```
[postgres@dv-pc-post backups]$ tar -xvf  /var/lib/pgsql/12/backups/pg_wal.tar -C /var/lib/pgsql/12/data/pg_wal
00000001000000000000000B
[postgres@dv-pc-post backups]$ cd ..
[postgres@dv-pc-post 12]$ cd data/pg_wal
[postgres@dv-pc-post pg_wal]$ ls -ltr
total 16384
drwx------. 2 postgres postgres        6 Dec 14 15:46 archive_status
-rw-------. 1 postgres postgres 16777216 Dec 14 15:46 00000001000000000000000B
[postgres@dv-pc-post pg_wal]$
```

You can see that the wal files are restored in the pg_wal folder.

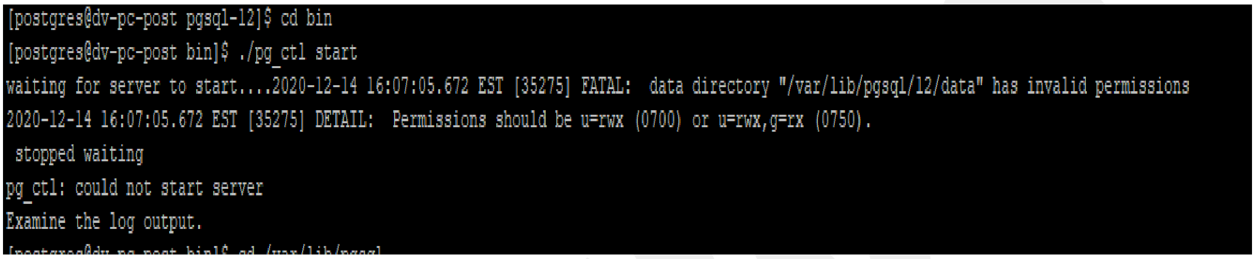
Step 7: Now we need to ensure that the database is consistent and tell our database server to copy files from our archived location to WAL file location. For this we need to edit postgresql.conf file.

Add the following entry in postgresql.conf
Restore_command = 'cp/var/lib/pgsql/12/archive_logs/%f %p'



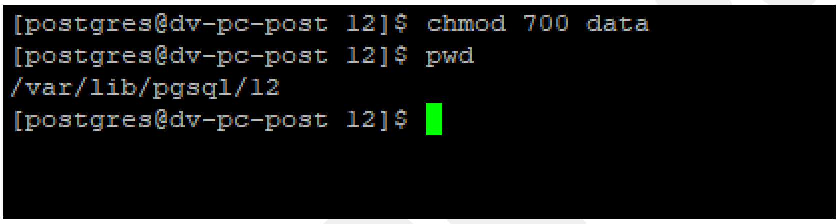
```
# These are only used in recovery mode.

restore_command = 'cp/var/lib/pgsql/12/archive_logs/%f %p'
                                # command to use to restore an archived logfile segment
                                # placeholders: %p = path of file to restore
                                #               %f = file name only
                                # e.g. 'cp /mnt/server/archivedir/%f %p'
                                # (change requires restart)
#archive_cleanup_command = ''   # command to execute at every restartpoint
```

Step 8: Start the cluster. (Remember all this while the cluster was down). You may get error like this when you start (Permission error on data directory)



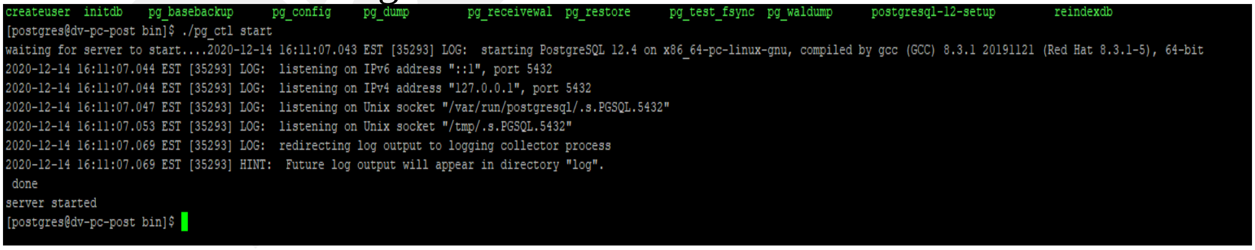
```
[postgres@dv-pc-post pgsql-12]$ cd bin
[postgres@dv-pc-post bin]$ ./pg_ctl start
waiting for server to start....2020-12-14 16:07:05.672 EST [35275] FATAL:  data directory "/var/lib/pgsql/12/data" has invalid permissions
2020-12-14 16:07:05.672 EST [35275] DETAIL:  Permissions should be u=rwx (0700) or u=rwx,g=rx (0750).
 stopped waiting
pg_ctl: could not start server
Examine the log output.
[postgres@dv-pc-post bin]$ cd /var/lib/pgsql
```

Just change the **permission on data directory to 700**



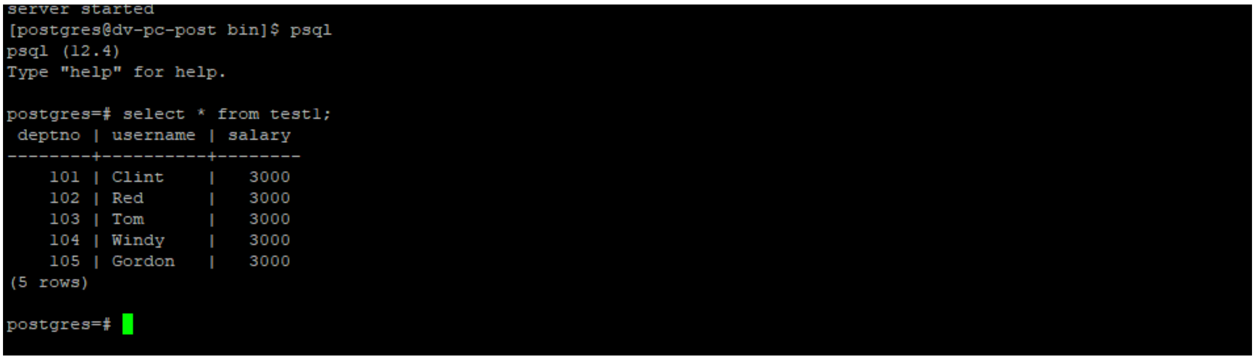
```
[postgres@dv-pc-post 12]$ chmod 700 data
[postgres@dv-pc-post 12]$ pwd
/var/lib/pgsql/12
[postgres@dv-pc-post 12]$
```

And start the cluster again



```
createuser  initdb    pg_basebackup     pg_config     pg_dump        pg_receivewal  pg_restore      pg_test_fsync  pg_waldump      postgresql-12-setup        reindexdb
[postgres@dv-pc-post bin]$ ./pg_ctl start
waiting for server to start....2020-12-14 16:11:07.043 EST [35293] LOG:  starting PostgreSQL 12.4 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5), 64-bit
2020-12-14 16:11:07.044 EST [35293] LOG:  listening on IPv6 address "::1", port 5432
2020-12-14 16:11:07.044 EST [35293] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2020-12-14 16:11:07.047 EST [35293] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2020-12-14 16:11:07.053 EST [35293] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
2020-12-14 16:11:07.069 EST [35293] LOG:  redirecting log output to logging collector process
2020-12-14 16:11:07.069 EST [35293] HINT:  Future log output will appear in directory "log".
 done
server started
[postgres@dv-pc-post bin]$
```

The cluster started successfully. Let us check whether the table test1 with 5 records exist or not.



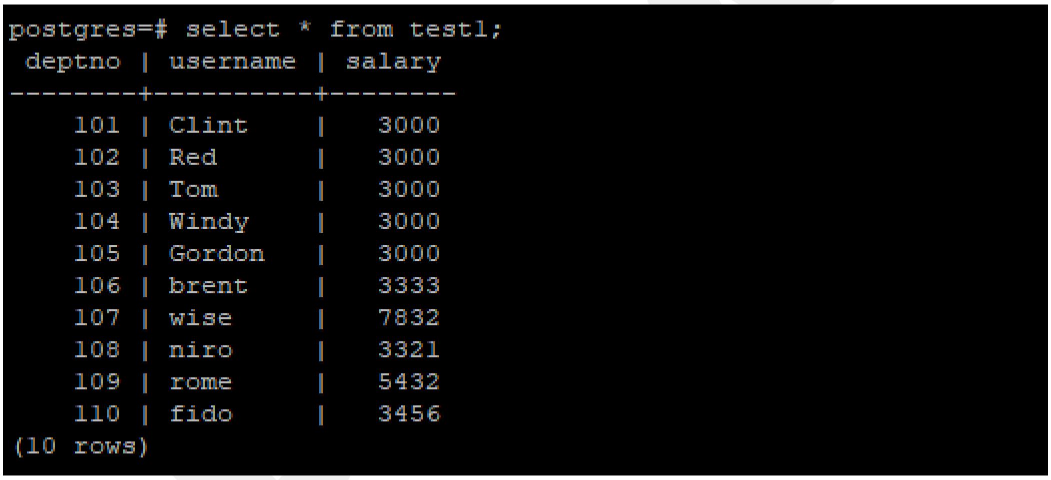
```
server started
[postgres@dv-pc-post bin]$ psql
psql (12.4)
Type "help" for help.

postgres=# select * from test1;
 deptno | username | salary
--------+----------+--------
    101 | Clint    |   3000
    102 | Red      |   3000
    103 | Tom      |   3000
    104 | Windy    |   3000
    105 | Gordon   |   3000
(5 rows)

postgres=#
```

So we have successfully deleted and restored our database using pg_basebackup.

PITR

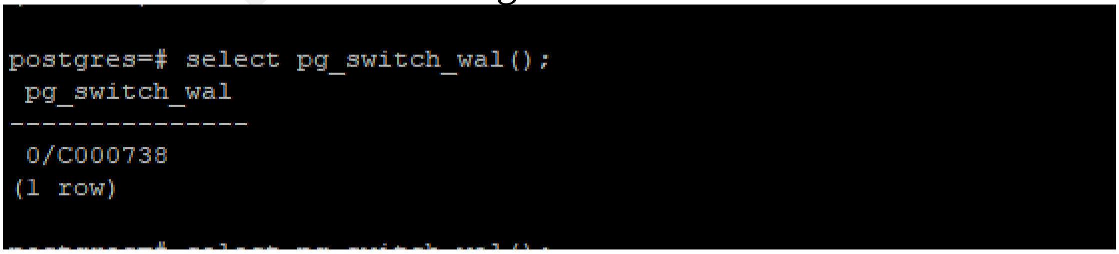
Let us try now to a PITR.

Step 1: We will use the same table to perform PITR. I will be adding few more row to the existing table to generate archives.



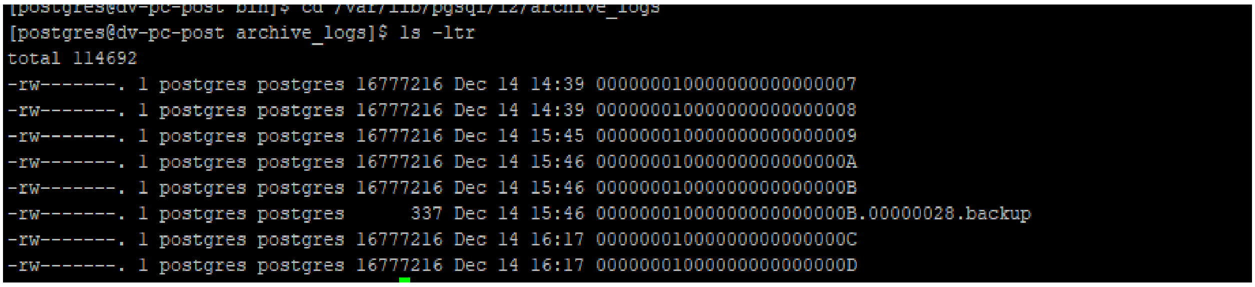
```
postgres=# select * from test1;
 deptno | username | salary
--------+----------+--------
    101 | Clint    |   3000
    102 | Red      |   3000
    103 | Tom      |   3000
    104 | Windy    |   3000
    105 | Gordon   |   3000
    106 | brent    |   3333
    107 | wise     |   7832
    108 | niro     |   3321
    109 | rome     |   5432
    110 | fido     |   3456
(10 rows)
```

Switch the current archive log.



```
postgres=# select pg_switch_wal();
 pg_switch_wal
---------------
 0/C000738
(1 row)

postgres=# select pg_switch_wal();
```

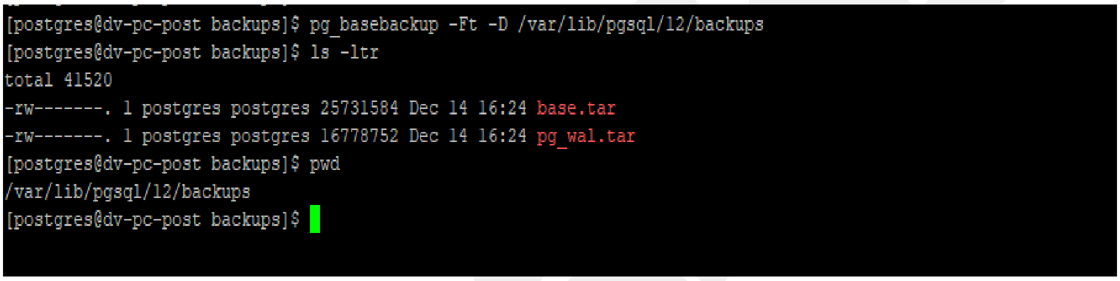
Now check the archive log folder whether we got any new archives.



```
[postgres@dv-pc-post bin]$ cd /var/lib/pgsql/12/archive_logs
[postgres@dv-pc-post archive_logs]$ ls -ltr
total 114692
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000007
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000008
-rw-------. 1 postgres postgres 16777216 Dec 14 15:45 000000010000000000000009
-rw-------. 1 postgres postgres 16777216 Dec 14 15:46 00000001000000000000000A
-rw-------. 1 postgres postgres 16777216 Dec 14 15:46 00000001000000000000000B
-rw-------. 1 postgres postgres      337 Dec 14 15:46 00000001000000000000000B.00000028.backup
-rw-------. 1 postgres postgres 16777216 Dec 14 16:17 00000001000000000000000C
-rw-------. 1 postgres postgres 16777216 Dec 14 16:17 00000001000000000000000D
```

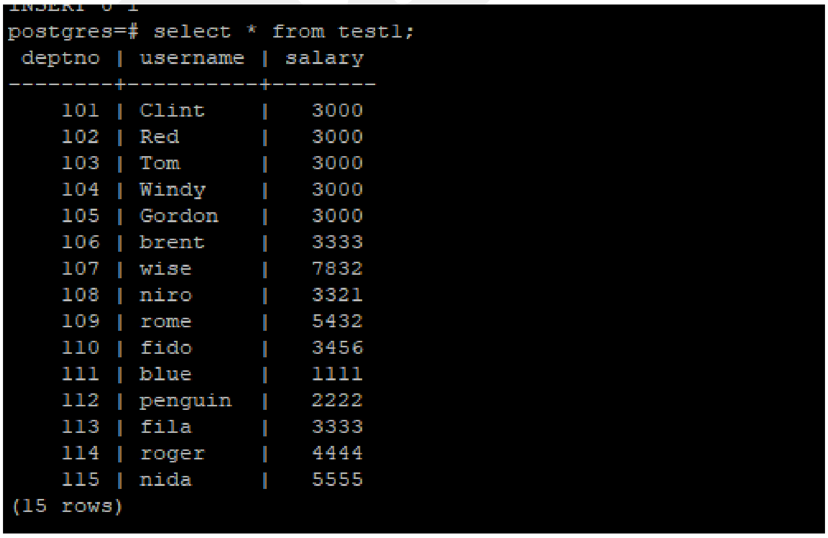
We can see that there are many new archives which are generated in the archive log folder.

Step 2:
Take a fresh pg_basebackup.



```
[postgres@dv-pc-post backups]$ pg_basebackup -Ft -D /var/lib/pgsql/12/backups
[postgres@dv-pc-post backups]$ ls -ltr
total 41520
-rw-------. 1 postgres postgres 25731584 Dec 14 16:24 base.tar
-rw-------. 1 postgres postgres 16778752 Dec 14 16:24 pg_wal.tar
[postgres@dv-pc-post backups]$ pwd
/var/lib/pgsql/12/backups
[postgres@dv-pc-post backups]$
```

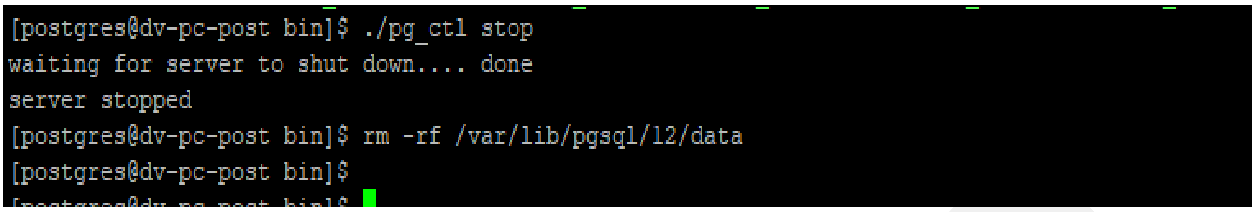
Step 3:
Now insert few more rows in the test1 table after that backup. Before my row count was 10 now it is 15.



```
INSERT 0 1
postgres=# select * from test1;
 deptno | username | salary
--------+----------+--------
    101 | Clint    |   3000
    102 | Red      |   3000
    103 | Tom      |   3000
    104 | Windy    |   3000
    105 | Gordon   |   3000
    106 | brent    |   3333
    107 | wise     |   7832
    108 | niro     |   3321
    109 | rome     |   5432
    110 | fido     |   3456
    111 | blue     |   1111
    112 | penguin  |   2222
    113 | fila     |   3333
    114 | roger    |   4444
    115 | nida     |   5555
(15 rows)
```

My task is restore the database when the table was with 10 records.

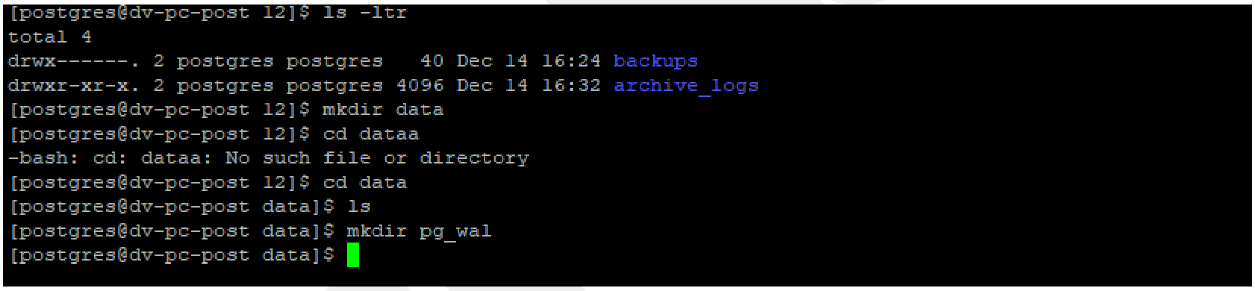
Step 4:
Now I will mimic a crash by deleting my data directory.



```
[postgres@dv-pc-post bin]$ ./pg_ctl stop
waiting for server to shut down.... done
server stopped
[postgres@dv-pc-post bin]$ rm -rf /var/lib/pgsql/12/data
[postgres@dv-pc-post bin]$
[postgres@dv-pc-post bin]$
```

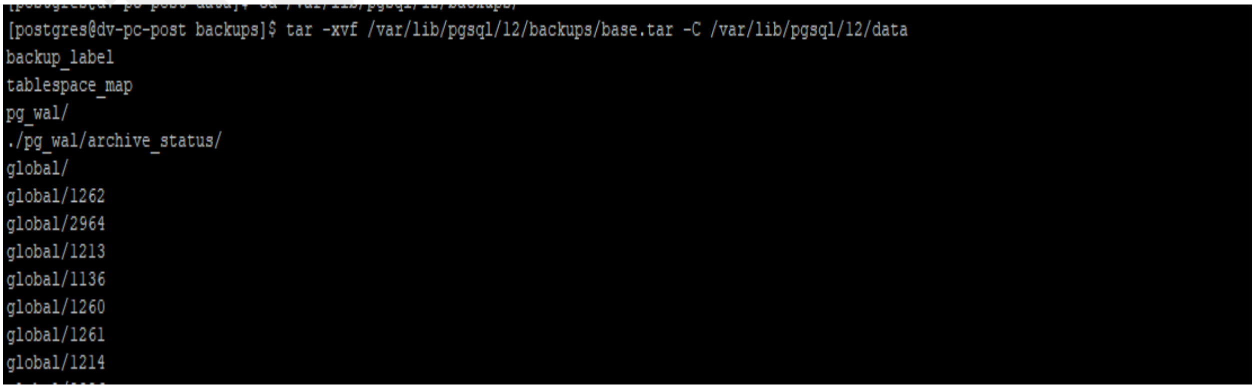
I have removed my data directory.

Step 5:
Restore the database from the backup which we took @ step 2.
Make sure we create the data and pg_wal folder before we start the restore.



```
[postgres@dv-pc-post 12]$ ls -ltr
total 4
drwx------. 2 postgres postgres   40 Dec 14 16:24 backups
drwxr-xr-x. 2 postgres postgres 4096 Dec 14 16:32 archive_logs
[postgres@dv-pc-post 12]$ mkdir data
[postgres@dv-pc-post 12]$ cd dataa
-bash: cd: dataa: No such file or directory
[postgres@dv-pc-post 12]$ cd data
[postgres@dv-pc-post data]$ ls
[postgres@dv-pc-post data]$ mkdir pg_wal
[postgres@dv-pc-post data]$
```

Start the restore operation

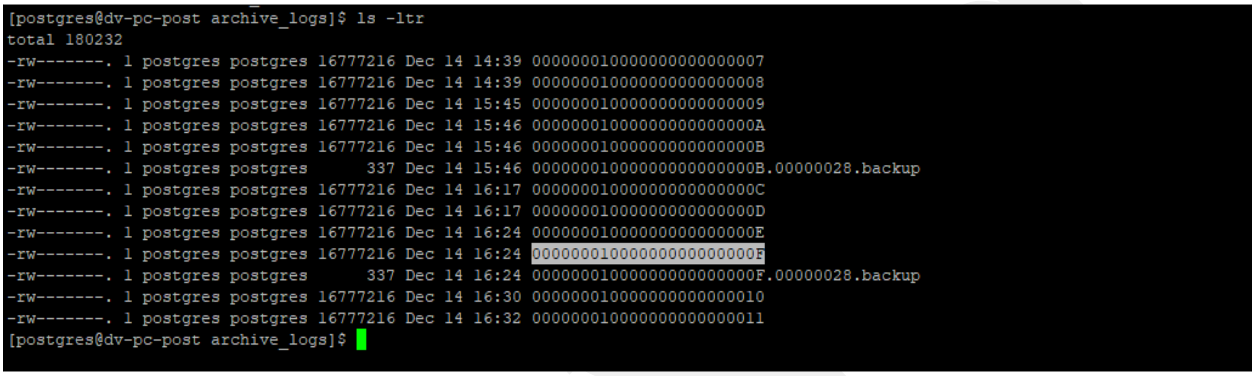


```
[postgres@dv-pc-post data]$ cd /var/lib/pgsql/12/backups/
[postgres@dv-pc-post backups]$ tar -xvf /var/lib/pgsql/12/backups/base.tar -C /var/lib/pgsql/12/data
backup_label
tablespace_map
pg_wal/
./pg_wal/archive_status/
global/
global/1262
global/2964
global/1213
global/1136
global/1260
global/1261
global/1214
```

ONLY DATA DIRECTORY SHOULD BE RESTORED. DON'T RESTORE PG_WAL TAR.

Recovery file will guide the Point in time to backup.

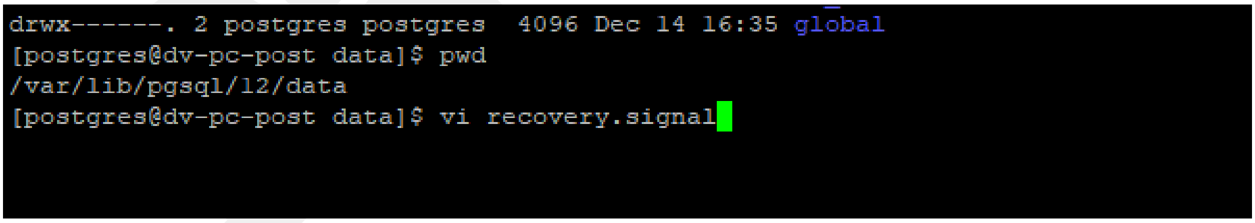
Step 6:
I have to recover my database till the point of 10 records. So I will check the archive log which was generated by that time.



```
[postgres@dv-pc-post archive_logs]$ ls -ltr
total 180232
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000007
-rw-------. 1 postgres postgres 16777216 Dec 14 14:39 000000010000000000000008
-rw-------. 1 postgres postgres 16777216 Dec 14 15:45 000000010000000000000009
-rw-------. 1 postgres postgres 16777216 Dec 14 15:46 00000001000000000000000A
-rw-------. 1 postgres postgres 16777216 Dec 14 15:46 00000001000000000000000B
-rw-------. 1 postgres postgres      337 Dec 14 15:46 00000001000000000000000B.00000028.backup
-rw-------. 1 postgres postgres 16777216 Dec 14 16:17 00000001000000000000000C
-rw-------. 1 postgres postgres 16777216 Dec 14 16:17 00000001000000000000000D
-rw-------. 1 postgres postgres 16777216 Dec 14 16:24 00000001000000000000000E
-rw-------. 1 postgres postgres 16777216 Dec 14 16:24 00000001000000000000000F
-rw-------. 1 postgres postgres      337 Dec 14 16:24 00000001000000000000000F.00000028.backup
-rw-------. 1 postgres postgres 16777216 Dec 14 16:30 000000010000000000000010
-rw-------. 1 postgres postgres 16777216 Dec 14 16:32 000000010000000000000011
[postgres@dv-pc-post archive_logs]$
```

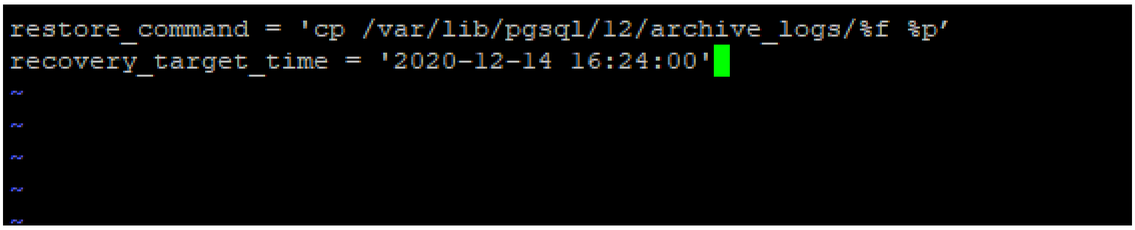
So I am going to recover my database till 16:24 before I did my pg_basebackup. There are few additional archive files @ 16:30 and 16:32 which hold the new 5 records. I don't want that.

Step 7:
Create recovery.signal file inside /data folder.



```
drwx------. 2 postgres postgres  4096 Dec 14 16:35 global
[postgres@dv-pc-post data]$ pwd
/var/lib/pgsql/12/data
[postgres@dv-pc-post data]$ vi recovery.signal
```

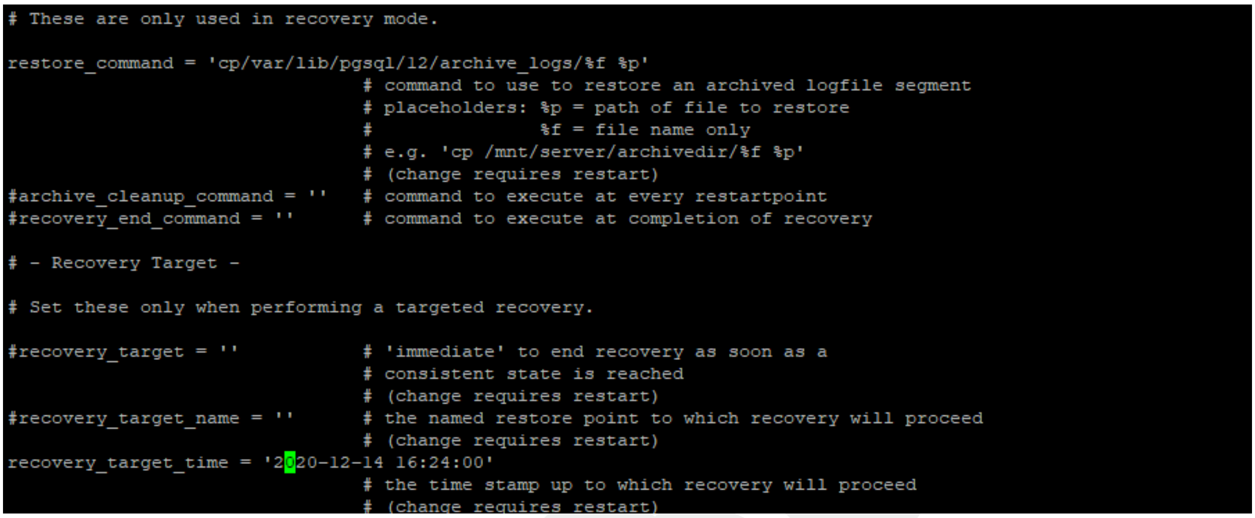
And add the following entries in the file



```
restore_command = 'cp /var/lib/pgsql/12/archive_logs/%f %p'
recovery_target_time = '2020-12-14 16:24:00'
~
~
~
~
~
```

And save the file using wq!. These two parameters are very important for PITR . If we don’t add these two parameters all the archive files will be applied and instead of 10 rows we will get 15 rows.

Copy the two commands in postgresql.conf file as well as

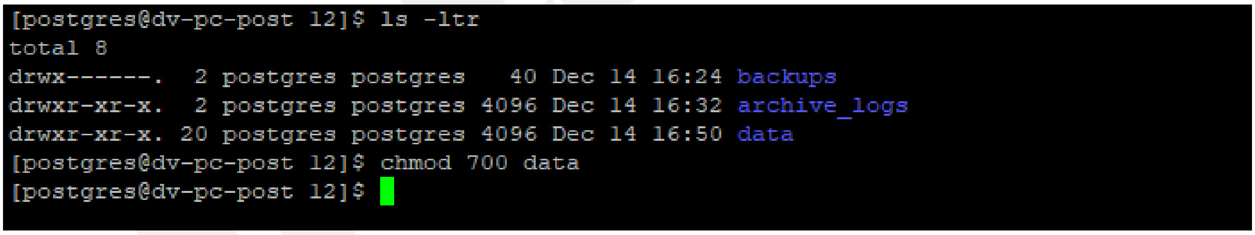


```
# These are only used in recovery mode.

restore_command = 'cp/var/lib/pgsql/12/archive_logs/%f %p'
                                # command to use to restore an archived logfile segment
                                # placeholders: %p = path of file to restore
                                #               %f = file name only
                                # e.g. 'cp /mnt/server/archivedir/%f %p'
                                # (change requires restart)
#archive_cleanup_command = ''   # command to execute at every restartpoint
#recovery_end_command = ''      # command to execute at completion of recovery

# - Recovery Target -

# Set these only when performing a targeted recovery.

#recovery_target = ''           # 'immediate' to end recovery as soon as a
                                # consistent state is reached
                                # (change requires restart)
#recovery_target_name = ''      # the named restore point to which recovery will proceed
                                # (change requires restart)
recovery_target_time = '2020-12-14 16:24:00'
                                # the time stamp up to which recovery will proceed
                                # (change requires restart)
```

Step 8:

Now start the cluster and it will recover the database till the specified time. Ensure to

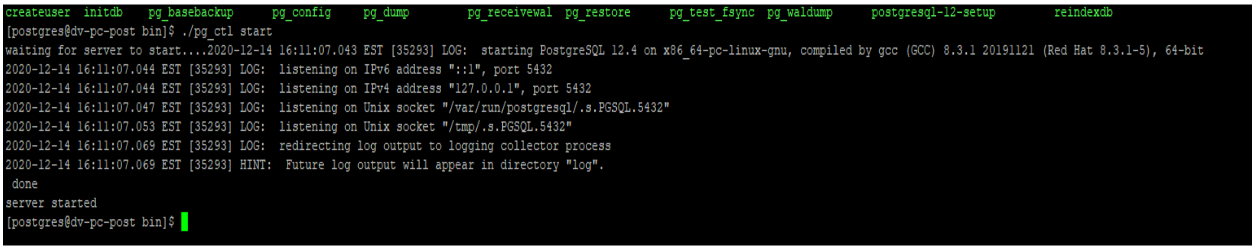
Change the permission on data folder to **700**



```
[postgres@dv-pc-post 12]$ ls -ltr
total 8
drwx------.  2 postgres postgres   40 Dec 14 16:24 backups
drwxr-xr-x.  2 postgres postgres 4096 Dec 14 16:32 archive_logs
drwxr-xr-x. 20 postgres postgres 4096 Dec 14 16:50 data
[postgres@dv-pc-post 12]$ chmod 700 data
[postgres@dv-pc-post 12]$
```

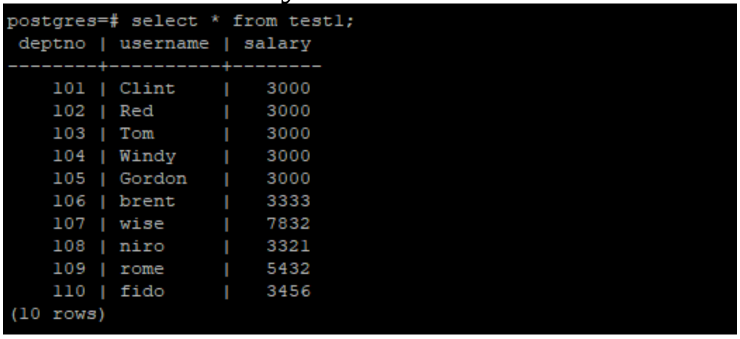
Step 9:

Start the cluster



```
createuser  initdb   pg_basebackup     pg_config     pg_dump         pg_receivewal  pg_restore      pg_test_fsync  pg_waldump      postgresql-12-setup         reindexdb
[postgres@dv-pc-post bin]$ ./pg_ctl start
waiting for server to start....2020-12-14 16:11:07.043 EST [35293] LOG:  starting PostgreSQL 12.4 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5), 64-bit
2020-12-14 16:11:07.044 EST [35293] LOG:  listening on IPv6 address "::1", port 5432
2020-12-14 16:11:07.044 EST [35293] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2020-12-14 16:11:07.047 EST [35293] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2020-12-14 16:11:07.053 EST [35293] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
2020-12-14 16:11:07.069 EST [35293] LOG:  redirecting log output to logging collector process
2020-12-14 16:11:07.069 EST [35293] HINT:  Future log output will appear in directory "log".
 done
server started
[postgres@dv-pc-post bin]$
```

Step 10:
Check how many rows are there in table test1.



```
postgres=# select * from test1;
 deptno | username | salary
--------+----------+--------
    101 | Clint    |   3000
    102 | Red      |   3000
    103 | Tom      |   3000
    104 | Windy    |   3000
    105 | Gordon   |   3000
    106 | brent    |   3333
    107 | wise     |   7832
    108 | niro     |   3321
    109 | rome     |   5432
    110 | fido     |   3456
(10 rows)
```

Hence we have successfully restored our database using Point in time recovery.

Maintenance and its options

- All databases requires some kind of maintenance tasks to be performed regularly to achieve optimum performance.
- Maintenance task are ongoing and repetitive which are ideally automated and schedule from cron scripts(linux) and task scheduler(windows).
- Postgresql provides the following maintenance option:
  - Updating Planner Statistics/Analyze
  - Vacuum
  - Routine Reindexing
  - Clustering

Updating Planner Statistics:

- PostgreSQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. whenever we executes a query it travels a path in order to get the results. The path defines that how fast and how effective the results can be given. If the stats about the table i.e. count, row count, size is not proper than optimizer can not generate the good plans and query execution may take the more time then usual. Because of the optimizer is not aware of how many rows and columns are there in table.

- These stats are **gathered by Analyze command**, which keeps the stats up- to-date about the current state of the table.
- Analyze command collects information about size, row count, average row size and row sampling information.
- Inaccurate or stale stats can mislead optimizer to choose plans which might degrade database performance.
- Tables with heavy update/delete need to be analyzed on a regular basis to ensure optimal performance is achieved.
- We can run Analyze command automatically by enabling autovaccum daemon or can run the analyze command manually. Command to enable , disable and status of autovacuum for any table.

```
 # alter table table_name set  (autovacuum_enabled=false) ; -- to disable
# alter table table_name set  (autovacuum_enabled=true) ; -- to enable ,

#show autovacuum; – from postgresql.conf.
```

Command to check last analyze and vacuums:

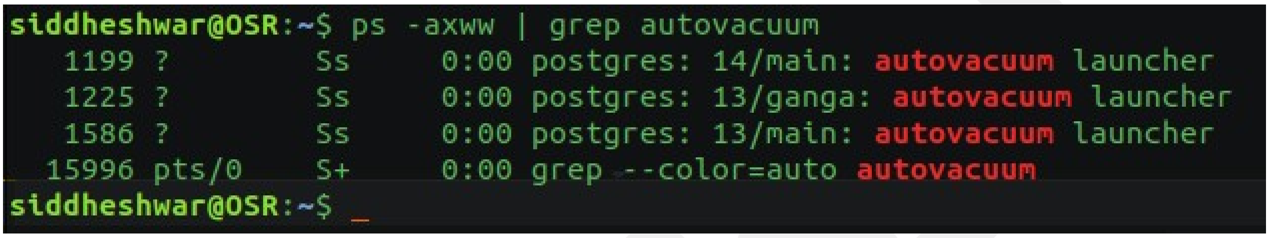
```
#select * from pg_stat_all_tables where schemaname='public';

#SELECT schemaname, relname, last_analyze FROM pg_stat_all_tables WHERE relname = 'a';
```

#SELECT schemaname, relname, last_vacuum, last_autovacuum, vacuum_count, autovacuum_count FROM pg_stat_user_tables;

Note: pg_stat_all_tables includes system catalog tables also. But pg_stat_user_tables is having only non system tables;

check autovacuum process on OS level:

#ps -axww | grep autovacuum



```
siddheshwar@OSR:~$ ps -axww | grep autovacuum
   1199 ?        Ss     0:00 postgres: 14/main: autovacuum launcher
   1225 ?        Ss     0:00 postgres: 13/ganga: autovacuum launcher
   1586 ?        Ss     0:00 postgres: 13/main: autovacuum launcher
  15996 pts/0    S+     0:00 grep --color=auto autovacuum
siddheshwar@OSR:~$ _
```

Check the logs:

#cat /var/log/postgresql/postgresql-13-main.log

Note: if the database was ever **shutdown uncleanly, all the stats are reset**. You could check pg_stat_database. Or they can be reset manually, either for the database or for individual tables.

#select datname,stats_reset from pg_stat_database;

Some Monitoring SQLs

```
-- ##Sessions and details
SELECT
   pid,
   datname,
   usename,
   application_name,
   client_addr,
   pg_catalog.to_char(backend_start, 'YYYY-MM-DD HH24:MI:SS TZ')
AS backend_start,
   state,
    wait_event_type || ': ' || wait_event AS wait_event,
    pg_catalog.pg_blocking_pids(pid) AS blocking_pids,
    query,
    pg_catalog.to_char(state_change, 'YYYY-MM-DD HH24:MI:SS TZ')
AS state_change,
    pg_catalog.to_char(query_start, 'YYYY-MM-DD HH24:MI:SS TZ') AS
query_start,
    backend_type,
    CASE WHEN state = 'active' THEN ROUND((extract(epoch from
now() - query_start) / 60)::numeric, 2) ELSE 0 END AS active_since
FROM
    pg_catalog.pg_stat_activity
ORDER BY pid;

SELECT relname, relpages FROM pg_class ORDER BY relpages DESC;

--#Database size
SELECT pg_size_pretty(pg_database_size('uidservices'));
SELECT datname as db_name,
pg_size_pretty(pg_database_size(datname)) as db_usage FROM
pg_database
order by 2 ;

create extension pageinspect;
--# get raw page values

SELECT t_xmin,
         t_xmax,
   tuple_data_split('adv_txn_log_012021'::regclass,
    t_data,
    t_infomask,
    t_infomask2, t_bits) FROM
heap_page_items(get_raw_page('adv_txn_log_012021', 0));

select txid_current();
```

```
-- #Live and dead tuples, if dead tuples>50% of live tuples table need to be
vacuumed.
select schemaname,
relname,
pg_size_pretty(pg_relation_size(schemaname|| '.' || relname)) as size,
n_live_tup,
n_dead_tup,
CASE WHEN n_live_tup > 0 THEN round((n_dead_tup::float /
n_live_tup::float)::numeric, 4) END AS dead_tup_ratio,
last_autovacuum,
last_autoanalyze
from pg_stat_user_tables
order by dead_tup_ratio desc NULLS LAST;

-- db level vacuum requirement if consumed_txid_pct>70% vacuum is
needed
SELECT
  datname,
  age(datfrozenxid) AS frozen_xid_age,
  ROUND(
    100 *(
      age(datfrozenxid)/ 2146483647.0 :: float
    )
  ) consumed_txid_pct,
  current_setting('autovacuum_freeze_max_age'):: int - age(datfrozenxid)
AS remaining_aggressive_vacuum
FROM
  pg_database
WHERE
  datname NOT IN (
    'template0', 'template1'
  );

--Connect to the database and run the following query to list tables that are
currently processed by the autovacuum daemon:
 SELECT p.pid,
      p.datname,
      p.query,
      p.backend_type,
      a.phase,
      a.heap_blks_scanned / a.heap_blks_total::float * 100 AS "% scanned",
      a.heap_blks_vacuumed / a.heap_blks_total::float * 100 AS "%
vacuumed",
      pg_size_pretty(pg_table_size(a.relid)) AS "table size",
pg_size_pretty(pg_indexes_size(a.relid)) AS "indexes size",
      pg_get_userbyid(c.relowner) AS owner
 FROM pg_stat_activity p
 JOIN pg_stat_progress_vacuum a ON a.pid = p.pid
 JOIN pg_class c ON c.oid = a.relid
WHERE p.query LIKE 'autovacuum%';

SELECT
  *,
  relid :: regclass,
  heap_blks_scanned / heap_blks_total :: float * 100 "% scanned",
  heap_blks_vacuumed / heap_blks_total :: float * 100 "% vacuumed"
FROM
  pg_stat_progress_vacuum;

-- top 10 tables transaction ID utilization  if consumed_txid_pct>70%
vacuum is needed

 SELECT c.relname AS table_name,
      age(c.relfrozenxid) AS frozen_xid_age,
      ROUND(100 * (age(c.relfrozenxid) / 2146483647)) AS
consumed_txid_pct,
      pg_size_pretty(pg_total_relation_size(c.oid)) AS table_size
 FROM pg_class c
 JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE c.relkind IN ('r', 't', 'm')
  AND n.nspname NOT IN ('pg_toast')
```

```
ORDER BY 2 DESC
LIMIT 10;

--## table extrasize
SELECT current_database(), schemaname, tblname, bs*tblpages AS
real_size,
  (tblpages-est_tblpages)*bs AS extra_size,
  CASE WHEN tblpages > 0 AND tblpages - est_tblpages > 0
    THEN 100 * (tblpages - est_tblpages)/tblpages::float
    ELSE 0
  END AS extra_pct, fillfactor,
  CASE WHEN tblpages - est_tblpages_ff > 0
    THEN (tblpages-est_tblpages_ff)*bs
    ELSE 0
  END AS bloat_size,
  CASE WHEN tblpages > 0 AND tblpages - est_tblpages_ff > 0
    THEN 100 * (tblpages - est_tblpages_ff)/tblpages::float
    ELSE 0
  END AS bloat_pct, is_na
  -- , tpl_hdr_size, tpl_data_size, (pst).free_percent +
(pst).dead_tuple_percent AS real_frag -- (DEBUG INFO)
FROM (
  SELECT ceil( reltuples / ( (bs-page_hdr)/tpl_size ) ) + ceil( toasttuples / 4
) AS est_tblpages,
    ceil( reltuples / ( (bs-page_hdr)*fillfactor/(tpl_size*100) ) ) +
ceil( toasttuples / 4 ) AS est_tblpages_ff,
    tblpages, fillfactor, bs, tblid, schemaname, tblname, heappages,
toastpages, is_na
    -- , tpl_hdr_size, tpl_data_size, pgstattuple(tblid) AS pst -- (DEBUG
INFO)
  FROM (
    SELECT
      ( 4 + tpl_hdr_size + tpl_data_size + (2*ma)
        - CASE WHEN tpl_hdr_size%ma = 0 THEN ma ELSE tpl_hdr_size
%ma END
```

```
        - CASE WHEN ceil(tpl_data_size)::int%ma = 0 THEN ma ELSE
ceil(tpl_data_size)::int%ma END
      ) AS tpl_size, bs - page_hdr AS size_per_block, (heappages +
toastpages) AS tblpages, heappages,
      toastpages, reltuples, toasttuples, bs, page_hdr, tblid, schemaname,
tblname, fillfactor, is_na
      -- , tpl_hdr_size, tpl_data_size
    FROM (
      SELECT
        tbl.oid AS tblid, ns.nspname AS schemaname, tbl.relname AS
tblname, tbl.reltuples,
        tbl.relpages AS heappages, coalesce(toast.relpages, 0) AS toastpages,
        coalesce(toast.reltuples, 0) AS toasttuples,
        coalesce(substring(
          array_to_string(tbl.reloptions, ' ')
          FROM 'fillfactor=([0-9]+)')::smallint, 100) AS fillfactor,
        current_setting('block_size')::numeric AS bs,
        CASE WHEN version()~'mingw32' OR version()~'64-bit|x86_64|
ppc64|ia64|amd64' THEN 8 ELSE 4 END AS ma,
        24 AS page_hdr,
        23 + CASE WHEN MAX(coalesce(s.null_frac,0)) > 0 THEN ( 7 +
count(s.attname) ) / 8 ELSE 0::int END
           + CASE WHEN bool_or(att.attname = 'oid' and att.attnum < 0)
THEN 4 ELSE 0 END AS tpl_hdr_size,
        sum( (1-coalesce(s.null_frac, 0)) * coalesce(s.avg_width, 0) ) AS
tpl_data_size,
        bool_or(att.atttypid = 'pg_catalog.name'::regtype)
          OR sum(CASE WHEN att.attnum > 0 THEN 1 ELSE 0 END) <>
count(s.attname) AS is_na
      FROM pg_attribute AS att
        JOIN pg_class AS tbl ON att.attrelid = tbl.oid
        JOIN pg_namespace AS ns ON ns.oid = tbl.relnamespace
        LEFT JOIN pg_stats AS s ON s.schemaname=ns.nspname
          AND s.tablename = tbl.relname AND s.inherited=false AND
s.attname=att.attname
        LEFT JOIN pg_class AS toast ON tbl.reltoastrelid = toast.oid
      WHERE NOT att.attisdropped
        AND tbl.relkind in ('r','m')
      GROUP BY 1,2,3,4,5,6,7,8,9,10
      ORDER BY 2,3
    ) AS s
  ) AS s2
) AS s3
-- WHERE NOT is_na
--   AND tblpages*((pst).free_percent +
(pst).dead_tuple_percent)::float4/100 >= 1
ORDER BY schemaname, tblname;


-- #index usage
SELECT
    t.schemaname,
    t.tablename,
    c.reltuples::bigint                            AS num_rows,
    pg_size_pretty(pg_relation_size(c.oid))        AS table_size,
    psai.indexrelname                              AS index_name,
    pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size,
    CASE WHEN i.indisunique THEN 'Y' ELSE 'N' END  AS "unique",
    psai.idx_scan                                  AS number_of_scans,
    psai.idx_tup_read                              AS tuples_read,
    psai.idx_tup_fetch                             AS tuples_fetched
FROM
    pg_tables t
    LEFT JOIN pg_class c ON t.tablename = c.relname
    LEFT JOIN pg_index i ON c.oid = i.indrelid
    LEFT JOIN pg_stat_all_indexes psai ON i.indexrelid = psai.indexrelid
WHERE
    t.schemaname NOT IN ('pg_catalog', 'information_schema')
ORDER BY 1, 2;

--#txid before wraparround
SELECT
```

```
      oid::regclass::text AS table,
      age(relfrozenxid) AS xid_age,
      mxid_age(relminmxid) AS mxid_age,
      least(
(SELECT setting::int
          FROM    pg_settings
          WHERE   name = 'autovacuum_freeze_max_age') -
age(relfrozenxid),
(SELECT setting::int
          FROM    pg_settings
          WHERE   name = 'autovacuum_multixact_freeze_max_age') -
mxid_age(relminmxid)
) AS tx_before_wraparound_vacuum,
pg_size_pretty(pg_total_relation_size(oid)) AS size,
pg_stat_get_last_autovacuum_time(oid) AS last_autovacuum
FROM    pg_class
WHERE   relfrozenxid != 0
AND oid > 1
ORDER BY tx_before_wraparound_vacuum;


--## index bloat calculating superuser
SELECT current_database(), nspname AS schemaname, tblname,
idxname, bs*(relpages)::bigint AS real_size,
  bs*(relpages-est_pages)::bigint AS extra_size,
  100 * (relpages-est_pages)::float / relpages AS extra_pct,
  fillfactor,
  CASE WHEN relpages > est_pages_ff
    THEN bs*(relpages-est_pages_ff)
    ELSE 0
  END AS bloat_size,
  100 * (relpages-est_pages_ff)::float / relpages AS bloat_pct,
  is_na
  -- , 100-(pst).avg_leaf_density AS pst_avg_bloat, est_pages,
index_tuple_hdr_bm, maxalign, pagehdr, nulldatawidth, nulldatahdrwidth,
reltuples, relpages -- (DEBUG INFO)
```

```
FROM (
  SELECT coalesce(1 +

ceil(reltuples/floor((bs-pageopqdata-pagehdr)/(4+nulldatahdrwidth)::float)
), 0 -- ItemIdData size + computed avg size of a tuple (nulldatahdrwidth)
      ) AS est_pages,
      coalesce(1 +

ceil(reltuples/floor((bs-pageopqdata-pagehdr)*fillfactor/(100*(4+nulldatah
drwidth)::float))), 0
      ) AS est_pages_ff,
      bs, nspname, tblname, idxname, relpages, fillfactor, is_na
      -- , pgstatindex(idxoid) AS pst, index_tuple_hdr_bm, maxalign,
pagehdr, nulldatawidth, nulldatahdrwidth, reltuples -- (DEBUG INFO)
  FROM (
      SELECT maxalign, bs, nspname, tblname, idxname, reltuples,
relpages, idxoid, fillfactor,
            ( index_tuple_hdr_bm +
                maxalign - CASE -- Add padding to the index tuple header to
align on MAXALIGN
                  WHEN index_tuple_hdr_bm%maxalign = 0 THEN maxalign
                  ELSE index_tuple_hdr_bm%maxalign
                END
              + nulldatawidth + maxalign - CASE -- Add padding to the data to
align on MAXALIGN
                  WHEN nulldatawidth = 0 THEN 0
                  WHEN nulldatawidth::integer%maxalign = 0 THEN maxalign
                  ELSE nulldatawidth::integer%maxalign
                END
            )::numeric AS nulldatahdrwidth, pagehdr, pageopqdata, is_na
            -- , index_tuple_hdr_bm, nulldatawidth -- (DEBUG INFO)
      FROM (
          SELECT n.nspname, ct.relname AS tblname, i.idxname, i.reltuples,
i.relpages,
              i.idxoid, i.fillfactor, current_setting('block_size')::numeric AS bs,
              CASE -- MAXALIGN: 4 on 32bits, 8 on 64bits (and mingw32 ?)
```

```
               WHEN version() ~ 'mingw32' OR version() ~ '64-bit|x86_64|
ppc64|ia64|amd64' THEN 8
               ELSE 4
            END AS maxalign,
            /* per page header, fixed size: 20 for 7.X, 24 for others */
            24 AS pagehdr,
            /* per page btree opaque data */
            16 AS pageopqdata,
            /* per tuple header: add IndexAttributeBitMapData if some cols
are null-able */
            CASE WHEN max(coalesce(s.stanullfrac,0)) = 0
               THEN 2 -- IndexTupleData size
               ELSE 2 + (( 32 + 8 - 1 ) / 8) -- IndexTupleData size +
IndexAttributeBitMapData size ( max num filed per index + 8 - 1 /8)
            END AS index_tuple_hdr_bm,
            /* data len: we remove null values save space using it fractionnal
part from stats */
            sum( (1-coalesce(s.stanullfrac, 0)) * coalesce(s.stawidth, 1024))
AS nulldatawidth,
            max( CASE WHEN a.atttypid = 'pg_catalog.name'::regtype
THEN 1 ELSE 0 END ) > 0 AS is_na
         FROM (
            SELECT idxname, reltuples, relpages, tbloid, idxoid, fillfactor,
               CASE WHEN indkey[i]=0 THEN idxoid ELSE tbloid END AS
att_rel,
               CASE WHEN indkey[i]=0 THEN i ELSE indkey[i] END AS
att_pos
            FROM (
               SELECT idxname, reltuples, relpages, tbloid, idxoid, fillfactor,
indkey, generate_series(1,indnatts) AS i
               FROM (
                  SELECT ci.relname AS idxname, ci.reltuples, ci.relpages,
i.indrelid AS tbloid,
                     i.indexrelid AS idxoid,
                     coalesce(substring(
                        array_to_string(ci.reloptions, ' ')
```

```
                          from 'fillfactor=([0-9]+)')::smallint, 90) AS fillfactor,
                        i.indnatts,
                        string_to_array(textin(int2vectorout(i.indkey)),' ')::int[] AS
indkey
                      FROM pg_index i
                      JOIN pg_class ci ON ci.oid=i.indexrelid
                      WHERE ci.relam=(SELECT oid FROM pg_am WHERE
amname = 'btree')
                       AND ci.relpages > 0
                    ) AS idx_data
                  ) AS idx_data_cross
              ) i
              JOIN pg_attribute a ON a.attrelid = i.att_rel
                         AND a.attnum = i.att_pos
              JOIN pg_statistic s ON s.starelid = i.att_rel
                         AND s.staattnum = i.att_pos
              JOIN pg_class ct ON ct.oid = i.tbloid
              JOIN pg_namespace n ON ct.relnamespace = n.oid
              GROUP BY 1,2,3,4,5,6,7,8,9,10
          ) AS rows_data_stats
     ) AS rows_hdr_pdg_stats
) AS relation_stats
ORDER BY nspname, tblname, idxname;
```

Explain plan and Query Execution Cost:

Explain plans, statement displays execution plan chosen by the optimizer, for select, update, insert and delete statement, which means whenever we execute a query, it goes through a certain path in order to create the output. We want to know how much time or how much cost the query is going to take and what path the query is going to traverse in order to get the output. So in that scenario, I can use option call explain in order to see what the query is doing.

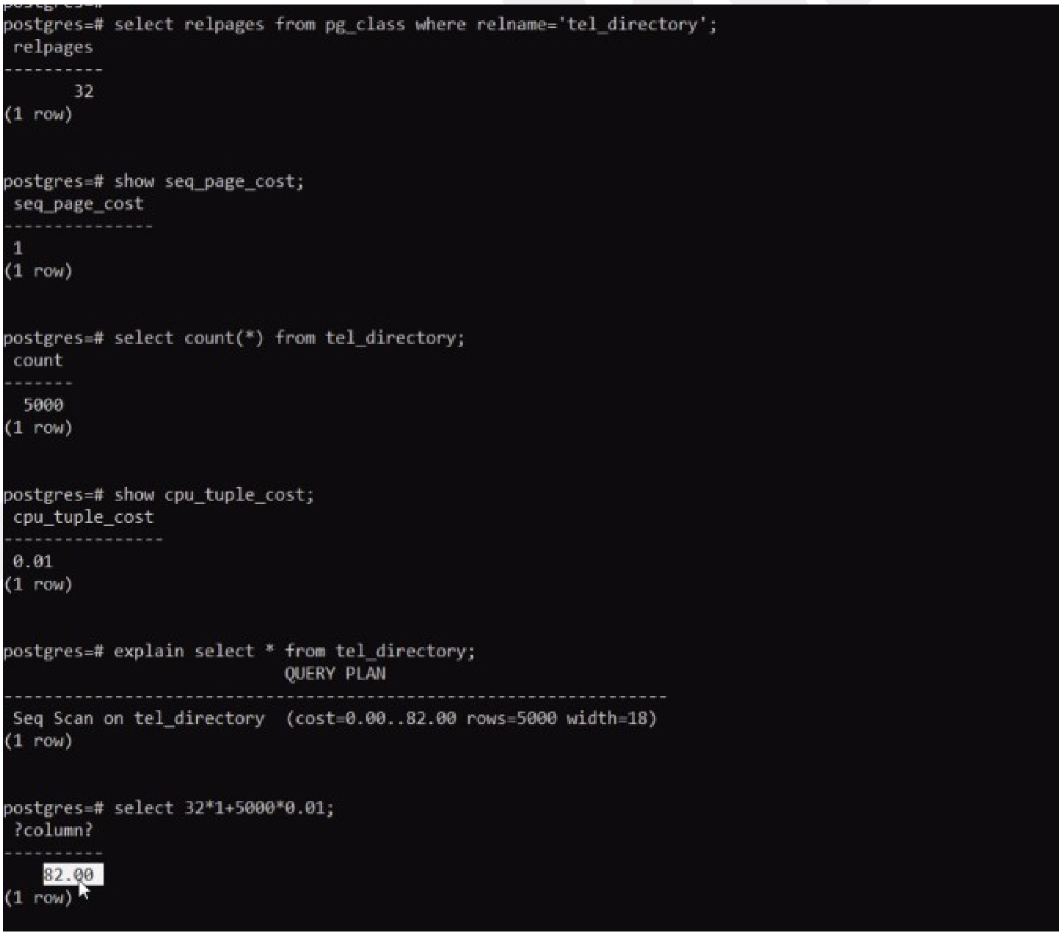
EXPLAIN PLAN statement displays execution plans chosen by the optimizer for SELECT, UPDATE, INSERT, and DELETE statements.

Example : explain select * from <tablename>;

```
              Aggregate  (cost=1.12..1.14 rows=1 width=8)
           ->  Seq Scan on august  (cost=0.00..1.10 rows=10 width=4)
               (2 rows)
```

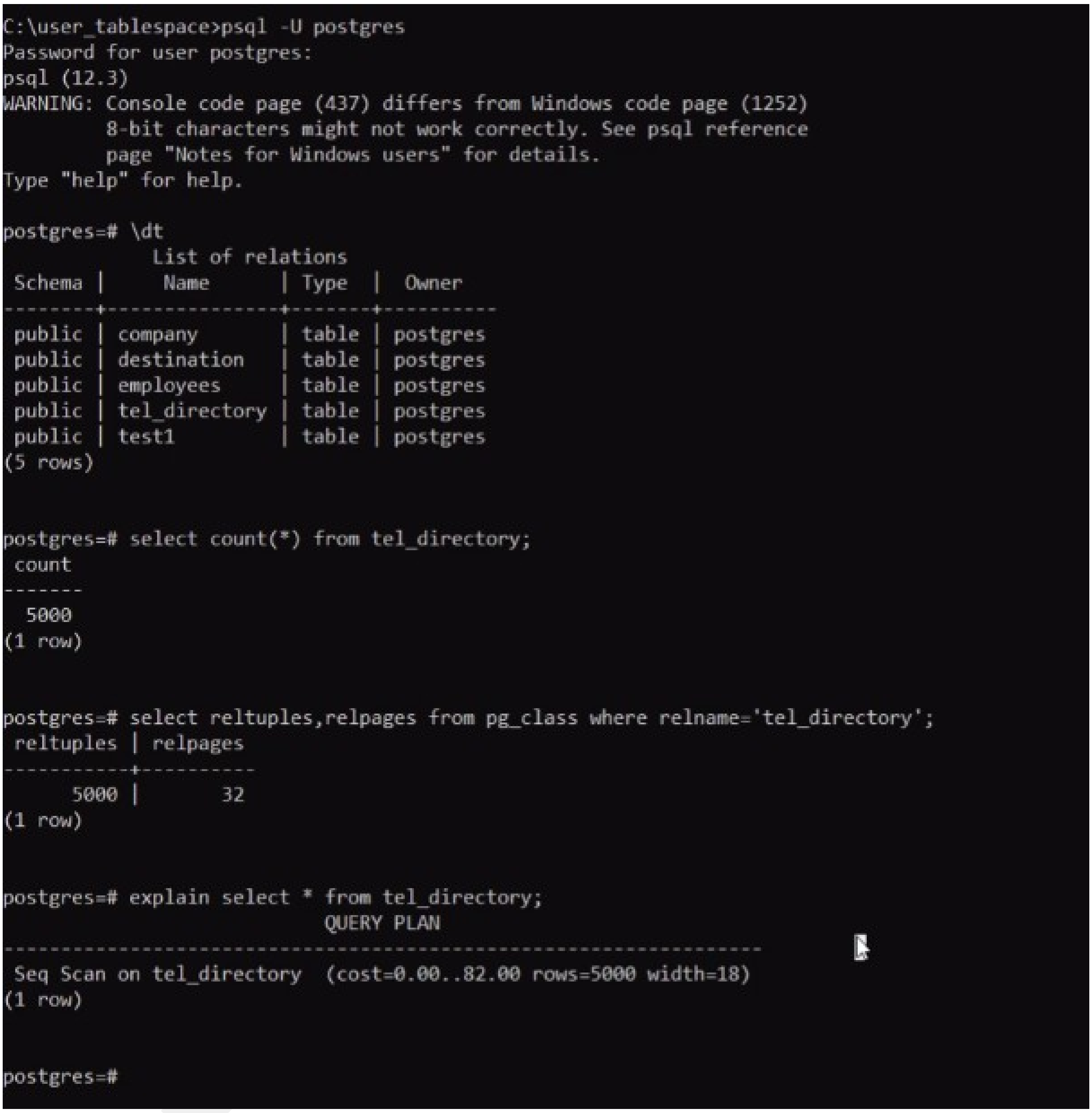
Cost of Query execution

Cost = number of pages * seq_page_cost + number of rows* cpu_tuple_cost



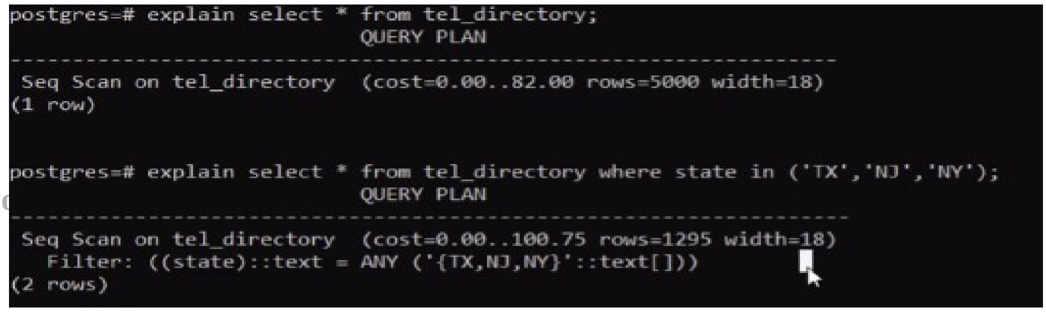
```
postgres=# select relpages from pg_class where relname='tel_directory';
 relpages
----------
       32
(1 row)


postgres=# show seq_page_cost;
 seq_page_cost
---------------
 1
(1 row)


postgres=# select count(*) from tel_directory;
 count
-------
  5000
(1 row)


postgres=# show cpu_tuple_cost;
 cpu_tuple_cost
----------------
 0.01
(1 row)


postgres=# explain select * from tel_directory;
                            QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on tel_directory  (cost=0.00..82.00 rows=5000 width=18)
(1 row)


postgres=# select 32*1+5000*0.01;
 ?column?
----------
    82.00
(1 row)
```

Example explain plan :



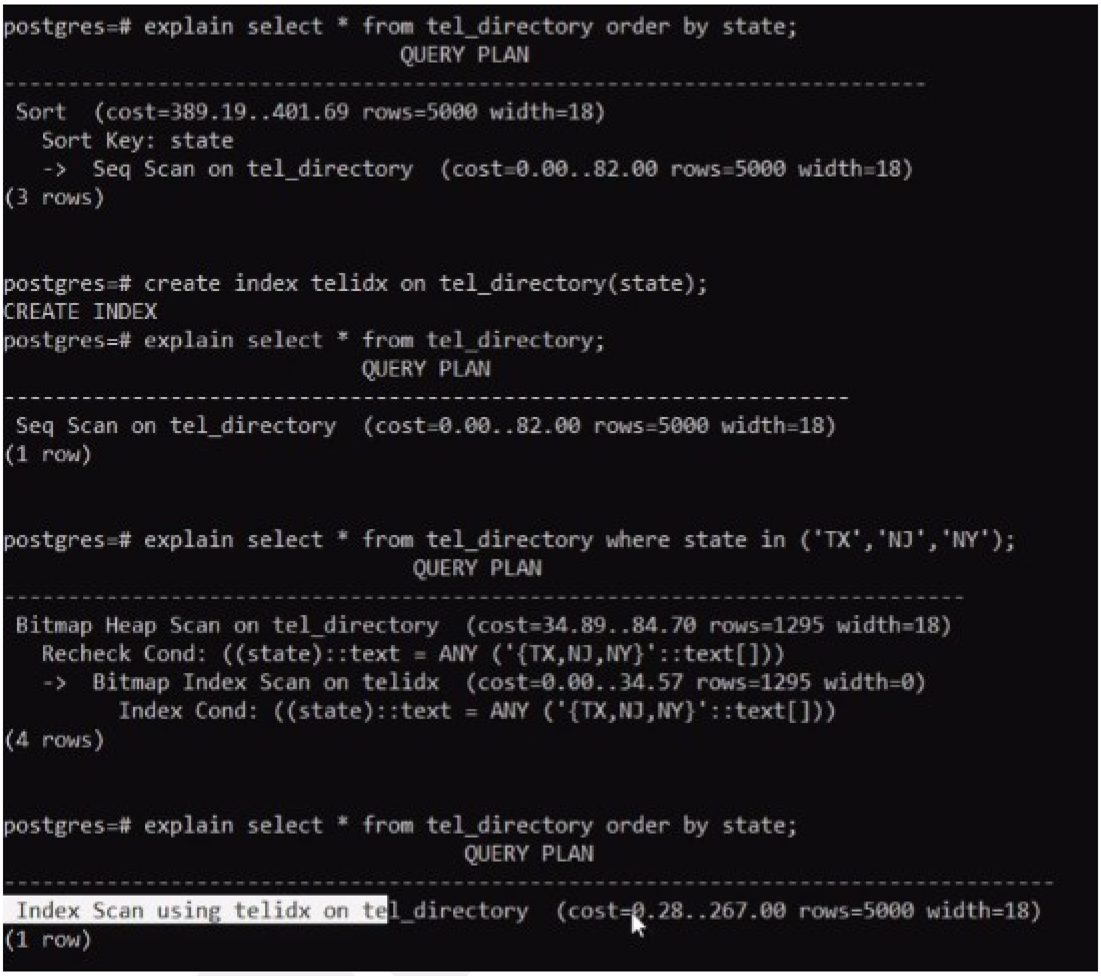
```
C:\user_tablespace>psql -U postgres
Password for user postgres:
psql (12.3)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | company       | table | postgres
 public | destination   | table | postgres
 public | employees     | table | postgres
 public | tel_directory | table | postgres
 public | test1         | table | postgres
(5 rows)


postgres=# select count(*) from tel_directory;
 count
-------
  5000
(1 row)


postgres=# select reltuples,relpages from pg_class where relname='tel_directory';
 reltuples | relpages
-----------+----------
      5000 |       32
(1 row)


postgres=# explain select * from tel_directory;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on tel_directory  (cost=0.00..82.00 rows=5000 width=18)
(1 row)


postgres=#
```

see above to display **all records** from **table tel_directory** it will be going thru **sequential scan** and **take 0.00 initial cost and 82.00 actual cost**. Below is the example with filter applied.



```
postgres=# explain select * from tel_directory;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on tel_directory  (cost=0.00..82.00 rows=5000 width=18)
(1 row)


postgres=# explain select * from tel_directory where state in ('TX','NJ','NY');
                           QUERY PLAN
------------------------------------------------------------------
 Seq Scan on tel_directory  (cost=0.00..100.75 rows=1295 width=18)
   Filter: ((state)::text = ANY ('{TX,NJ,NY}'::text[]))
(2 rows)
```

Explain With index:



```
postgres=# explain select * from tel_directory order by state;
                              QUERY PLAN
------------------------------------------------------------------------
 Sort  (cost=389.19..401.69 rows=5000 width=18)
   Sort Key: state
   ->  Seq Scan on tel_directory  (cost=0.00..82.00 rows=5000 width=18)
(3 rows)


postgres=# create index telidx on tel_directory(state);
CREATE INDEX
postgres=# explain select * from tel_directory;
                           QUERY PLAN
------------------------------------------------------------------
 Seq Scan on tel_directory  (cost=0.00..82.00 rows=5000 width=18)
(1 row)


postgres=# explain select * from tel_directory where state in ('TX','NJ','NY');
                               QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on tel_directory  (cost=34.89..84.70 rows=1295 width=18)
   Recheck Cond: ((state)::text = ANY ('{TX,NJ,NY}'::text[]))
   ->  Bitmap Index Scan on telidx  (cost=0.00..34.57 rows=1295 width=0)
         Index Cond: ((state)::text = ANY ('{TX,NJ,NY}'::text[]))
(4 rows)


postgres=# explain select * from tel_directory order by state;
                                   QUERY PLAN
----------------------------------------------------------------------------------
 Index Scan using telidx on tel_directory  (cost=0.28..267.00 rows=5000 width=18)
(1 row)
```

as you see in above image after creating index the cost is reduced and path is also change. So by this we can check whether index is using effectively or not.

Data Fragmentation:

- Fragmentation is often called bloat in PostgreSQL.
- PostgreSQL in line with Multiversion concurrency control (MVCC) does not UPDATE in place or DELETE a row directly from the disk.
- These rows are marked as old versions.

- As the old version become obsolete and keep piling up. This causes fragmentation and bloating in the table.
- Tables or Indexes become bigger than their actual size.

How to detect bloat:

For the routine check we can use the **Extension pgstattuple** module.

#Create extension pgstattuple; – - Run this command to install extension.

The following sql shows table txn_log is heavily fragmented with 98.71 % free space, and tuple percent is only 9%.

```
#postgres=>   SELECT * FROM pgstattuple('txn_log');
```

```
   table_len    | tuple_count | tuple_len  | tuple_percent | dead_tuple_count | dead_tuple_Olen | dead_tuple_percent |
free_space       | free_percent
-------------+-------------+-----------+---------------+------------------+----------------+--------------------+-------------
+--------------
 10459709440 |      58988   |   9583432 |          0.09        |            7876         |       1430999    |             0.01          |
10324502496 |       98.71

(1 row)
```

```
postgres=> select count(*) from txn_log;
count
-------
58988
(1 row)
```

pgstattuple output columns:

Column	Type	Description
table_len	bigint	Physical relation length in bytes
tuple_count	bigint	Number of live tuples
tuple_len	bigint	Total length of live tuples in bytes
tuple_percent	float8	Percentage of live

Column	Type	Description
		tuples
dead_tuple_count	bigint	Number of dead tuples
dead_tuple_len	bigint	Total length of dead tuples in bytes
dead_tuple_percent	float8	Percentage of dead tuples
free_space	bigint	Total free space in bytes
free_percent	float8	Percentage of free space

USER pgstateindex to get index bloat.

#SELECT * FROM pgstatindex('idx_customers');

```
 version | tree_level | index_size | root_block_no | internal_pages | leaf_pages | empty_pages | deleted_pages |
avg_leaf_density | leaf_fragmentation
---------+------------+------------+---------------+----------------+------------+-------------+---------------+------------------
+--------------------
       3 |          3 | 1030766592 |         25480 |            835 |     101582 |           0 |         23408 |            77.48 |              67.07
(1 row)
```

pgstatindex output columns:

Column	Type	Description
version	integer	Btree version number
tree_level	integer	Tree level of the root page
index_size	bigint	Total number of pages in index
root_block_no	bigint	Location of root block

Column	Type	Description
internal_pages	bigint	Number of “internal” (upper-level) pages
leaf_pages	bigint	Number of leaf pages
empty_pages	bigint	Number of empty pages
deleted_pages	bigint	Number of deleted pages
avg_leaf_density	float8	Average density of leaf pages
leaf_fragmentation	float8	Leaf page fragmentation

Query with simplified output :

```
SELECT pg_size_pretty(pg_relation_size('txn_req_log_022022')) as
table_size, pg_size_pretty
(pg_relation_size('activity_req_log_pkey_txn_req_log_022022')) as
index_size, (pgstattuple ('txn_req_log_022022')).dead_tuple_percent;
```

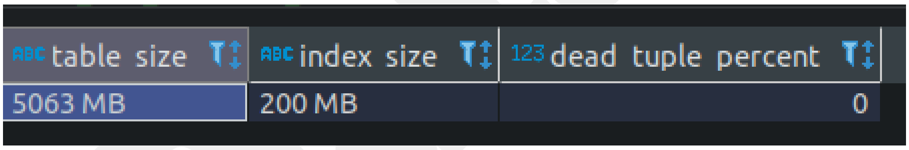


table size	index size	dead tuple percent
5063 MB	200 MB	0

Query to get tuples ratio:

```
select schemaname,

relname,

pg_size_pretty(pg_relation_size(schemaname|| '.' || relname)) as size,

n_live_tup,

n_dead_tup,

CASE WHEN n_live_tup > 0 THEN round((n_dead_tup::float /
n_live_tup::float)::numeric, 4) END AS dead_tup_ratio,

last_autovacuum,

last_autoanalyze

from pg_stat_user_tables

order by dead_tup_ratio desc NULLS LAST;
```

Vacuum:

Whenever we update or delete a record. Those record doesn't get deleted from the disk, they are marked as old version, and this does set in the same table occupying space. PostgreSQL maintains both the past image and the latest image of a row in its own Table. It means, UNDO is maintained within each table. And this is done through versioning. Now, we may get a hint that, every row of PostgreSQL table has a version number. And that is absolutely correct. In order to understand how these versions are maintained within each table, you should understand the hidden columns of a table (especially xmin) in PostgreSQL.

When you describe a table, you would only see the columns you have added. However, if you look at all the columns of the table in pg_attribute, you should see several hidden columns.

```
# SELECT attname, format_type (atttypid, atttypmod) FROM pg_attribute
WHERE  attrelid::regclass::text='scott.employee' ORDER BY attnum;

attname  |      format_type
----------+-----------------------
 tableoid  | oid
 cmax       | cid
 xmax       | xid
 cmin       | cid
 xmin       | xid
 ctid       | tid
 emp_id     | integer
 emp_name       | character varying(100)
 dept_id  | integer
(9 rows)
```

tableoid : Contains the Object ID of the table that contains this row. Used by queries that select from inheritance hierarchies.

xmin : The transaction ID(xid) of the inserting transaction for this row version. Upon update, a new row version is inserted. Let's see the following log to understand the xmin more.

```
# select txid_current();
 txid_current
--------------
         646  – Remember this
(1 row)

dataxorg =# INSERT into scott.employee VALUES (9,'avi',9);
INSERT 0 1

# select xmin,xmax,cmin,cmax,* from scott.employee where emp_id = 9;
 xmin | xmax | cmin | cmax | emp_id | emp_name | dept_id
------+------+------+------+--------+----------+---------
  647 |    0 |    0 |    0 |      9 | avi      |       9
(1 row)
```

As you see in the above log, the transaction ID was 646 for the command => select txid_current(). Thus, the immediate INSERT statement got a transaction ID 647. Hence, the record was assigned an xmin of 647. This means, no running transaction ID that has started before the ID 647, can see this row. In other words, already running transactions with txid less than 647 cannot see the row inserted by txid 647.

With the above example, you should now understand that every tuple has an xmin that is assigned the txid that inserted it.

xmax : This values is 0 if it was not a deleted row version. Before the DELETE is committed, the xmax of the row version changes to the ID of the transaction that has issued the DELETE. Let’s observe the following log to understand that better.

Before the Delete

```
------------------
# select xmin,xmax,cmin,cmax,* from scott.employee where emp_id = 10;

 xmin | xmax | cmin | cmax | emp_id | emp_name | dept_id
------+------+------+------+--------+----------+---------
  649 |    0 |    0 |    0 |     10 | avi      |      10
```

After the Delete

```
------------------
# select xmin,xmax,cmin,cmax,* from scott.employee where emp_id = 10;
 xmin | xmax | cmin | cmax | emp_id | emp_name | dept_id
------+------+------+------+--------+----------+---------
  649 |  655 |    0 |    0 |     10 | avi      |      10
(1 row)
```

As you see in the above logs, the **xmax** value changed to the transaction ID that has issued the delete. If you have issued a ROLLBACK, or if the transaction got aborted, xmax remains at the transaction ID that tried to DELETE it (which is 655) in this case.

when you check the count after DELETE, you would not see the records that have been DELETED. **To see any row versions that exist in the table but are not visible, we have an extension called pageinspect**. The pageinspect module provides functions that allow you to inspect the contents of database pages at a low level, which is useful for debugging purposes. Let’s create this extension to see the older row versions those have been deleted.

```
# SELECT t_xmin, t_xmax, tuple_data_split('scott.employee'::regclass,
t_data, t_infomask, t_infomask2, t_bits) FROM
heap_page_items(get_raw_page('scott.employee', 0));
 t_xmin | t_xmax |            tuple_data_split
--------+--------+-------------------------------------------
    668 |      0 | {"\\x01000000","\\x09617669","\\x01000000"}
    668 |      0 | {"\\x02000000","\\x09617669","\\x01000000"}
    668 |      0 | {"\\x03000000","\\x09617669","\\x01000000"}
    668 |      0 | {"\\x04000000","\\x09617669","\\x01000000"}
    668 |      0 | {"\\x05000000","\\x09617669","\\x01000000"}
    668 |    669 | {"\\x06000000","\\x09617669","\\x01000000"}
    668 |    669 | {"\\x07000000","\\x09617669","\\x01000000"}
    668 |    669 | {"\\x08000000","\\x09617669","\\x01000000"}
    668 |    669 | {"\\x09000000","\\x09617669","\\x01000000"}
    668 |    669 | {"\\x0a000000","\\x09617669","\\x01000000"}
(10 rows)
```

we could still see 10 records in the table even after deleting 5 records from it. Also, you can observe here that t_xmax is set to the transaction ID that has deleted them. These deleted records are retained in the same table to serve any of the older transactions that are still accessing them.

An UPDATE in PostgreSQL would perform an insert and a delete. Hence, all the records being UPDATED have been deleted and inserted back with the new value. Deleted records have non-zero t_xmax value.
Records for which you see a non-zero value for t_xmax may be required by the previous transactions to ensure consistency based on appropriate isolation levels.

cmax : The command identifier within the deleting transaction or zero. (As per the documentation). However, both cmin and cmax are always the same as per the PostgreSQL source code.

cmin : The command identifier within the inserting transaction. You could see the cmin of the 3 insert statements starting with 0.

Every such record that has been deleted but is still taking some space is called a dead tuple. Once there is no dependency on those dead tuples with the already running transactions, the dead tuples are no longer needed. Thus, PostgreSQL runs VACUUM on such Tables. VACUUM reclaims the storage occupied by these dead tuples. The space occupied by these dead tuples may be referred to as **Bloat**. VACUUM scans the pages for dead tuples and marks them to the free space map (FSM). Each relation apart from hash indexes has an FSM stored in a separate file called <relation_oid>_fsm.

ctid : The physical location of the row version within its table. Note that although the ctid can be used to locate the row version very quickly, a row's ctid will change if it is updated or moved by VACUUM FULL. Therefore ctid is useless as a long-term row identifier. A primary key should be used to identify logical rows.

There is a utility or an option called vacuum that is provided in postgreSQL.

- VACUUM reclaims storage occupied by dead tuples\rows.
- Tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done.
- There are two type of vacuum in PostgreSQL, Vacuum and Vacuum full.

  Vacuum:

- Plain VACUUM (without FULL) simply reclaims space and makes it available for re-use.
- Deleted and obsolete tuples are removed when vacuum is done.

- Extra free space is not returned to the operating system it's just kept available for re-use in the object itself.
- No exclusive lock on table.
- Frequently updated tables are good candidates for vacuuming.
- VACUUM does an additional task. All the rows that are inserted and successfully committed in the past are marked as frozen, which indicates that they are visible to all the current and future transactions.
- VACUUM does not usually reclaim the space to filesystem unless the dead tuples are beyond the high water mark. if there are pages(Blank pages at the end) with no more live tuples after the high water mark, the subsequent pages can be flushed away to the disk by VACUUM.

Vacuum Full

- VACUUM FULL rewrites the entire contents of the table into a new disk file with no extra space.
- Unused space to be returned to the operating system.
- Compacts tables and reclaims more space.
- Takes much longer than regular vacuum and **places exclusive lock the tables**. We should do this activity in non business hours.
- **full vacuum takes extra disk space**, since it writes a new copy of the table and doesn't release the old copy until the operation is complete.

- **vacuumdb** is a utility for cleaning a PostgreSQL database. vacuumdb will also **generate internal statistics** used by the PostgreSQL query optimizer.
- Syntax : vacuumdb -f –dbname=mydb --verbose

vacuum cannot run in a transaction and cannot run in a function or in a procedure. Generate the vacuum statements with the list of tables to be processed with psql CLI and \gexec internal command to run them.

```
select format('vacuum %s.%s', schemaname, tablename)

from (select schemaname, tablename

    from pg_tables where tablename like '%102020%'

) lt;

\gexec
```

Determine TransactionID Utilization

```
SELECT
  datname,
  age(datfrozenxid) AS frozen_xid_age,
  ROUND(
    100 *(
      age(datfrozenxid)/ 2146483647.0 :: float
    )
  ) consumed_txid_pct,
  current_setting('autovacuum_freeze_max_age'):: int - age(datfrozenxid)
AS remaining_aggressive_vacuum
FROM
  pg_database
WHERE
  datname NOT IN (
    'cloudsqladmin', 'template0', 'template1'
  );
```

A value of >80% in the consumed_txid_pct column indicates that tables in that database are in need of vacuuming to recycle transaction IDs.

Run the following query to list tables that are currently processed by the autovacuum daemon:

```
SELECT p.pid,
      p.datname,
      p.query,
      p.backend_type,
      a.phase,
      a.heap_blks_scanned / a.heap_blks_total::float * 100 AS "% scanned",
      a.heap_blks_vacuumed / a.heap_blks_total::float * 100 AS "%
vacuumed",
      pg_size_pretty(pg_table_size(a.relid)) AS "table size",
pg_size_pretty(pg_indexes_size(a.relid)) AS "indexes size",
      pg_get_userbyid(c.relowner) AS owner
 FROM pg_stat_activity p
 JOIN pg_stat_progress_vacuum a ON a.pid = p.pid
 JOIN pg_class c ON c.oid = a.relid
WHERE p.query LIKE 'autovacuum%';
```

Output:

```
-[ RECORD 2 ]+------------------------------------------
pid          | 286964
datname      | test_db
query        | autovacuum: VACUUM public.my_table
backend_type | autovacuum worker
phase        | vacuuming indexes
% scanned    | 100
% vacuumed   | 0
table size   | XX
indexes size | XX
```

```
owner        | test_user
-[ RECORD 3 ]+------------------------------------------
pid          | 271948
datname      | test_db
query        | autovacuum: VACUUM ANALYZE public.sample1
backend_type | autovacuum worker
```

Each record in the output corresponds to one autovacuum operation running in the database. Review the output and identify any records where the vacuum is in the **“vacuuming indexes”** phase as shown in the phase field. This indicates an operation that can potentially be sped up by canceling the autovacuum and performing a manual vacuum instead.

If there are multiple tables eligible for manual vacuuming, focus on the largest ones first. The larger the table, the longer the autovacuum process can take. Therefore, applying this procedure on the largest tables first can produce the biggest gains.

Vacuum then :

```
SELECT
  pg_cancel_backend(286964)
FROM
  pg_stat_progress_vacuum
WHERE
  relid = 'public.pgbench_accounts' :: regclass;
VACUUM (
  TRUNCATE off, INDEX_CLEANUP false,
  VERBOSE, FREEZE
) public.my_table;
```

Monitor the vacuum Process

```
SELECT
  *,
  relid :: regclass,
  heap_blks_scanned / heap_blks_total :: float * 100 "% scanned",
  heap_blks_vacuumed / heap_blks_total :: float * 100 "% vacuumed"
FROM
  pg_stat_progress_vacuum;
```

After the vacuum completes, you can optionally reindex the table. Our optimized VACUUM command contained the INDEX_CLEANUP false clause, which skips the index optimization stage. Bypassing index optimization doesn’t cause any immediate issues, but if you frequently vacuum the same tables with INDEX_CLEANUP false, it can lead to index bloat in the long term. You may want to REINDEX your table periodically if index bloat becomes a problem.

```
REINDEX (VERBOSE) TABLE CONCURRENTLY public.my_table;
```

(To recreate all the indexes of a table, you use the TABLE keyword and specify the name of the table:)

Recheck the consumed_tcid_pct .

You can obtain TXID information at table level to identify tables that still need vacuuming. SQL for top 10 tables ordered by transaction ID utilization.

```
SELECT c.relname AS table_name,
      age(c.relfrozenxid) AS frozen_xid_age,
      ROUND(100 * (age(c.relfrozenxid) / 2146483647)) AS
consumed_txid_pct,
      pg_size_pretty(pg_total_relation_size(c.oid)) AS table_size
 FROM pg_class c
 JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE c.relkind IN ('r', 't', 'm')
  AND n.nspname NOT IN ('pg_toast')
ORDER BY 2 DESC
LIMIT 10;
```

Transaction ID Wraparound Failures

Every time when we do insert, update or delete a unique transaction ID number is assigned. Now, this ID number gets incremented automatically so as you might do a transaction, a number is assigned to that transaction and the counter will increment automatically.

PostgresQL stores this transaction information about each and every row in the system. And this information is used to determine whether a row will be visible or not to other transactions.

Assume a system is running for a long time without any kind of maintenance, there's no order vacuum, there is no vacuuming happening then when the system reaches or the transaction number reaches anywhere close to 4 billion transaction,It will suffer something called **Transaction ID wrapper**. The transaction ID, which has to be assigned to each and every transaction, will become zero because there is no ID to be assigned after four billion transaction. Once the ID after down to zero or we can say starts from beginning , PG will not allow any new connection. This is to maintain sanity of the database.

What happens after four billion transaction is that whatever rows you have deleted in the past will be visible. Whatever updates you have done to record the old value will be visible.

The database will enter in inconsistent state. So in order to avoid this, we have to regularly do a vacuum.

- Multiversion concurrency control (MCC or MVCC), is a Concurrency Control method commonly used by DBMS to provide concurrent access to the database.
- MVCC depends on transaction ID numbers.
- Transaction IDs have limited size (32 bits)

- Cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound.
- The XID or transaction ID will wrap around to zero means starts from first transaction ID value. This is called catastrophic failure on the server.
- Transactions that were in the past appear to be in the future — which means their output become invisible.
- To void this situation it is necessary to vacuum every table in every database at least once every two billion transactions.

Query to know current status of transaction ID

```
#select
datname,age(datfrozenxid),current_setting('autovacuum_freeze_max_age')
from pg_database order by 2 desc;
```

datname	age	current_setting
iptr_prod	94,638	200000000
template0	94,638	200000000
postgres	94,638	200000000
dvdrental	94,638	200000000
auakua	94,638	200000000
template1	94,638	200000000
sdrh_tmp	231	200000000
vbts	96	200000000

```
#select txid_current();
```

```
txid_current
-----------------
95117
```

Check Tables for last vacuumed/ live and dead tuples:

```
SELECT schemaname, relname, n_live_tup, n_dead_tup, last_autovacuum
FROM pg_stat_all_tables
ORDER BY n_dead_tup
    / (n_live_tup
       * current_setting('autovacuum_vacuum_scale_factor')::float8
          + current_setting('autovacuum_vacuum_threshold')::float8)
     DESC
LIMIT 10;
```

--#db_level wraparound risk

```
SELECT datname
, age(datfrozenxid)
, current_setting('autovacuum_freeze_max_age')
, (age(datfrozenxid)::numeric/1000000000*100)::numeric(4,2) as
WRAPAROUND_RISK
FROM pg_database
 ORDER BY 2 DESC;
```

--Table level wraparound risk.

```
select c.oid::regclass as table_name,
greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as "TXID age",
(greatest(age(c.relfrozenxid),age(t.relfrozenxid))::numeric/
1000000000*100)::numeric(4,2) as "% WRAPAROUND RISK"

FROM pg_class c

LEFT JOIN pg_class t ON c.reltoastrelid = t.oid

WHERE c.relkind IN ('r', 'm')

ORDER BY 2 DESC;


select name, setting

FROM pg_settings

WHERE name ~ 'vacuum'

AND name ~'_age$'

ORDER BY 1 ASC;
```

Vacuum Freeze :

- Vacuum freeze is a special kind of vaccum, which marks rows as frozen.
- Vacuum Freeze marks a table's contents with a very special transaction timestamp that tells postgres that it does not need to be vacuumed, ever.
- Postgres reserves a special XID called FrozenTransactionId.
- FrozenTransacationId is always considered older than normal XID
- Vaccum_freeze_min_age controls how old an XID value has to be before it's replaced with FrozenXID

- VACUUM normally skips pages that don't have any dead row versions, but those pages might still have row versions with old XID values
- vacuum_freeze_table_age ensure all old XIDs have been replaced by FrozenXID, a scan of the whole table is needed.
- The debian package’s configuration is quite odd, putting the configuration files in the /etc/postgresql instead of the data area. The following example is the standalone backend’s

  ```
  $ /usr/lib/postgresql/13/bin/postgres  --single -D
  /var/lib/postgresql/13/main/base/
  --config-file=/etc/postgresql/13/main/postgresql.conf
  ```

  PostgreSQL stand-alone backend will start.

Below Settings are to make anti-wraparound autovacuum run more often, so that individual runs are smaller. Further improve matters for this table if you set vacuum_freeze_min_age to 0, so that all rows are frozen when autovacuum runs.

```
autovacuum_freeze_max_age = 100000000,

autovacuum_multixact_freeze_max_age = 100000000,

vacuum_freeze_min_age = 0
```

It can be set on the table too:

```
ALTER TABLE tab SET (

   autovacuum_freeze_max_age = 100000000,

   autovacuum_multixact_freeze_max_age = 100000000,

   vacuum_freeze_min_age = 0
```

```
);
```

To See how far the table is from TW:

```
# SELECT
       oid::regclass::text AS table,
       age(relfrozenxid) AS xid_age,
       mxid_age(relminmxid) AS mxid_age,
       least(
(SELECT setting::int
            FROM    pg_settings
            WHERE   name = 'autovacuum_freeze_max_age') -
age(relfrozenxid),
(SELECT setting::int
            FROM    pg_settings
            WHERE   name = 'autovacuum_multixact_freeze_max_age') -
mxid_age(relminmxid)
) AS tx_before_wraparound_vacuum,
pg_size_pretty(pg_total_relation_size(oid)) AS size,
pg_stat_get_last_autovacuum_time(oid) AS last_autovacuum
FROM    pg_class
WHERE   relfrozenxid != 0
AND oid > 16384
ORDER BY tx_before_wraparound_vacuum;
```

Routine Reindexing:

- Insert, updates and delete operations fragments the index over a period of time.
- A Fragmented index will have pages where logical order based on key value differs from the physical ordering inside the data file.
- Heavily fragmented indexes can degrade query performance because additional I/O is required to locate data to which the index points.
- Reindex rebuilds an index using the data stored in index table and eliminates empty spaces between pages
- Syntax : reindex index <index_name>;

Detect index fragmentation:

#select * from pgstatindex('idxtmp_cdt');

check the leaf_fragmentation value to get the % of fragmentation.

Run #reindex index idxtmp_cdt; – to achive the 0 fragmentation.

Cluster a Table:

When the data is being inserted into the table we can not restrict it to be entered in particular order, so the data is collected across multiple pages and these pages will be recalled when we search the data or totaling the count because , it is not entered in a particular sequence and it is scattered across pages. So cluster table is the option to arrange the data physically on the disk according to index.

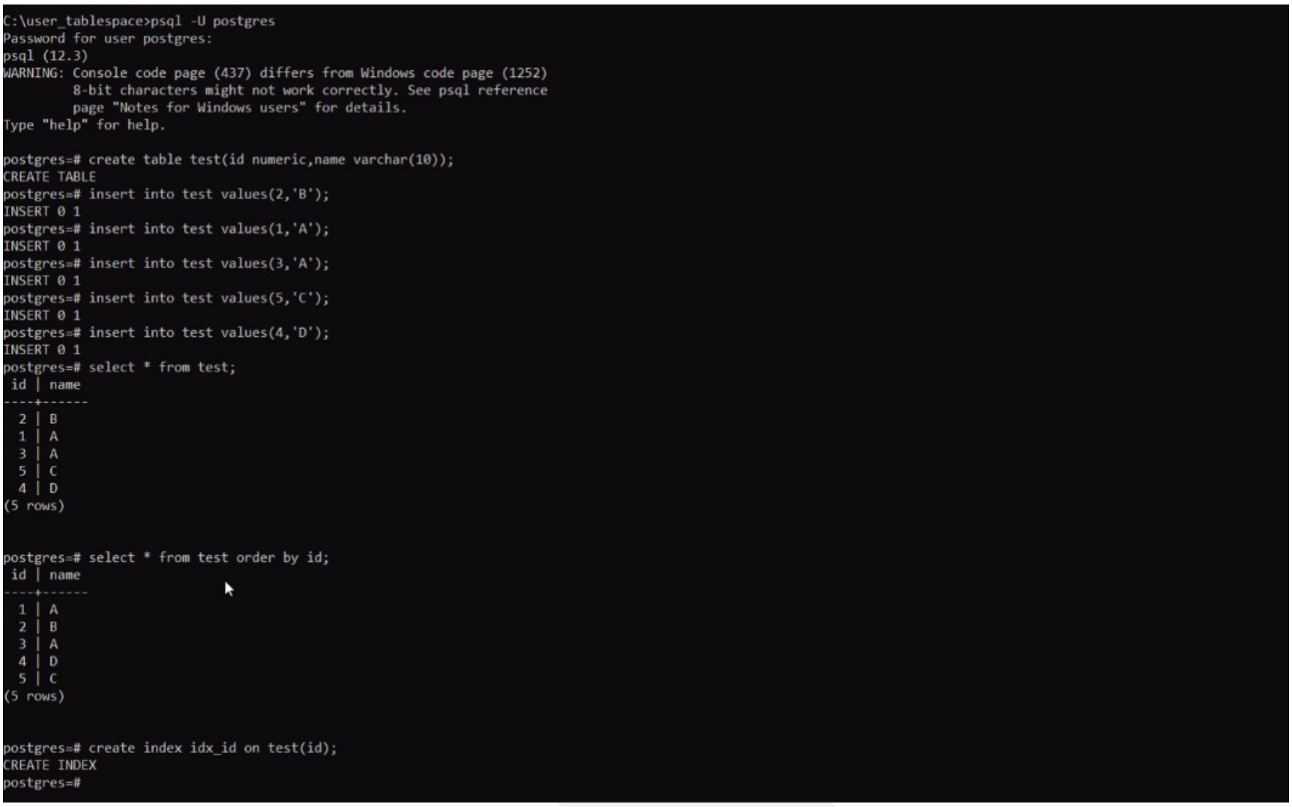
- CLUSTER instructs PostgreSQL to cluster the table specified by table_name based on the index specified by index_name.

- When a table is clustered, it is physically reordered based on the index information.
- **Clustering is a one-time operation**: when the table is subsequently updated, the changes are not clustered.
- **An Access Exclusive lock** is acquired .
- Cluster, lowers disk access and speeds up query when accessing a range of indexed values. Because it has less pages to search.
- Cluster should not be executed during peak hours in production environment.
- Syntax for cluster:
- **CLUSTER table USING index_name;**
- Next time you should execute only CLUSTER TABLE because It knows that which index already defined as CLUSTER.

PostgreSQL does not have direct implementation of CLUSTER index like Microsoft SQL Server. In PostgreSQL, we have one **CLUSTER command which is similar to Cluster Index**.

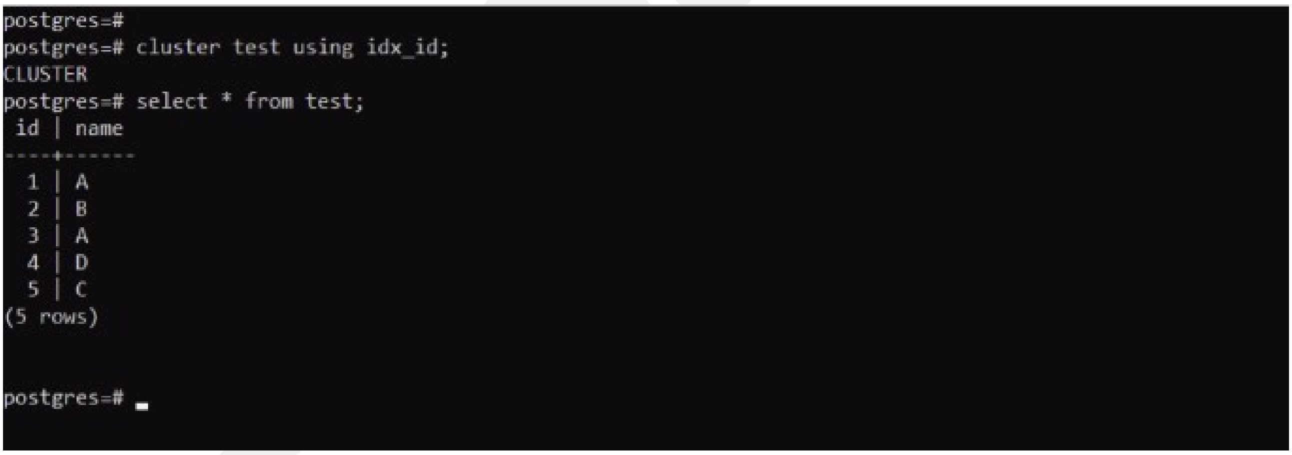
Example cluster table:

Created a table and insert data without ordering. Now we need to fetch the data with order by clause to get sorted data. We created an index on ID.



```
C:\user_tablespace>psql -U postgres
Password for user postgres:
psql (12.3)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# create table test(id numeric,name varchar(10));
CREATE TABLE
postgres=# insert into test values(2,'B');
INSERT 0 1
postgres=# insert into test values(1,'A');
INSERT 0 1
postgres=# insert into test values(3,'A');
INSERT 0 1
postgres=# insert into test values(5,'C');
INSERT 0 1
postgres=# insert into test values(4,'D');
INSERT 0 1
postgres=# select * from test;
 id | name
----+------
  2 | B
  1 | A
  3 | A
  5 | C
  4 | D
(5 rows)


postgres=# select * from test order by id;
 id | name
----+------
  1 | A
  2 | B
  3 | A
  4 | D
  5 | C
(5 rows)


postgres=# create index idx_id on test(id);
CREATE INDEX
postgres=#
```

Now we can get the ordered results without order by clause by clustering the table according to index.

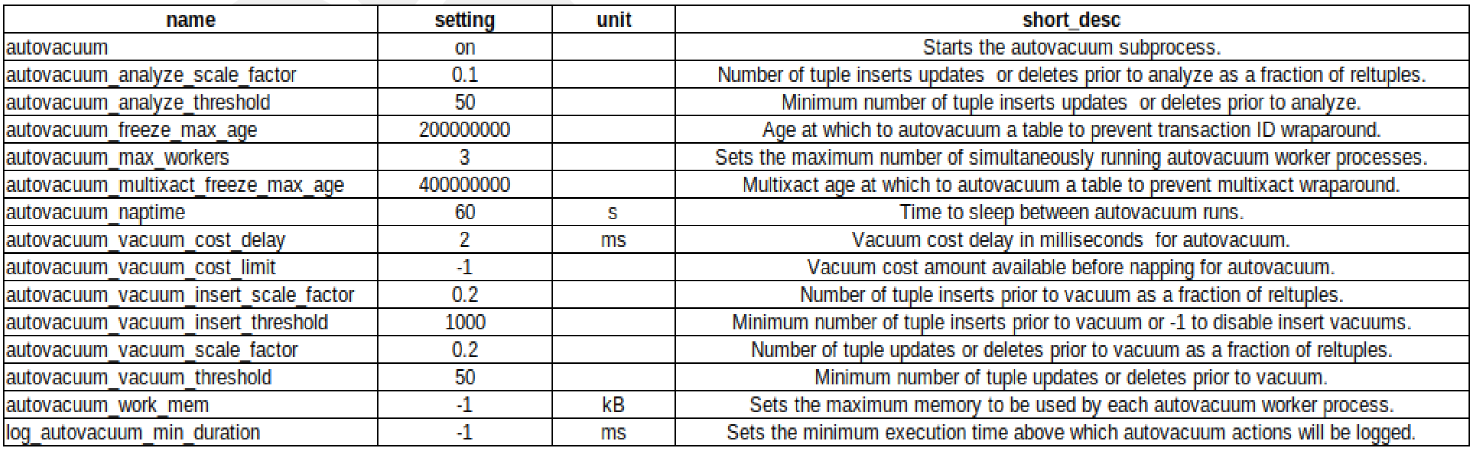


```
postgres=#
postgres=# cluster test using idx_id;
CLUSTER
postgres=# select * from test;
 id | name
----+------
  1 | A
  2 | B
  3 | A
  4 | D
  5 | C
(5 rows)


postgres=# _
```

Autovacuum Daemon:

- Autovacuum feature is used to automate the execution of VACUUM and ANALYZE commands.
- Autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples based on statistics collection.
- Autovacuum launcher is in charge of starting autovacuum worker processes for all databases.
- Launcher will distribute the work across time , attempting to start the worker start one worker within each database every autovacuum_naptime seconds(sleep time).
- Workers check for inserts, update and deletes and execute vacuum and analyze if needed.
- View Autovaccum settings

  # select * from pg_settings where name like '%autovacuum%';



name	setting	unit	short_desc
autovacuum	on		Starts the autovacuum subprocess.
autovacuum_analyze_scale_factor	0.1		Number of tuple inserts updates or deletes prior to analyze as a fraction of reltuples.
autovacuum_analyze_threshold	50		Minimum number of tuple inserts updates or deletes prior to analyze.
autovacuum_freeze_max_age	200000000		Age at which to autovacuum a table to prevent transaction ID wraparound.
autovacuum_max_workers	3		Sets the maximum number of simultaneously running autovacuum worker processes.
autovacuum_multixact_freeze_max_age	400000000		Multixact age at which to autovacuum a table to prevent multixact wraparound.
autovacuum_naptime	60	s	Time to sleep between autovacuum runs.
autovacuum_vacuum_cost_delay	2	ms	Vacuum cost delay in milliseconds for autovacuum.
autovacuum_vacuum_cost_limit	-1		Vacuum cost amount available before napping for autovacuum.
autovacuum_vacuum_insert_scale_factor	0.2		Number of tuple inserts prior to vacuum as a fraction of reltuples.
autovacuum_vacuum_insert_threshold	1000		Minimum number of tuple inserts prior to vacuum or -1 to disable insert vacuums.
autovacuum_vacuum_scale_factor	0.2		Number of tuple updates or deletes prior to vacuum as a fraction of reltuples.
autovacuum_vacuum_threshold	50		Minimum number of tuple updates or deletes prior to vacuum.
autovacuum_work_mem	-1	kB	Sets the maximum memory to be used by each autovacuum worker process.
log_autovacuum_min_duration	-1	ms	Sets the minimum execution time above which autovacuum actions will be logged.

Uninstalling PostgreSQL Windows:

By executing the uninstall utility In postgreSQL installation folder postgreSQL canbe uninstalled from windows.

Uninstall popup will ask to uninstall individual components i.e. PGADMIN, stack builder or library files as required or Uninstall entire application.

By choosing the “Entire Application” and clicking ok, uninstall process will starts and uninstall entire application from disk except data directory.

Data directory can be removed manually.

Uninstall PostgreSQL from Linux

Check the potgreSQL is running : #systemctl status postgresql12

if running then stop it : #sysemctl stop postgresql12

Now uninstall : yum remove postgres/* – **linux**

and remove data folder manually.

Ubuntu :

```
#sudo apt-get --purge remove postgresql
#sudo apt-get purge postgresql*
#sudo apt-get --purge remove postgresql postgresql-doc postgresql-common
#dpkg -l | grep postgres
#sudo rm -rf /var/lib/postgresql/
#sudo rm -rf /var/log/postgresql/
#sudo rm -rf /etc/postgresql/
```

Upgrading PostgreSQL

What is Upgrade

- Upgrading database from one PostgreSQL release to a newer one.
- PostgreSQL version numbers consist of a major and a minor version number. Ex: 10.1. (major is 10 , 1 is minor)
- Major releases of PostgreSQL, the internal data storage format is subject to change.
- Minor releases never change the internal storage format and are always compatible with earlier and later minor releases of the same major version number.
- Before PostgreSQL version 10.0, version numbers consist of three numbers. Ex: 9.5.6 (9.5 is major , 6 is minor)

REASON FOR UPGRADE

- Security Fixes
- Enhanced Features
- Resolved Bugs and Other Issues
- Reduced Costs by performance improvement in newer version
- End of Support

UPGRADE SOLUTIONS

- **Upgrading Data via pg_dumpall.**

  This is traditional method , very effective in order to use but it takes a lot of time so it is good for smaller databases. It requires downtime and adequate additional disk space to hold the new and old copies of dump.

- **Upgrading Data via pg_upgrade.**

  This is the faster upgrade utility provided for all kind of major and minor upgrade.

- Upgrading Data via Replication.

UPGRADING VIA PG_DUMPALL

1. Take the entire backup.

Ensure all the application closed and all connections should be stopped before backup to take consistent copy of backup.

Connections can be blocked in pg_hba.conf using reject authentication method.

```
windows/> pg_dumpall -U postgres > c:\bkpfolder_old\bkp1.sql
```

```
Linux #pg_dumpall > /opt/bkpfolder/bkp1.sql
```

it will ask the password for each database, if the database count is huge the environment variable PGPASS can be set to avoid inputting the password for each db.

2. After the successful backup stop the old cluster #pg_ctl -D <datafolder path> stop.

3. Rename the old PG directory.

4. Install and start new version and check proper installation.

5. Stop new cluster and restore the configuration changes in new postgresql.conf and pg_hba.conf from old one.

6. After making the changes in configuration file, start the cluster.

7. Delete the “Create role postgres ..“ line from the dump. (Optional)

8. Restore the dump

```
C:\>psql -U postgres -f C:\bkpfolder_old\bkp1.sql
```

```
# psql -f /opt/bkpfolder/bkp1.sql
```

9. on the successful restore old postgres version can be uninstalled as per process described in uninstall section of this course.

UPGRADE VIA PG_UPGRADE (Formerly PG_MIGRATOR)

- Pg_Upgrade (formerly called pg_migrator) allows data stored in PostgreSQL data files to be upgraded to a later PostgreSQL major version without the data dump/reload.
- Primary used for major PostgreSQL version upgrades.
- Major PostgreSQL releases regularly add new features that often change the layout of the system tables, but the internal data storage format rarely changes.
- pg_upgrade perform rapid upgrades by creating new system tables and simply reusing the old user data files.
- pg_upgrade does its best to make sure the old and new clusters are binary-compatible

PG_UPGRADE

Syntax:

PG_UPGRADE utility is more efficient then pg_dumpall, **it is having compatibility check option to check the compatibility between old and new version**. It is faster then pg_dumpall.

/usr/pgsql-12/bin/pg_upgrade --old-bindir=/usr/pgsql-10/bin --new-bindir=/usr/pgsql-12/bin --old-datadir=/var/lib/pgsql/10/data --new-datadir=/var/lib/pgsql/12/data –Link/Clone

Options:

- -b bindir

--old-bindir=bindir

The old PostgreSQL executable directory; environment variable PGBINOLD

-B bindir

--new-bindir=bindir

The new PostgreSQL executable directory; environment variable PGBINNEW

-c

--check

check clusters only, don't change any data

-d configdir

--old-datadir=configdir

the old database cluster configuration directory; environment variable PGDATAOLD

-D configdir

--new-datadir=configdir

the new database cluster configuration directory; environment variable PGDATANEW

-k

--link

Use hard links instead of copying files to the new cluster. This is the fastest upgrade. Data directories remains in same location and creates a link between new cluster and old data dirs. T**he drawback is we can’t go back to old version.**

--clone (Default)

Use efficient file cloning instead of copying files to the new cluster. can go back to old version because links will not reset, but sufficient disk space is required to hold both old and new copies.

-?

--help

show help, then exit

See the practical example on next page.

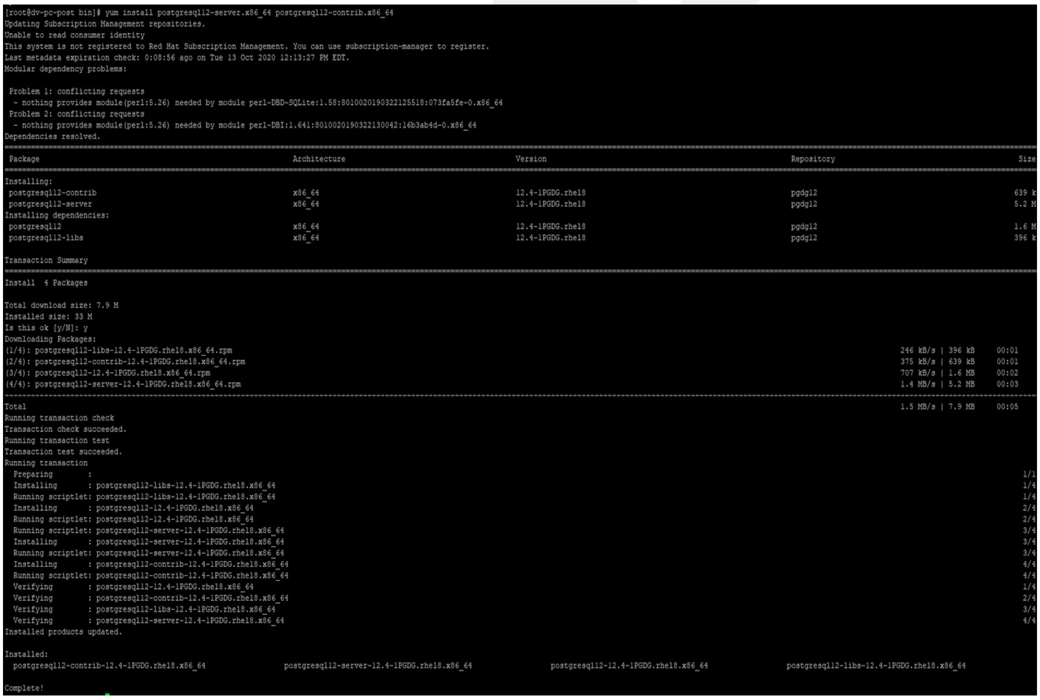
- Check whether PostgreSQL 10 is running or not and also find the location of data and bin directory.

  Data Location: /var/lib/pgsql/10/data
  Bin location: /usr/pgsql-10/bin

- Backup data. (Depending on the backup strategy being used to backup existing PostgreSQL) as postgres user

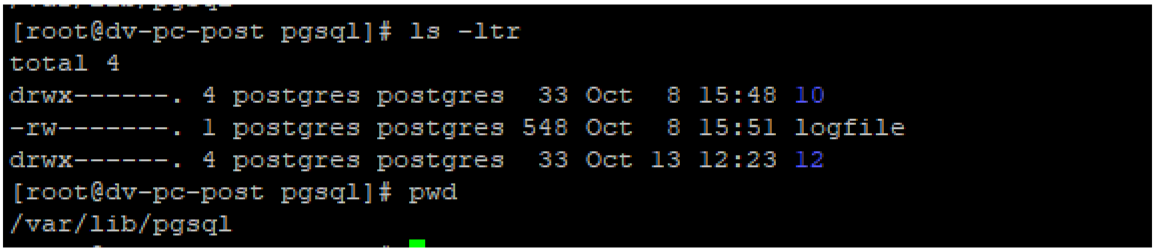
```
pg_basebackup  -D /var/lib/pgsql/10/backups
                 or
pg_dumpall > /var/lib/pgsql/10/backups/clusterall.sql
```

3) Yum install PostgreSQL 12 as root or as admin user.
Yum install postgresql12-server.x86_64 postgresql12-contrib.x86_64



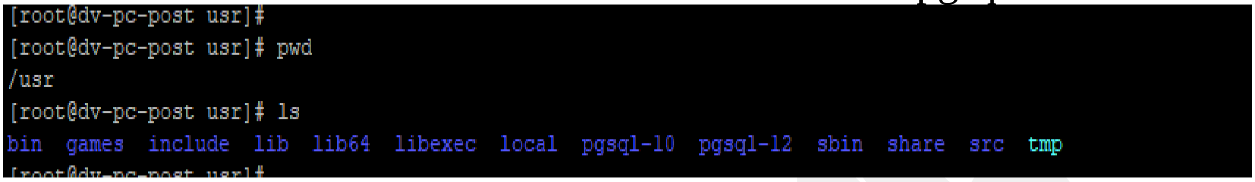
```
[root@dv-pc-post bin]# yum install postgresql12-server.x86_64 postgresql12-contrib.x86_64
Updating Subscription Management repositories.
Unable to read consumer identity
This system is not registered to Red Hat Subscription Management. You can use subscription-manager to register.
Last metadata expiration check: 0:08:56 ago on Tue 13 Oct 2020 12:13:27 PM EDT.
Modular dependency problems:

 Problem 1: conflicting requests
  - nothing provides module(perl:5.26) needed by module perl-DBD-SQLite:1.58:8010020190322125518:073fa5fe-0.x86_64
 Problem 2: conflicting requests
  - nothing provides module(perl:5.26) needed by module perl-DBI:1.641:8010020190322130042:16b3ab4d-0.x86_64
Dependencies resolved.
================================================================================================================
 Package                       Architecture          Version                   Repository                  Size
================================================================================================================
Installing:
 postgresql12-contrib          x86_64                12.4-1PGDG.rhel8          pgdg12                     639 k
 postgresql12-server           x86_64                12.4-1PGDG.rhel8          pgdg12                     5.2 M
Installing dependencies:
 postgresql12                  x86_64                12.4-1PGDG.rhel8          pgdg12                     1.6 M
 postgresql12-libs             x86_64                12.4-1PGDG.rhel8          pgdg12                     396 k

Transaction Summary
================================================================================================================
Install  4 Packages

Total download size: 7.9 M
Installed size: 33 M
Is this ok [y/N]: y
Downloading Packages:
(1/4): postgresql12-libs-12.4-1PGDG.rhel8.x86_64.rpm                           246 kB/s | 396 kB     00:01
(2/4): postgresql12-contrib-12.4-1PGDG.rhel8.x86_64.rpm                        375 kB/s | 639 kB     00:01
(3/4): postgresql12-12.4-1PGDG.rhel8.x86_64.rpm                                707 kB/s | 1.6 MB     00:02
(4/4): postgresql12-server-12.4-1PGDG.rhel8.x86_64.rpm                         1.4 MB/s | 5.2 MB     00:03
----------------------------------------------------------------------------------------------------------------
Total                                                                          1.5 MB/s | 7.9 MB     00:05
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                                                        1/1
  Installing       : postgresql12-libs-12.4-1PGDG.rhel8.x86_64                                              1/4
  Running scriptlet: postgresql12-libs-12.4-1PGDG.rhel8.x86_64                                              1/4
  Installing       : postgresql12-12.4-1PGDG.rhel8.x86_64                                                   2/4
  Running scriptlet: postgresql12-12.4-1PGDG.rhel8.x86_64                                                   2/4
  Running scriptlet: postgresql12-server-12.4-1PGDG.rhel8.x86_64                                            3/4
  Installing       : postgresql12-server-12.4-1PGDG.rhel8.x86_64                                            3/4
  Running scriptlet: postgresql12-server-12.4-1PGDG.rhel8.x86_64                                            3/4
  Installing       : postgresql12-contrib-12.4-1PGDG.rhel8.x86_64                                           4/4
  Running scriptlet: postgresql12-contrib-12.4-1PGDG.rhel8.x86_64                                           4/4
  Verifying        : postgresql12-12.4-1PGDG.rhel8.x86_64                                                   1/4
  Verifying        : postgresql12-contrib-12.4-1PGDG.rhel8.x86_64                                           2/4
  Verifying        : postgresql12-libs-12.4-1PGDG.rhel8.x86_64                                              3/4
  Verifying        : postgresql12-server-12.4-1PGDG.rhel8.x86_64                                            4/4
Installed products updated.

Installed:
  postgresql12-contrib-12.4-1PGDG.rhel8.x86_64      postgresql12-server-12.4-1PGDG.rhel8.x86_64      postgresql12-12.4-1PGDG.rhel8.x86_64      postgresql12-libs-12.4-1PGDG.rhel8.x86_64

Complete!
```

4)Check the location of the installed PostgreSQL 12 and old version Postgresql 10.



```
[root@dv-pc-post pgsql]# ls -ltr
total 4
drwx------. 4 postgres postgres  33 Oct  8 15:48 10
-rw-------. 1 postgres postgres 548 Oct  8 15:51 logfile
drwx------. 4 postgres postgres  33 Oct 13 12:23 12
[root@dv-pc-post pgsql]# pwd
/var/lib/pgsql
```

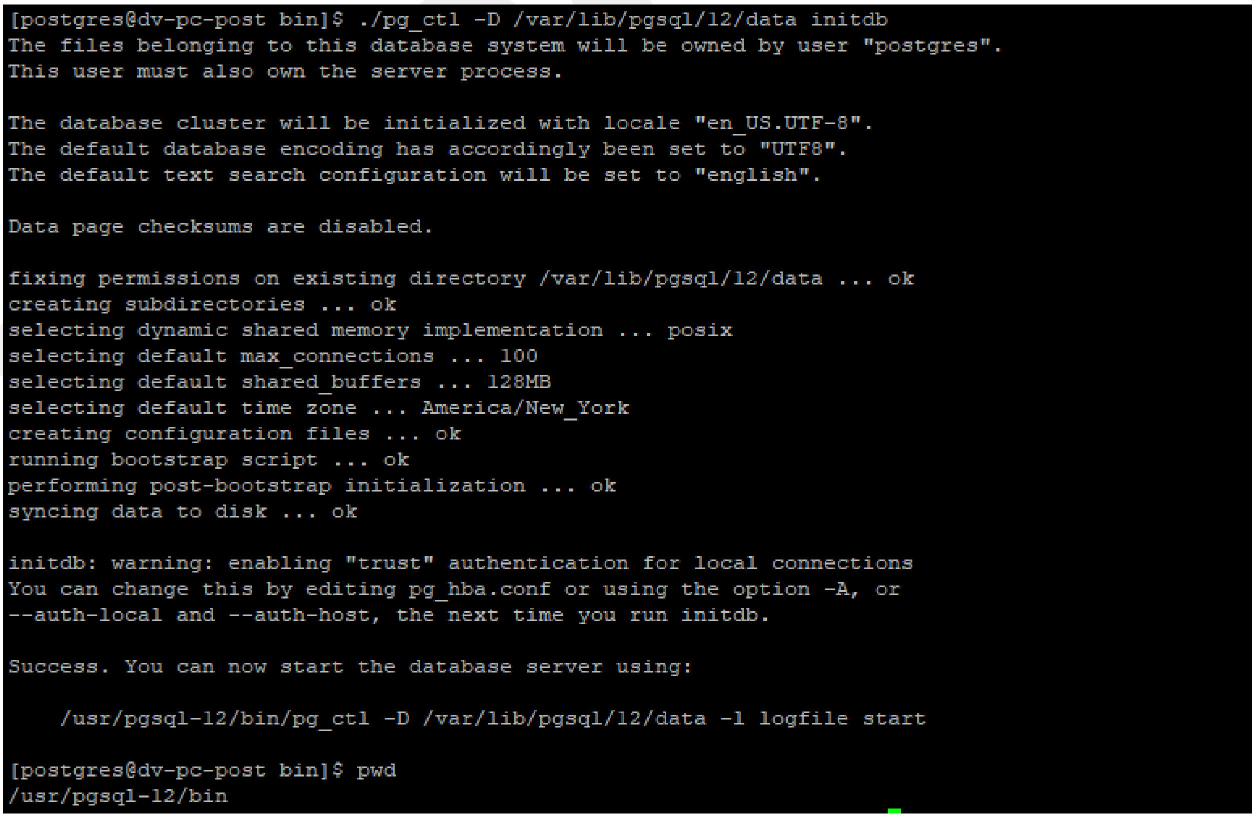
You should see both the version 10 and 12 in /var/lib/pgsql/ and usr/



```
[root@dv-pc-post usr]#
[root@dv-pc-post usr]# pwd
/usr
[root@dv-pc-post usr]# ls
bin  games  include  lib  lib64  libexec  local  pgsql-10  pgsql-12  sbin  share  src  tmp
[root@dv-pc-post usr]#
```

5 Initialize the PostgreSQL 12 cluster by navigating to the /usr/pgsql-12/bin directory

Run pg_ctl as postgres user.
./pg_ctl -D /var/lib/pgsql/12/data initdb



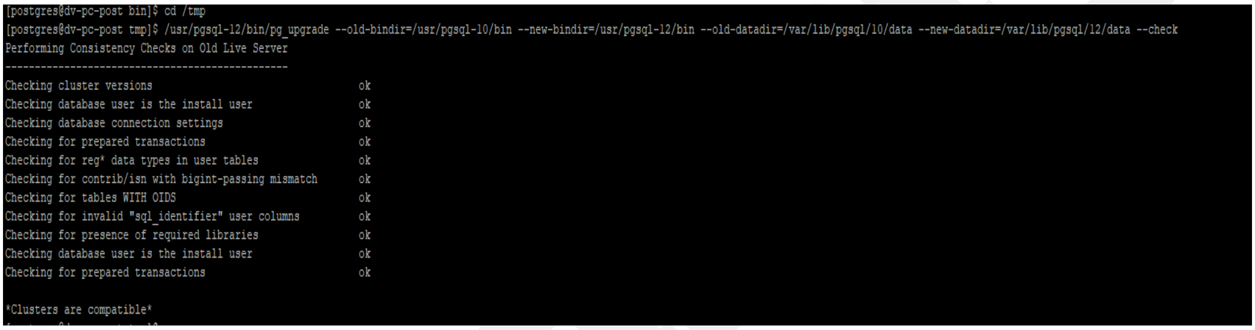
```
[postgres@dv-pc-post bin]$ ./pg_ctl -D /var/lib/pgsql/12/data initdb
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

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.

fixing permissions on existing directory /var/lib/pgsql/12/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... America/New_York
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    /usr/pgsql-12/bin/pg_ctl -D /var/lib/pgsql/12/data -l logfile start

[postgres@dv-pc-post bin]$ pwd
/usr/pgsql-12/bin
```

6 Check the postgresql-12 data directory for newly populated directory.
/var/lib/pgsql/12/data

7 Change the directory to /tmp or any directory where you have permission to write log file

Execute the below command to check whether new and old data directory are compatible.

/usr/pgsql-12/bin/pg_upgrade --old-bindir=/usr/pgsql-10/bin --new-bindir=/usr/pgsql-12/bin --old-datadir=/var/lib/pgsql/10/data --new-datadir=/var/lib/pgsql/12/data –check



```
[postgres@dv-pc-post bin]$ cd /tmp
[postgres@dv-pc-post tmp]$ /usr/pgsql-12/bin/pg_upgrade --old-bindir=/usr/pgsql-10/bin --new-bindir=/usr/pgsql-12/bin --old-datadir=/var/lib/pgsql/10/data --new-datadir=/var/lib/pgsql/12/data --check
Performing Consistency Checks on Old Live Server
------------------------------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Checking for tables WITH OIDS                               ok
Checking for invalid "sql_identifier" user columns          ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok

*Clusters are compatible*
```

This test should pass without any error and should say clusters are compatible. (Ensure that you are in the postgresql-10 bin folder when you execute this command)

8 Ensure the application is down and no connections can be made to PostgreSQL. If needed block connection from pg_hba.conf file.

9 Backup data again(Depending on the backup strategy being used to backup existing PostgreSQL) as postgres user

pg_basebackup -D /var/lib/pgsql/10/backups

or

pg_dumpall > /var/lib/pgsql/10/backups/clusterall.sql

10 .[postgres@dv-pc-post bin]$ cd /usr/pgsql-10/bin

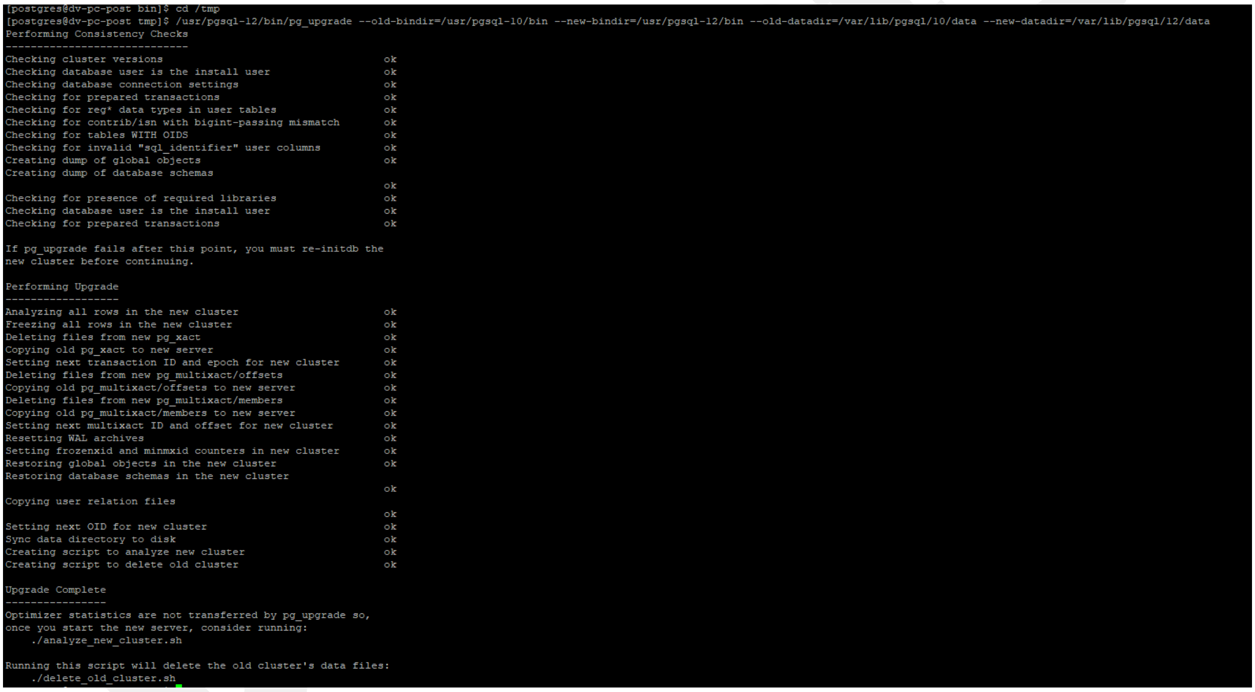
Stop postgresql-10 running on the server.

```
[postgres@dv-pc-post bin]$ ./pg_ctl -D /var/lib/pgsql/10/data stop
waiting for server to shut down.... done
server stopped
```

```
[postgres@dv-pc-post bin]$ ./pg_ctl -D /var/lib/pgsql/10/data status
pg_ctl: no server running
```

11 Navigate to /tmp folder and execute the below mentioned command as postgresql user.

```
/usr/pgsql-12/bin/pg_upgrade --old-bindir=/usr/pgsql-10/bin --new-bindir=/usr/pgsql-12/bin --old-datadir=/var/lib/pgsql/10/data --new-datadir=/var/lib/pgsql/12/data
```

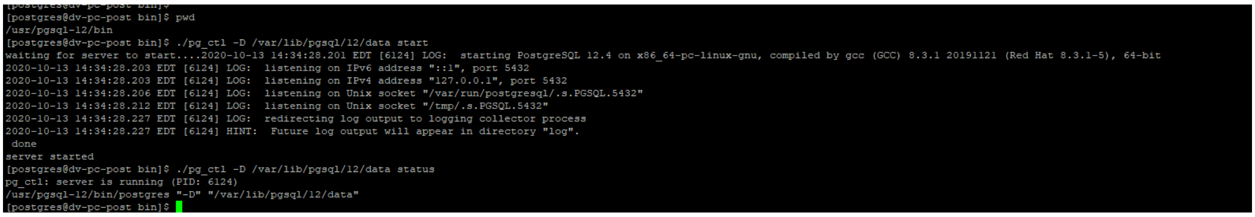


```
[postgres@dv-pc-post bin]$ cd /tmp
[postgres@dv-pc-post tmp]$ /usr/pgsql-12/bin/pg_upgrade --old-bindir=/usr/pgsql-10/bin --new-bindir=/usr/pgsql-12/bin --old-datadir=/var/lib/pgsql/10/data --new-datadir=/var/lib/pgsql/12/data
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Checking for tables WITH OIDS                               ok
Checking for invalid "sql_identifier" user columns          ok
Creating dump of global objects                             ok
Creating dump of database schemas
                                                            ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok

If pg_upgrade fails after this point, you must re-initdb the
new cluster before continuing.

Performing Upgrade
------------------
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
Deleting files from new pg_xact                             ok
Copying old pg_xact to new server                           ok
Setting next transaction ID and epoch for new cluster       ok
Deleting files from new pg_multixact/offsets                ok
Copying old pg_multixact/offsets to new server              ok
Deleting files from new pg_multixact/members                ok
Copying old pg_multixact/members to new server              ok
Setting next multixact ID and offset for new cluster        ok
Resetting WAL archives                                      ok
Setting frozenxid and minmxid counters in new cluster       ok
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster
                                                            ok
Copying user relation files
                                                            ok
Setting next OID for new cluster                            ok
Sync data directory to disk                                 ok
Creating script to analyze new cluster                      ok
Creating script to delete old cluster                       ok

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
```

12 Change the port to 5432 in Postgresql.conf file in the location /var/lib/pgsql/12/data/ (optional)
Change the port to 5433 in postgresql.conf file in the location /var/lib/pgsql/10/data/ (optional)

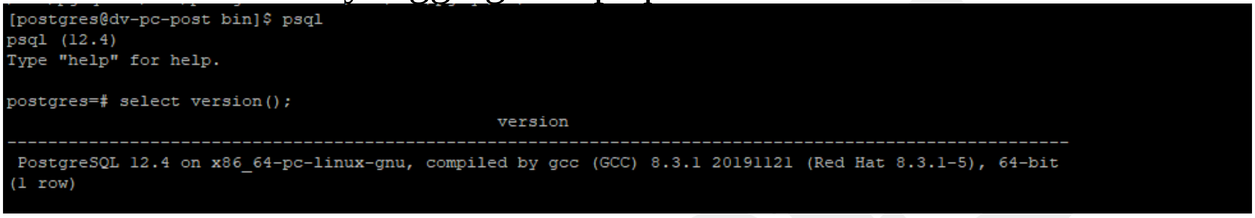
13 Now start the new postgresql-12 instance.

```
cd /usr/pgsql-12/bin/
./pg_ctl -D /var/lib/pgsql/12/data start
```



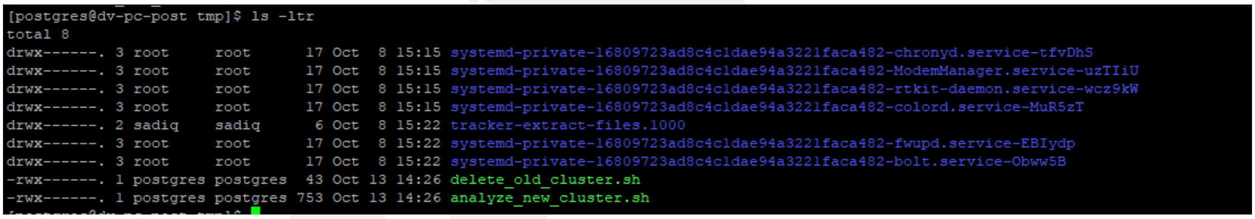
```
[postgres@dv-pc-post bin]$
[postgres@dv-pc-post bin]$ pwd
/usr/pgsql-12/bin
[postgres@dv-pc-post bin]$ ./pg_ctl -D /var/lib/pgsql/12/data start
waiting for server to start....2020-10-13 14:34:28.201 EDT [6124] LOG:  starting PostgreSQL 12.4 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5), 64-bit
2020-10-13 14:34:28.203 EDT [6124] LOG:  listening on IPv6 address "::1", port 5432
2020-10-13 14:34:28.203 EDT [6124] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2020-10-13 14:34:28.206 EDT [6124] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2020-10-13 14:34:28.212 EDT [6124] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
2020-10-13 14:34:28.227 EDT [6124] LOG:  redirecting log output to logging collector process
2020-10-13 14:34:28.227 EDT [6124] HINT:  Future log output will appear in directory "log".
 done
server started
[postgres@dv-pc-post bin]$ ./pg_ctl -D /var/lib/pgsql/12/data status
pg_ctl: server is running (PID: 6124)
/usr/pgsql-12/bin/postgres "-D" "/var/lib/pgsql/12/data"
[postgres@dv-pc-post bin]$
```

14 Check the version by logging into psql



```
[postgres@dv-pc-post bin]$ psql
psql (12.4)
Type "help" for help.

postgres=# select version();
                                                 version
---------------------------------------------------------------------------------------------------------
 PostgreSQL 12.4 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5), 64-bit
(1 row)
```

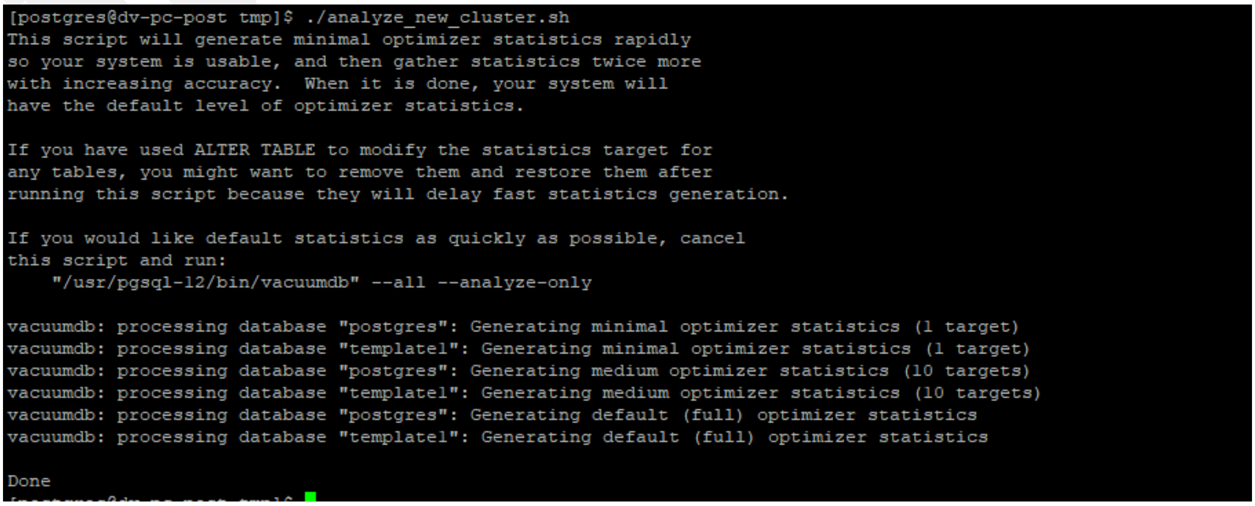
15 .There will be two scripts in the /tmp folder namely delete_old_cluster and analyze_new_cluster.sh



```
[postgres@dv-pc-post tmp]$ ls -ltr
total 8
drwx------. 3 root     root      17 Oct  8 15:15 systemd-private-16809723ad8c4c1dae94a3221faca482-chronyd.service-tfvDhS
drwx------. 3 root     root      17 Oct  8 15:15 systemd-private-16809723ad8c4c1dae94a3221faca482-ModemManager.service-uzTIiU
drwx------. 3 root     root      17 Oct  8 15:15 systemd-private-16809723ad8c4c1dae94a3221faca482-rtkit-daemon.service-wcz9kW
drwx------. 3 root     root      17 Oct  8 15:15 systemd-private-16809723ad8c4c1dae94a3221faca482-colord.service-MuR5zT
drwx------. 2 sadiq    sadiq      6 Oct  8 15:22 tracker-extract-files.1000
drwx------. 3 root     root      17 Oct  8 15:22 systemd-private-16809723ad8c4c1dae94a3221faca482-fwupd.service-EBIydp
drwx------. 3 root     root      17 Oct  8 15:22 systemd-private-16809723ad8c4c1dae94a3221faca482-bolt.service-Obww5B
-rwx------. 1 postgres postgres  43 Oct 13 14:26 delete_old_cluster.sh
-rwx------. 1 postgres postgres 753 Oct 13 14:26 analyze_new_cluster.sh
[postgres@dv-pc-post tmp]$
```

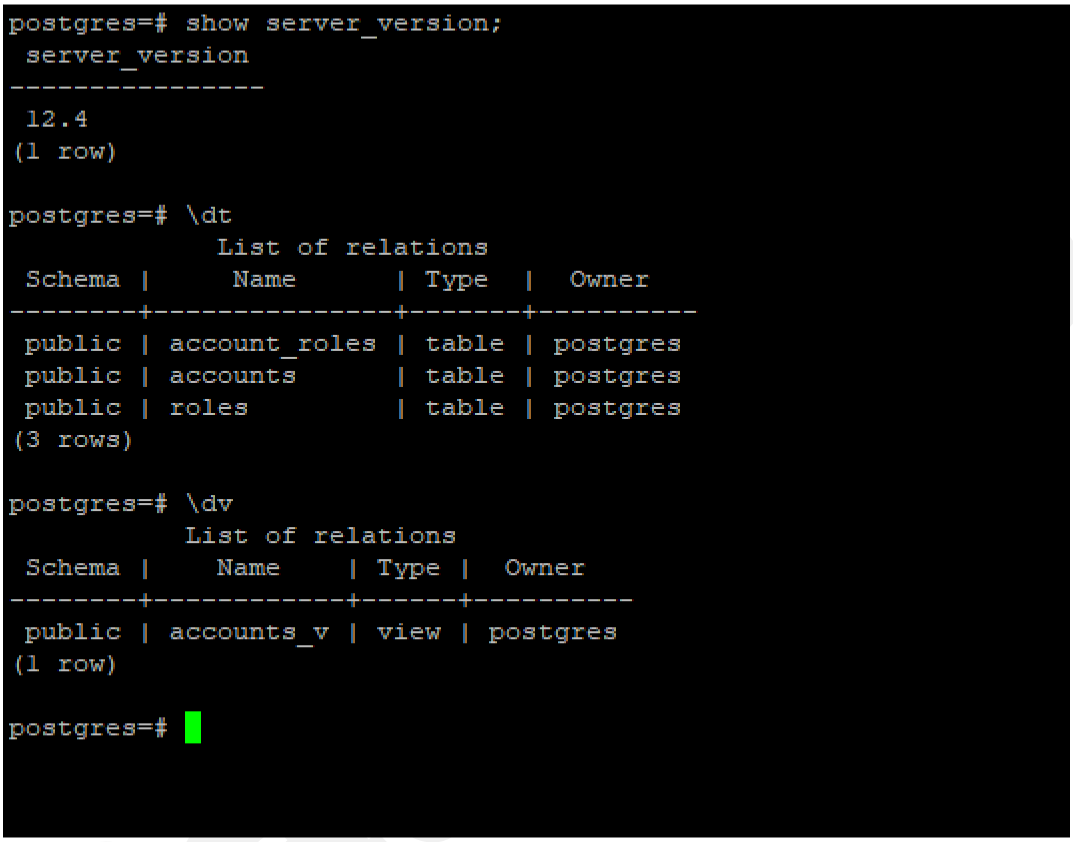
16 Execute the below mentioned script logged in as postgres user.

./ analyze_new_cluster.sh



```
[postgres@dv-pc-post tmp]$ ./analyze_new_cluster.sh
This script will generate minimal optimizer statistics rapidly
so your system is usable, and then gather statistics twice more
with increasing accuracy.  When it is done, your system will
have the default level of optimizer statistics.

If you have used ALTER TABLE to modify the statistics target for
any tables, you might want to remove them and restore them after
running this script because they will delay fast statistics generation.

If you would like default statistics as quickly as possible, cancel
this script and run:
    "/usr/pgsql-12/bin/vacuumdb" --all --analyze-only

vacuumdb: processing database "postgres": Generating minimal optimizer statistics (1 target)
vacuumdb: processing database "template1": Generating minimal optimizer statistics (1 target)
vacuumdb: processing database "postgres": Generating medium optimizer statistics (10 targets)
vacuumdb: processing database "template1": Generating medium optimizer statistics (10 targets)
vacuumdb: processing database "postgres": Generating default (full) optimizer statistics
vacuumdb: processing database "template1": Generating default (full) optimizer statistics

Done
[postgres@dv-pc-post tmp]$
```

17 Check the tables, views and other objects are present in the new upgrade database.



```
postgres=# show server_version;
 server_version
----------------
 12.4
(1 row)

postgres=# \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | account_roles | table | postgres
 public | accounts      | table | postgres
 public | roles         | table | postgres
(3 rows)

postgres=# \dv
           List of relations
 Schema |    Name    | Type |  Owner
--------+------------+------+----------
 public | accounts_v | view | postgres
(1 row)

postgres=#
```

18 Uninstall old postgresql 10 – (Purely your choice if you want to drop it right away or want to keep it for some time) software and old postgresql-10 directory.

19 Run ./ delete_old_cluster command to remove old cluster.

Introduction to Postgresql-13:

- Postgresql-13 was released on 2020-09-24 with performance improvement features.

- It includes significant improvements to its indexing and lookup system that benefit large databases.
- Space savings and performance gains from de-duplication of B-tree index entries
- Improved performance for queries that use aggregates or partitioned tables.
- Better query planning when using extended statistics.
- Parallelized vacuuming of indexes
- Incremental sorting

B-Tree Deduplication

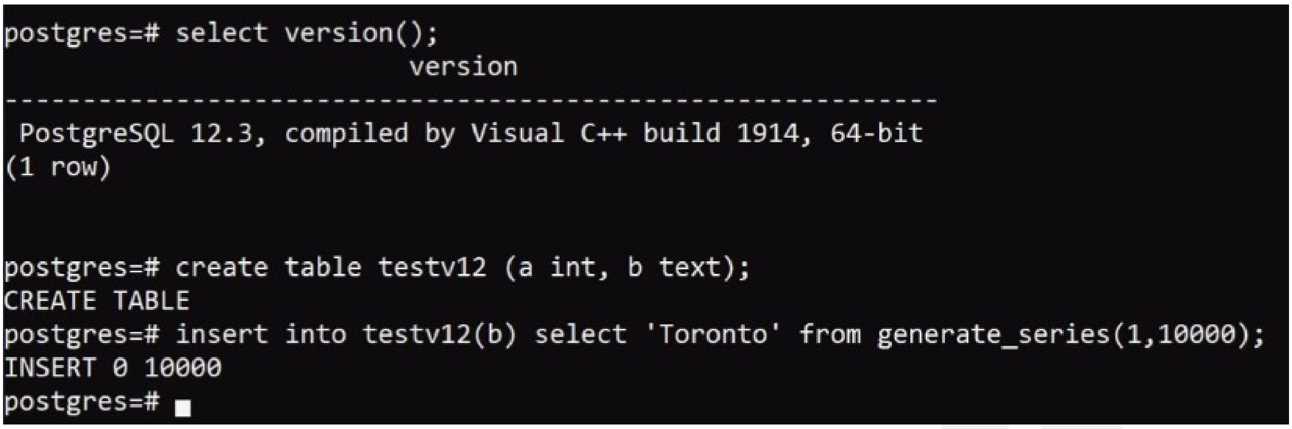
- Merging of duplicate values together and forming a single list for each value. So, key value appears only once.

  Ex: Before : 'Key A' ,(1,1), 'Key A',(1,2) ,'Key A', (1,3)

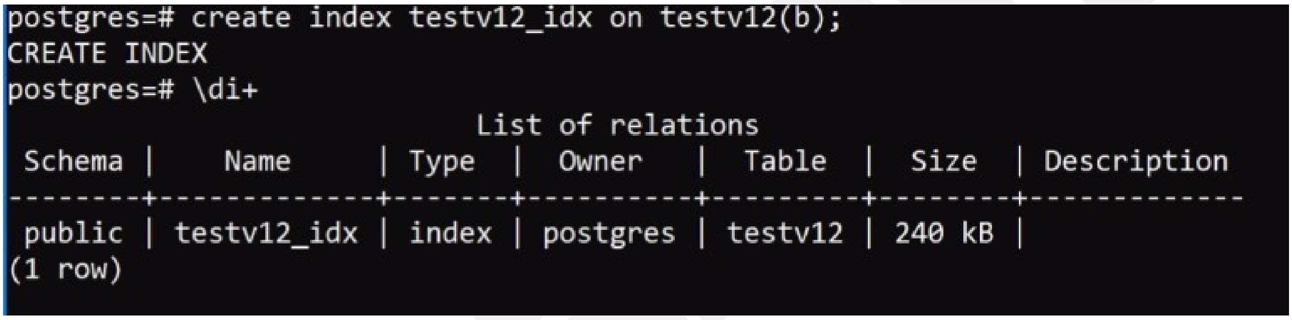
  Now : 'Key A' (1,1)(1,2)(1,3)
- Deduplication results in a smaller index size for indexes with **repeating entries**.
- Ram is efficiently used when the index is cached in shared buffers.
- Improved performance for queries that uses index scanning.
- Index bloating and Routine Index vacuum overhead is reduced.
- Users upgrading with **Pg_Upgrade will need to use "REINDEX" to make an existing index use this feature**.

Deduplication example before and now

Version 12:



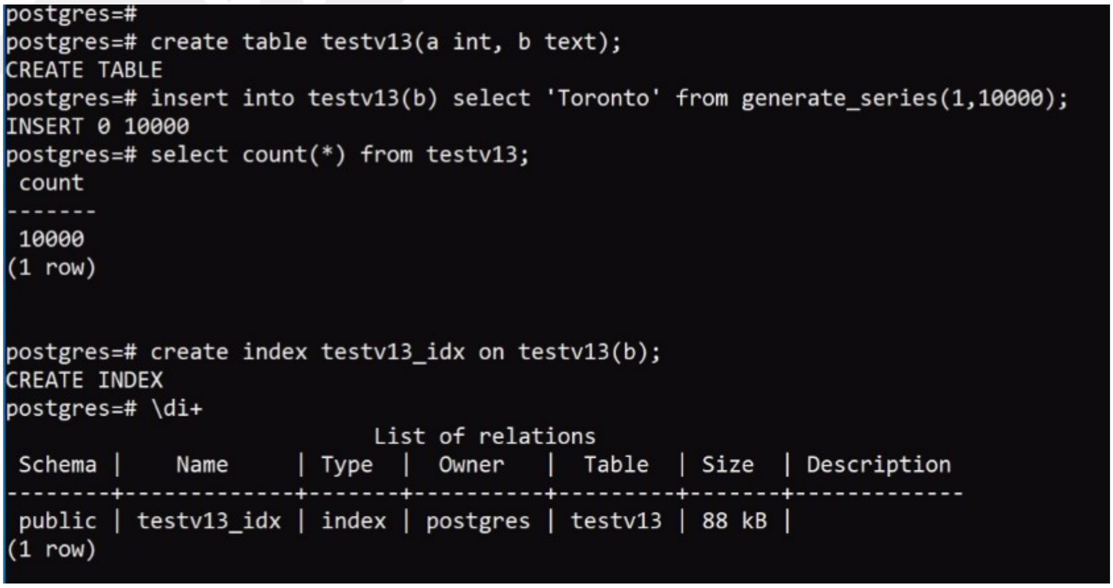
```
postgres=# select version();
                          version
------------------------------------------------------------
 PostgreSQL 12.3, compiled by Visual C++ build 1914, 64-bit
(1 row)


postgres=# create table testv12 (a int, b text);
CREATE TABLE
postgres=# insert into testv12(b) select 'Toronto' from generate_series(1,10000);
INSERT 0 10000
postgres=# ■
```



```
postgres=# create index testv12_idx on testv12(b);
CREATE INDEX
postgres=# \di+
                              List of relations
 Schema |    Name     | Type  |  Owner   |  Table  |  Size  | Description
--------+-------------+-------+----------+---------+--------+-------------
 public | testv12_idx | index | postgres | testv12 | 240 kB |
(1 row)
```

index size is 240 KB.

Version 13:



```
postgres=#
postgres=# create table testv13(a int, b text);
CREATE TABLE
postgres=# insert into testv13(b) select 'Toronto' from generate_series(1,10000);
INSERT 0 10000
postgres=# select count(*) from testv13;
 count
-------
 10000
(1 row)


postgres=# create index testv13_idx on testv13(b);
CREATE INDEX
postgres=# \di+
                              List of relations
 Schema |    Name     | Type  |  Owner   |  Table  | Size  | Description
--------+-------------+-------+----------+---------+-------+-------------
 public | testv13_idx | index | postgres | testv13 | 88 kB |
(1 row)
```

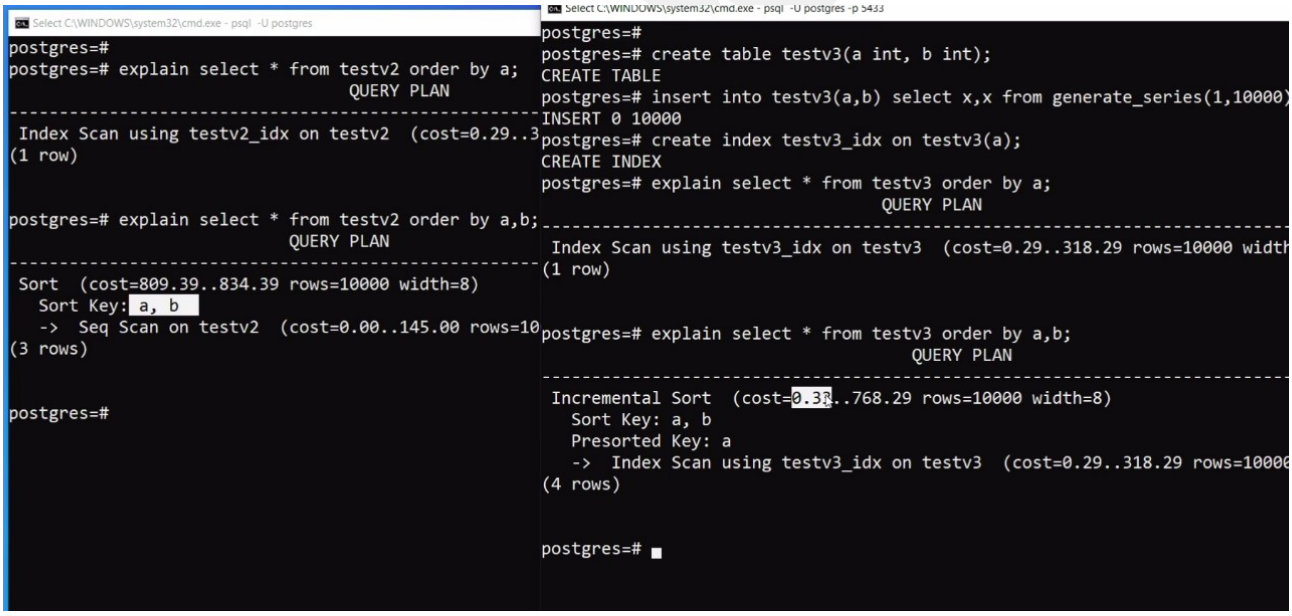
Index size is 88kB with same scenario. Because all rows have same duplicate data.

Incremental Sorting

Incremental sorting, which accelerates sorting data when data that is sorted from earlier parts of a query are already sorted.

Example: index on c1 and you need to sort dataset by c1, c2. Then incremental sort can help you because it wouldn’t sort the whole dataset, but sort individual groups whose have the same value of c1 instead. The incremental sort is extremely helpful when you have a LIMIT clause.

See diff. Between version 12 and 13. cost is dramatically reduced for same query in version 13.


Select C:\WINDOWS\system32\cmd.exe - psql -U postgres
Select C:\WINDOWS\system32\cmd.exe - psql -U postgres -p 5433

```
postgres=#
postgres=# explain select * from testv2 order by a;
                                           QUERY PLAN
-----------------------------------------------------
 Index Scan using testv2_idx on testv2  (cost=0.29..3
(1 row)


postgres=# explain select * from testv2 order by a,b;
                                     QUERY PLAN
-----------------------------------------------------
 Sort  (cost=809.39..834.39 rows=10000 width=8)
   Sort Key: a, b
   ->  Seq Scan on testv2  (cost=0.00..145.00 rows=10
(3 rows)


postgres=#
```

```
postgres=#
postgres=# create table testv3(a int, b int);
CREATE TABLE
postgres=# insert into testv3(a,b) select x,x from generate_series(1,10000)
INSERT 0 10000
postgres=# create index testv3_idx on testv3(a);
CREATE INDEX
postgres=# explain select * from testv3 order by a;
                                  QUERY PLAN
--------------------------------------------------------------------------
 Index Scan using testv3_idx on testv3  (cost=0.29..318.29 rows=10000 widt
(1 row)


postgres=# explain select * from testv3 order by a,b;
                                     QUERY PLAN
--------------------------------------------------------------------------
 Incremental Sort  (cost=0.33..768.29 rows=10000 width=8)
   Sort Key: a, b
   Presorted Key: a
   ->  Index Scan using testv3_idx on testv3  (cost=0.29..318.29 rows=1000
(4 rows)


postgres=# 
```

Parallel Vacuum

VACUUM reclaims storage occupied by dead tuples

Tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done.

Max_parallel_maintenance_workers , Min_parallel_index_scan_size parameter governs parallel vacuum.

The degree of parallelization is either specified by the user or determined based on the number of indexes that the table has.

AutoVacuum for Append only transactions.

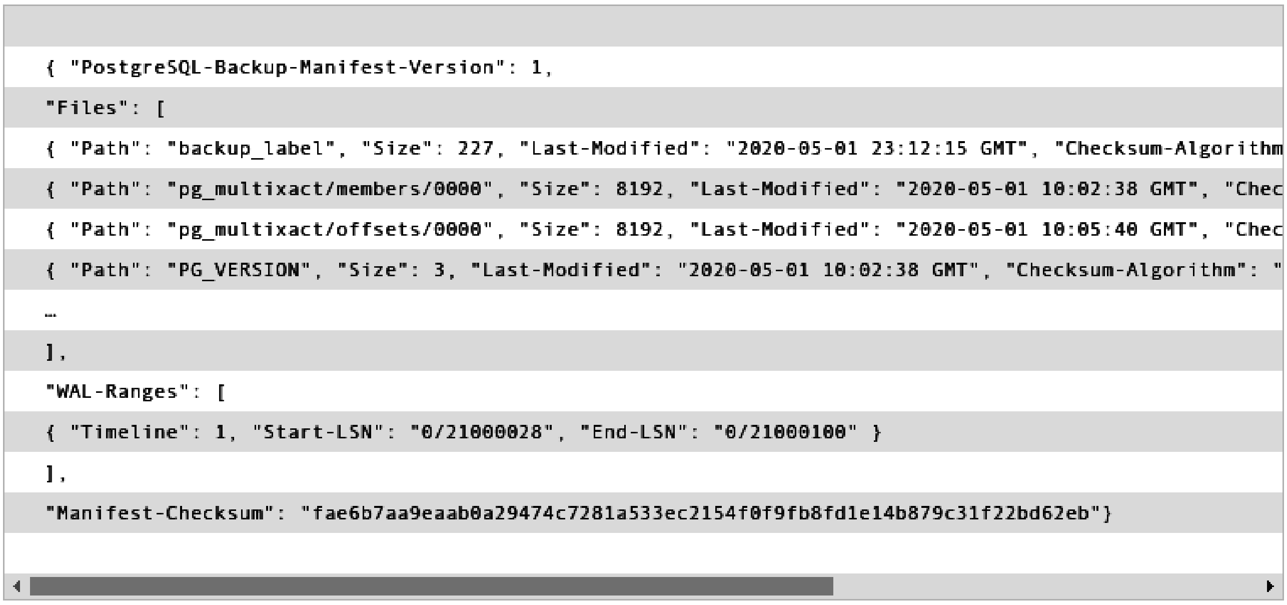
Syntax : VACUUM (PARALLEL 2, VERBOSE) <TableName>

Backup Manifests and pg_verifybackup in PostgreSQL 13

PostgreSQL 13 introduces two useful features to enhance automated validation of physical backups: backup manifests and a new tool called pg_verifybackup. In this short article I attempt to provide an overview of them.

Backup manifests

JSON listing of the content taken by a physical backup using pg_basebackup.



```
{ "PostgreSQL-Backup-Manifest-Version": 1,
"Files": [
{ "Path": "backup_label", "Size": 227, "Last-Modified": "2020-05-01 23:12:15 GMT", "Checksum-Algorithm
{ "Path": "pg_multixact/members/0000", "Size": 8192, "Last-Modified": "2020-05-01 10:02:38 GMT", "Chec
{ "Path": "pg_multixact/offsets/0000", "Size": 8192, "Last-Modified": "2020-05-01 10:05:40 GMT", "Chec
{ "Path": "PG_VERSION", "Size": 3, "Last-Modified": "2020-05-01 10:02:38 GMT", "Checksum-Algorithm": "
…
],
"WAL-Ranges": [
{ "Timeline": 1, "Start-LSN": "0/21000028", "End-LSN": "0/21000100" }
],
"Manifest-Checksum": "fae6b7aa9eaab0a29474c7281a533ec2154f0f9fb8fd1e14b879c31f22bd62eb"}
```

pg_verifybackup

A tool to verify that the content of a backup matches the given manifest.

pg_verifybackup expects to find also the WAL files from the start to the end of the backup, which I deliberately did not include in the backup. I can skip verification of the WAL files with the "-n” option:

#pg_verifybackup -n ~/backups/1588374735/

Trusted Extensions

Can install extensions without super user privileges if we have create privilege on database.

Ex : plperl,pgcrypto and ltree.

Drop Database:

DROP DATABASE DBNAME WITH (FORCE)

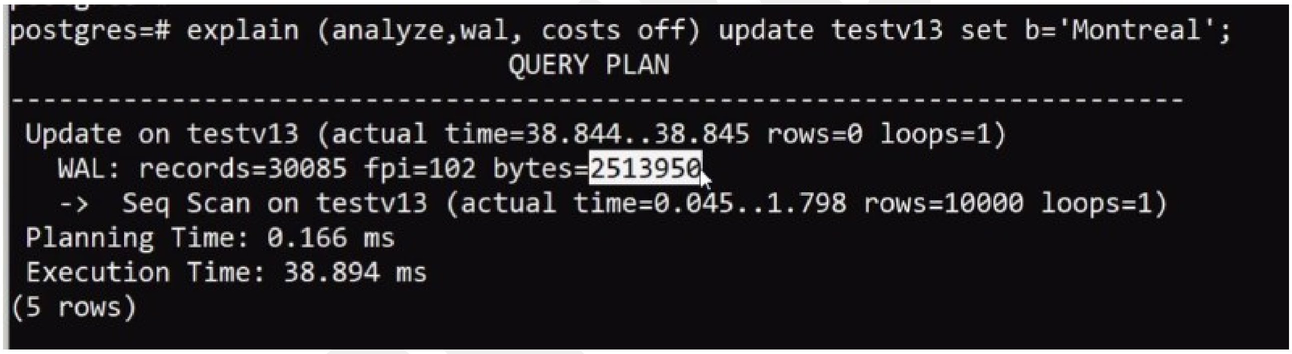
(in version 12 , If a user is connected to a database, postgresql will not allow you to drop that database until all user sessions are killed. Now, if there are multiple users who are connected to the database, it becomes a big task, to kill each and every session.) WITH (FORCE) is introduced in version 13 to do the same thing.

Explain tracks wal_usage:

EX: EXPLAIN (ANALYZE, WAL, COSTS OFF) UPDATE t1 SET id = 1 WHERE id = 1;

Whenever we do a bulk insert or update or delete, that changes are logged in the wall files.

Right now, if I want to know how much data or how much changes are written to WAL files in bytes we can use this feature.



```
postgres=# explain (analyze,wal, costs off) update testv13 set b='Montreal';
                               QUERY PLAN
-------------------------------------------------------------------------
 Update on testv13 (actual time=38.844..38.845 rows=0 loops=1)
   WAL: records=30085 fpi=102 bytes=2513950
   ->  Seq Scan on testv13 (actual time=0.045..1.798 rows=10000 loops=1)
 Planning Time: 0.166 ms
 Execution Time: 38.894 ms
(5 rows)
```

SYSTEM VIEWES:

- Pg_stat_activity to report a parallel worker's leader process.
- Pg_stat_progress_basebackup to report the progress of streaming base backups.
- Pg_stat_progress_analyze to report ANALYZE progress.
- Pg_shmem_allocations to display shared memory usage.