Introduction to PostgreSQL for Oracle and MySQL DBAs

Avinash Vallarapu
Percona

The History of PostgreSQL

Ingres

Year 1973 - INGRES (INteractive GRaphics Retrieval System) work on one of the world's first RDBMS was started by Eugene Wong and Michael Stonebraker at University of California at Berkeley

Year 1979 - Oracle Database first version was released

Early 1980's - INGRES used QUEL as its preferred Query Language. Whereas Oracle used SQL

Year 1985 - UC Berkeley INGRES research project officially ended

Postgres

Year 1986 - Postgres was introduced as a Post-Ingres evolution. Used POSTQUEL as its query language until 1994

Year 1995 - Postgres95 replaced Postgres with its support for SQL as a query language

PostgreSQL

Year 1996 - Project renamed to PostgreSQL to reflect the original name Postgres and its SQL Compatibility

Year 1997 - PostgreSQL first version - PostgreSQL 6.0 released

PostgreSQL Features

- Portable
  - Written in C
  - Flexible across all the UNIX platforms, Windows, MacOS and others
  - World’s most advanced open source database. Community-driven
  - ANSI/ISO Compliant SQL support
- Reliable
  - ACID Compliant
  - Supports Transactions
  - Uses Write Ahead Logging
- Scalable
  - MVCC
  - Table Partitioning
  - Tablespaces
  - FDWs
  - Sharding

PostgreSQL Advanced Features

- Security
  - Host-Based Access Control
  - Object-Level and Row-Level Security
  - Logging and Auditing
  - Encryption using SSL
- High Availability
  - Synchronous/Asynchronous Replication and Delayed Standby
  - Cascading Replication
  - Online Consistent Physical Backups and Logical Backups
  - PITR
- Other Features
  - Triggers and Functions/Stored Procedures
  - Custom Stored Procedural Languages like PL/pgSQL, PL/perl, PL/TCL, PL/php, PL/python, PL/java.
  - PostgreSQL Major Version Upgrade using pg_upgrade
  - Unlogged Tables, Parallel Query, Native Partitioning, FDWs
  - Materialized Views
  - Hot Standby - Slaves accept Reads

PostgreSQL Cluster

- After Initializing your PostgreSQL using initdb (similar to mysqld --initialize) and starting it, you can create multiple databases in it
- A group of databases running on one Server & One Port - Is called a Cluster in PostgreSQL
- PostgreSQL Cluster may be referred to as a PostgreSQL Instance as well
- A PostgreSQL Cluster or an Instance:
  - Serves only one TCP/IP Port
  - Has a Dedicated Data Directory
  - Contains 3 default databases: postgres, template0 and template1
- When you add a Slave(aka Standby) to your PostgreSQL Cluster(Master), it may be referred to as a PostgreSQL High Availability Cluster or a PostgreSQL Replication Cluster
- PostgreSQL Cluster that can accept Writes and ships WALs to Slave(Standby), is called a Master

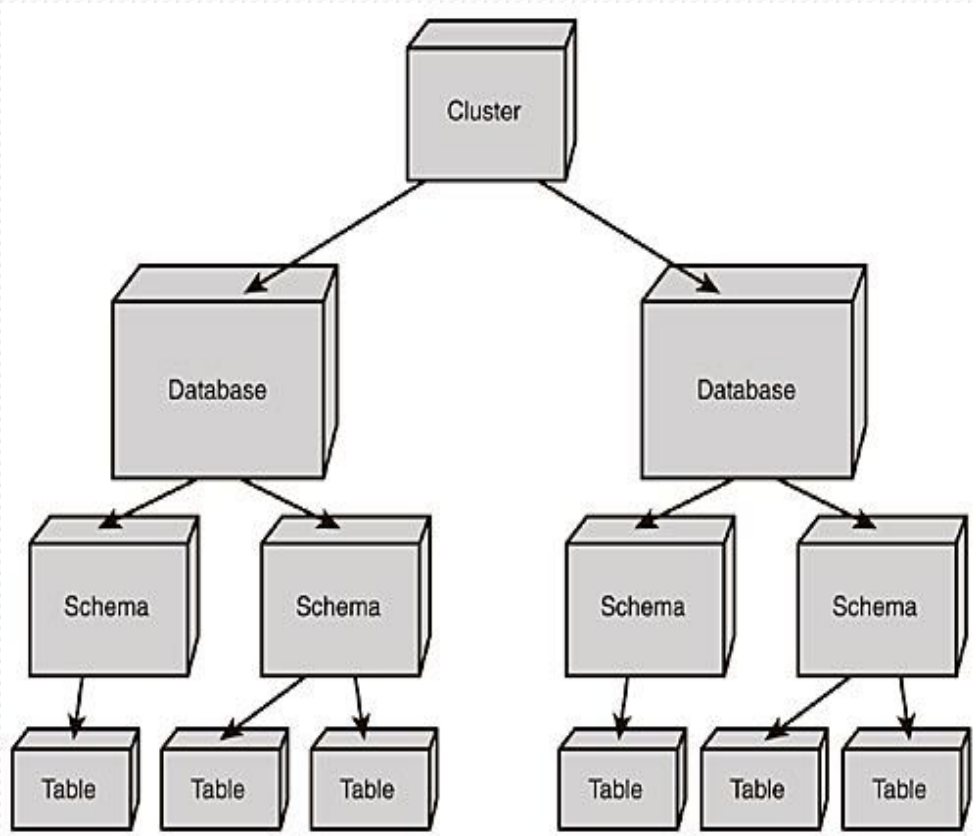
PostgreSQL Database and Schema

- A PostgreSQL Database can contain one or more Schemas
- Default Schema is - public schema
- A Schema in PostgreSQL is a logical entity that helps you group objects of a certain Application logic together. This helps you create multiple objects with the same name in one Database
- A Database can be related to a Parent Folder/Directory.You can always have more than 1 Database with one or more Schemas in it
- For example: In a Database named percona, a Table employee can exist in both scott and tiger schemas

Database: percona
Schema(s): scott & tiger
Tables: 1. scott.employee
2. tiger.employee

- A Fully Qualified Table Name: schemaname.tablename must be used to query a particular Table in a Schema

For example:
select * from scott.employee where salary > 10000;



PostgreSQL ACID Compliance

- **Atomicity**: Transactions. Either All or Nothing
  BEGIN …SQL1, SQL2, …SQLn…..COMMIT/ROLLBACK/END

- **Consistency**: Give me a consistent picture of the data based on Isolation Levels
  Let us see the following example when Isolation Level is READ_COMMITTED

  Query 1 : select count(*) from employees;

  9am: Records in employee table: 10000
  9:10 am: Query 1 Started by User 1
  9:11am: 2 employee records deleted by User 2
  9:12am: Query 1 that was started by User 1 Completed

  Result of Query 1 at 9:12am would still be 10000. A Consistent image as how it was at 9:00am

- **Isolation**: Prevent Concurrent data access through Locking

- **Durability**: Once the Data is committed, it must be safe
  Through WAL's, fsync, synchronous_commit, Replication

PostgreSQL Terminology

- PostgreSQL was designed in academia
  - Objects are defined in academic terms
  - Terminology based on relational calculus/algebra

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

PostgreSQL Installation

PostgreSQL Installation Using RPM's

PGDG Repository : PostgreSQL Global Development Group maintains YUM and APT repository

For YUM

https://yum.postgresql.org

For APT

https://apt.postgresql.org/pub/repos/apt/

Step 1:

Choose the appropriate rpm that adds pgdg repo to your server

yum install https://yum.postgresql.org/11/redhat/rhel-7.5-x86_64/pgdg-centos11-11-2.noarch.rpm

Step 2:

Install PostgreSQL using the following step

yum install postgresql11 postgresql11-contrib postgresql11-libs postgresql11-server

Initialize Your First PostgreSQL Cluster

- **initdb** is used to Initialize a PostgreSQL cluster

```
$ echo "PATH=/usr/pgsql-11/bin:$PATH">>~/.bash_profile
$ source .bash_profile

$ echo $PGDATA
/var/lib/pgsql/11/data

$initdb --version
initdb (PostgreSQL) 11.0

$ initdb
```

```
avi@percona:~ $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_CA.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/11/data ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting dynamic shared memory implementation ... posix
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

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:

    pg_ctl -D /var/lib/pgsql/11/data -l logfile start
```

Starting and Stopping a PostgreSQL

- PostgreSQL can be stopped and started from command line using **pg_ctl**
  - Starting PostgreSQL
    - ***pg_ctl -D $PGDATA start***
  - Stopping PostgreSQL
    - ***pg_ctl -D $PGDATA stop***

Shutdown Modes in PostgreSQL

- **-ms (Smart Mode - Default mode)**
  - Waits for all connections to exit and does not allow new transactions
  - Committed transactions applied to Disk through a CHECKPOINT before shutdown
  - May take more time on busy systems

  $ pg_ctl -D $PGDATA stop -ms
- **-mf (Fast Mode - Recommended on Busy Systems)**
  - Closes/Kills all the open transactions and does not allow new transactions. SIGTERM is sent to server processes to exit promptly
  - Committed transactions applied to Disk through a CHECKPOINT before shutdown
  - Recommended on Busy Systems

  $ pg_ctl -D $PGDATA stop -mf
- **-mi (Immediate Mode - Forced and Abnormal Shutdown during Emergencies)**
  - SIGQUIT is sent to all the processes to exit immediately, without properly shutting down
  - Requires Crash Recovery after Instance Start
  - Recommended in Emergencies

  $ pg_ctl -D $PGDATA stop -mi

psql and shortcuts

- Connect to your PostgreSQL using psql
  - ***$ psql***

    List the databases

    \l
    \l + (Observe the difference)

    To connect to your database

    \c dbname

    List Objects

    \dt -> List all the tables
    \dn -> List all the schemas
  - Show all backslash (shortcut) commands

    \?

PostgreSQL Architecture

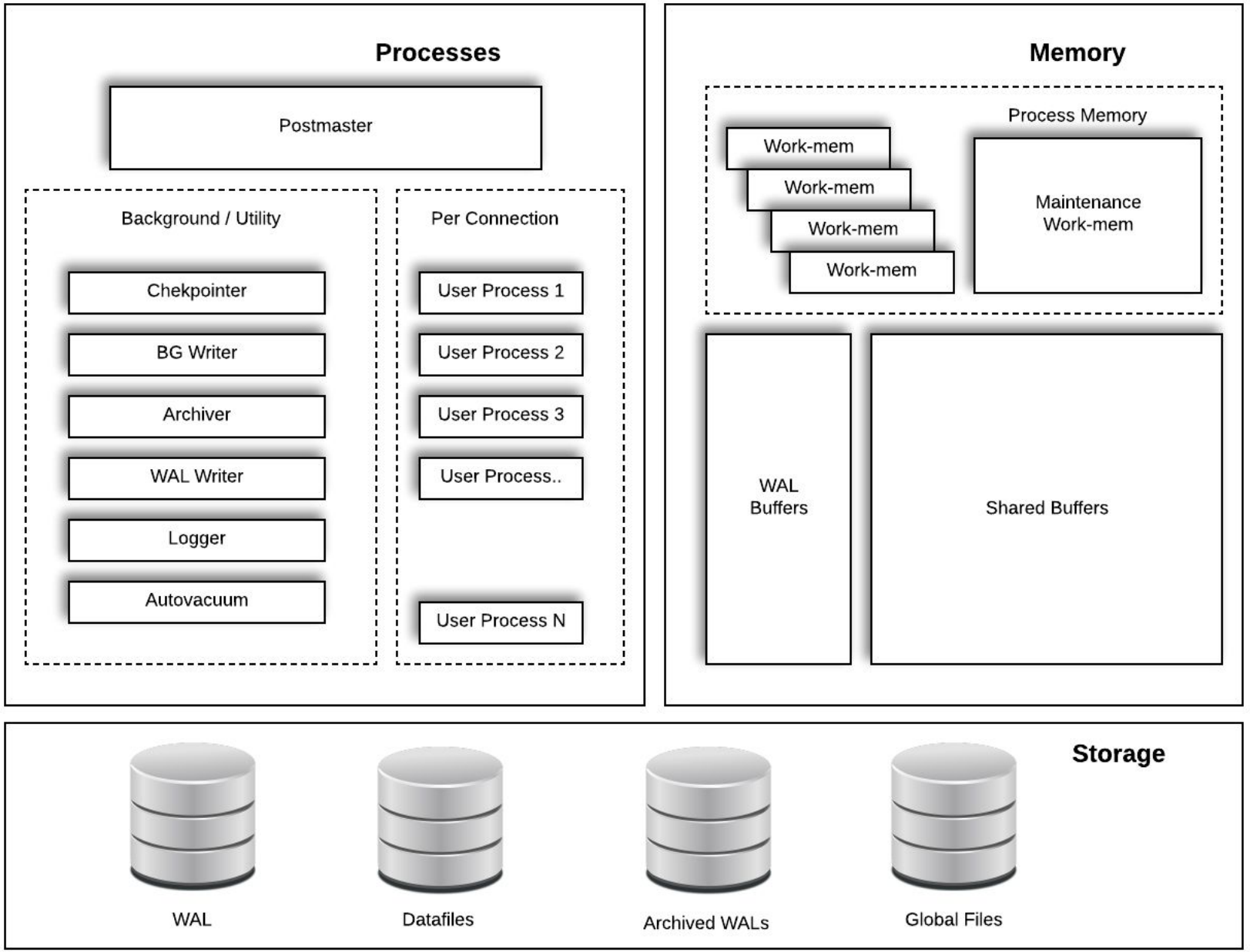
PostgreSQL Server

- Multi-Process Architecture
  - Postmaster (Parent PostgreSQL Process)
  - Backend Utility Processes
  - Per-Connection backend processes

Background Utility Processes

Start your PostgreSQL Instance and see the Postgres processes

```
avi@percona:~ $ps -eaf | grep postgres
postgres  1211     1  0 12:57 ?        00:00:00 /usr/pgsql-11/bin/postgres -D /var/lib/pgsql/11/data
postgres  1212  1211  0 12:57 ?        00:00:00 postgres: logger
postgres  1214  1211  0 12:57 ?        00:00:00 postgres: checkpointer
postgres  1215  1211  0 12:57 ?        00:00:00 postgres: background writer
postgres  1216  1211  0 12:57 ?        00:00:00 postgres: walwriter
postgres  1217  1211  0 12:57 ?        00:00:00 postgres: autovacuum launcher
postgres  1218  1211  0 12:57 ?        00:00:00 postgres: stats collector
postgres  1219  1211  0 12:57 ?        00:00:00 postgres: logical replication launcher
```


Processes
Postmaster
Background / Utility
Chekpointer
BG Writer
Archiver
WAL Writer
Logger
Autovacuum
Per Connection
User Process 1
User Process 2
User Process 3
User Process..
User Process N
Memory
Process Memory
Work-mem
Work-mem
Work-mem
Work-mem
Maintenance
Work-mem
WAL
Buffers
Shared Buffers
Storage
WAL
Datafiles
Archived WALs
Global Files

PostgreSQL Components

- **Postmaster:**
  - Master database control process
  - Responsible for startup and shutdown
  - Spawning other necessary backend processes

Utility Processes

- **BGWriter**:
  - Background Writer
  - Writes/Flushes dirty data blocks to disk
- **WAL Writer**:
  - Writes WAL Buffers to Disk
  - WAL Buffers are written to WALs(Write-Ahead Logs) on the Disk
- **Autovacuum**:
  - Starts Autovacuum worker processes to start a vacuum and analyze
- **Checkpointer**:
  - Perform a CHECKPOINT that ensures that all the changes are flushed to Disk
  - Depends on configuration parameters

Utility Processes

- **Archiver**:
  - Archives Write-Ahead-Logs
  - Used for High Availability, Backups, PITR
- **Logger**:
  - Logs messages, events, error to syslog or log files.
  - Errors, slow running queries, warnings,..etc. are written to log files by this Process
- **Stats Collector**:
  - Collects statistics of Relations.

Utility Processes

- **WAL Sender**:
  - Sends WALs to Replica(s)
  - One WAL Sender for each Slave connected for Replication
- **WAL Receiver**:
  - Started on a Slave(aka Standby or Replica) in Replication
  - Streams WALs from Master
- **bgworker**:
  - PostgreSQL is extensible to run user-supplied code in separate processes that are monitored by Postgres
  - Such processes can access PostgreSQL's shared memory area
  - Connect as a Client using libpq
- **bgworker: logical replication launcher**
  - Logical Replication between a Publisher and a Subscriber

Memory Components

- **Shared Buffers**:
  - PostgreSQL Database Memory Area
  - Shared by all the Databases in the Cluster
  - Pages are fetched from Disk to Shared Buffers during Reads/Writes
  - Modified Buffers are also called as Dirty Buffers
  - Parameter : *shared_buffers* sets the amount of RAM allocated to shared_buffers
  - Uses LRU Algorithm to flush less frequently used buffers
  - Dirty Buffers written to disk after a CHECKPOINT

- **WAL Buffers**:
  - Stores Write Ahead Log Records
  - Contains the change vector for a buffer being modified
  - WAL Buffers written to WAL Segments(On Disk)

- **work_mem**:
  - Memory used by each Query for internal sort operations such as ORDER BY and DISTINCT
  - Postgres writes to disk(temp files) if memory is not sufficient

Memory Components

- **maintenance_work_mem**:
  - Amount of RAM used by VACUUM, CREATE INDEX, REINDEX like maintenance operations
  - Setting this to a bigger value can help in faster database restore

PostgreSQL Does Not Use Direct IO

- When it needs a Page(Data Block), it searches it’s own memory aka Shared Buffers
- If not found in shared buffers, it will request the OS for the same block
- The OS fetches the block from the Disk and gives it to Postgres, if the block is not found in OS Cache
- More important to Caching when Database and Active Data set cannot fit in memory

Disk Components

- **Data Directory**
  - In MySQL, Data Directory is created when you initialize your MySQL Instance
  - Initialized using **initdb** in PostgreSQL. Similar to mysqld --initialize
  - Contains Write-Ahead-Logs, Log Files, Databases, Objects and other configuration files
  - You can move WAL's and Logs to different directories using symlinks and parameters
  - Environment Variable: $PGDATA

- **Configuration Files inside the Data Directory**
  - postgresql.conf (Similar to my.cnf file for MySQL)
  - Contains several configurable parameters
  - pg_ident.conf
  - pg_hba.conf
  - postgresql.auto.conf

What's Inside Data Directory?

```
avi@percona:~ $ls -l $PGDATA
total 48
drwx------. 5 postgres postgres    41 Oct 30 13:54 base
drwx------. 2 postgres postgres  4096 Oct 30 13:54 global
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_commit_ts
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_dynshmem
-rw-------. 1 postgres postgres  4513 Oct 30 13:54 pg_hba.conf
-rw-------. 1 postgres postgres  1636 Oct 30 13:54 pg_ident.conf
drwx------. 4 postgres postgres    68 Oct 30 13:54 pg_logical
drwx------. 4 postgres postgres    36 Oct 30 13:54 pg_multixact
drwx------. 2 postgres postgres    18 Oct 30 13:54 pg_notify
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_replslot
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_serial
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_snapshots
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_stat
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_stat_tmp
drwx------. 2 postgres postgres    18 Oct 30 13:54 pg_subtrans
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_tblspc
drwx------. 2 postgres postgres     6 Oct 30 13:54 pg_twophase
-rw-------. 1 postgres postgres     3 Oct 30 13:54 PG_VERSION
drwx------. 3 postgres postgres    60 Oct 30 13:54 pg_wal
drwx------. 2 postgres postgres    18 Oct 30 13:54 pg_xact
-rw-------. 1 postgres postgres    88 Oct 30 13:54 postgresql.auto.conf
-rw-------. 1 postgres postgres 23796 Oct 30 13:54 postgresql.conf
avi@percona:~ $
```

Configuration Files Inside Data Directory?

- PG_VERSION
  - Version String of the Database Cluster
- pg_hba.conf
  - Host-Based access control file (built-in firewall)
- pg_ident.conf
  - ident-based access file for OS User to DB User Mapping
- postgresql.conf
  - Primary Configuration File for the Database
- postmaster.opts
  - Contains the options used to start the PostgreSQL Instance
- postmaster.pid
  - The Parent Process ID or the Postmaster Process ID

postgresql.conf vs postgresql.auto.conf

- **postgresql.conf**
  - Configuration file for PostgreSQL similar to my.cnf for MySQL
  - This file contains all the parameters and the values required to run your PostgreSQL Instance
  - Parameters are set to their default values if no modification is done to this file manually
  - Located in the data directory or /etc depending on the distribution you choose and the location can be modifiable
- **postgresql.auto.conf**
  - PostgreSQL gives Oracle like compatibility to modify parameters using "ALTER SYSTEM"
  - Any parameter modified using ALTER SYSTEM is written to this file for persistence
  - This is last configuration file read by PostgreSQL, when started. Empty by default
  - Always located in the data directory

View/Modify Parameters in postgresql.conf

- Use show to view a value set to a parameter

  $ psql -c "show work_mem"

- To see all the settings, use show all

  $ psql -c "show all"

- Modifying a parameter value by manually editing the postgresql.conf file

  $ vi $PGDATA/postgresql.conf

- Use ALTER SYSTEM to modify a parameter

  $ psql -c "ALTER SYSTEM SET archive_mode TO ON"
  $ pg_ctl -D $PGDATA restart -mf

- Use reload using the following syntax to get the changes into effect for parameters not needing RESTART

  $ psql -c "select pg_reload_conf()"
  Or
  $ pg_ctl -D $PGDATA reload

Base Directory and Datafiles on Disk

- **Base Directory**
  - Contains Subdirectories for every Database you create
  - Every Database Sub-Directory contains files for every Relation/Object created in the Database
- **Datafiles**
  - Datafiles are the files for Relations in the base directory
  - Base Directory contains Relations
  - Relations stored on Disk as 1GB segments
  - Each 1GB Datafile is made up of several 8KB Pages that are allocated as needed
  - Segments are automatically added unlike Oracle

Base Directory (Database)

1. Create a database with name as: percona

   $ psql -c "CREATE DATABASE percona"

2. Get the datid for the database and see if it exists in the base directory

   $ psql -c "select datid, datname from pg_stat_database where datname = 'percona'"

```
postgres=# CREATE DATABASE percona;
CREATE DATABASE
postgres=# select datid, datname from pg_stat_database where datname = 'percona';
 datid | datname
-------+---------
 16384 | percona
(1 row)

postgres=# \q
avi@percona:~ $ls -ld $PGDATA/base/16384
drwx------. 2 postgres postgres 8192 Oct 30 18:53 /var/lib/pgsql/11/data/base/16384
avi@percona:~ $
```

Base Directory (Schema and Relations)

1. Create a schema named: scott

   $ psql -d percona -c "CREATE SCHEMA scott"

2. Create a table named: employee in scott schema

   $ psql -d percona -c "CREATE TABLE scott.employee(id int PRIMARY KEY, name varchar(20))"

3. Locate the file created for the table: scott.employee in the base directory

   $ psql -d percona -c "select pg_relation_filepath('scott.employee')"

```
avi@percona:~ $psql -d percona -c "CREATE SCHEMA scott"
CREATE SCHEMA
avi@percona:~ $psql -d percona -c "CREATE TABLE scott.employee (id int PRIMARY KEY, name varchar(20))"
CREATE TABLE
avi@percona:~ $psql -d percona -c "select pg_relation_filepath('scott.employee')"
 pg_relation_filepath
----------------------
 base/16384/16386
(1 row)
```

Base Directory (Block Size)

1. Check the size of the table in the OS and value of parameter: block_size

   $ psql -c "show block_size"

2. INSERT a record in the table and see the size difference

   $ psql -d percona -c "INSERT INTO scott.employee VALUES (1, 'frankfurt')"

3. INSERT more records and check the size difference

   $ psql -d percona -c "INSERT INTO scott.employee VALUES (generate_series(2,1000), 'junk')"

```
avi@percona:~ $psql -c "show block_size"
 block_size
------------
 8192
(1 row)

avi@percona:~ $ls -lh $PGDATA/base/16384/16386
-rw-------. 1 postgres postgres 0 Oct 30 18:59 /var/lib/pgsql/11/data/base/16384/16386
avi@percona:~ $psql -d percona -c "INSERT INTO scott.employee VALUES (1, 'frankfurt')"
INSERT 0 1
avi@percona:~ $ls -lh $PGDATA/base/16384/16386
-rw-------. 1 postgres postgres 8.0K Oct 30 20:47 /var/lib/pgsql/11/data/base/16384/16386
```

Write-Ahead Logs (WAL)

- **WALs**
  - When Client commits a transaction, it is written to WAL Segments (on Disk) before a success message is sent to Client
  - Transaction Journal aka REDO Logs. Similar to InnoDB Buffers in MySQL
  - Written by WAL Writer background process
  - Ensures Durability with fsync and synchronous_commit set to ON and commit_delay set to 0
  - Used during Crash Recovery
  - Size of each WAL is 16MB. Modifiable during Initialization
  - Created in **pg_xlog** directory until PostgreSQL 9.6Location of WALs is renamed to **pg_wal** from PostgreSQL 10
  - WAL Directory exits in Data Directory by default. Can be modified using Symlinks
  - WALs are deleted depending on the parameters : **wal_keep_segments and checkpoint_timeout**

WAL Archiving?

- **Archived WALs**
  - WALs in pg_wal or pg_xlog are gone after a certain threshold. Archiving ensures recoverability and helps a Slave catch-up during replication lag
  - Archiving in PostgreSQL can be enabled through parameters : archive_mode and archive_command
  - Ships WALs to safe locations like a Backup Server or Cloud Storage like S3 or Object Store
  - WALs are archived by archiver background process
  - archive_command can be set with the appropriate shell command to archive WALs
- Lets enable Archiving now …

  $ psql

  ALTER SYSTEM SET archive_mode TO 'ON';

  ALTER SYSTEM SET archive_command TO 'cp %p /var/lib/pgsql/archive/%f';

  $ pg_ctl -D $PGDATA restart -mf

Switch a WAL

- Switch a WAL and see if the WAL is safely archived…

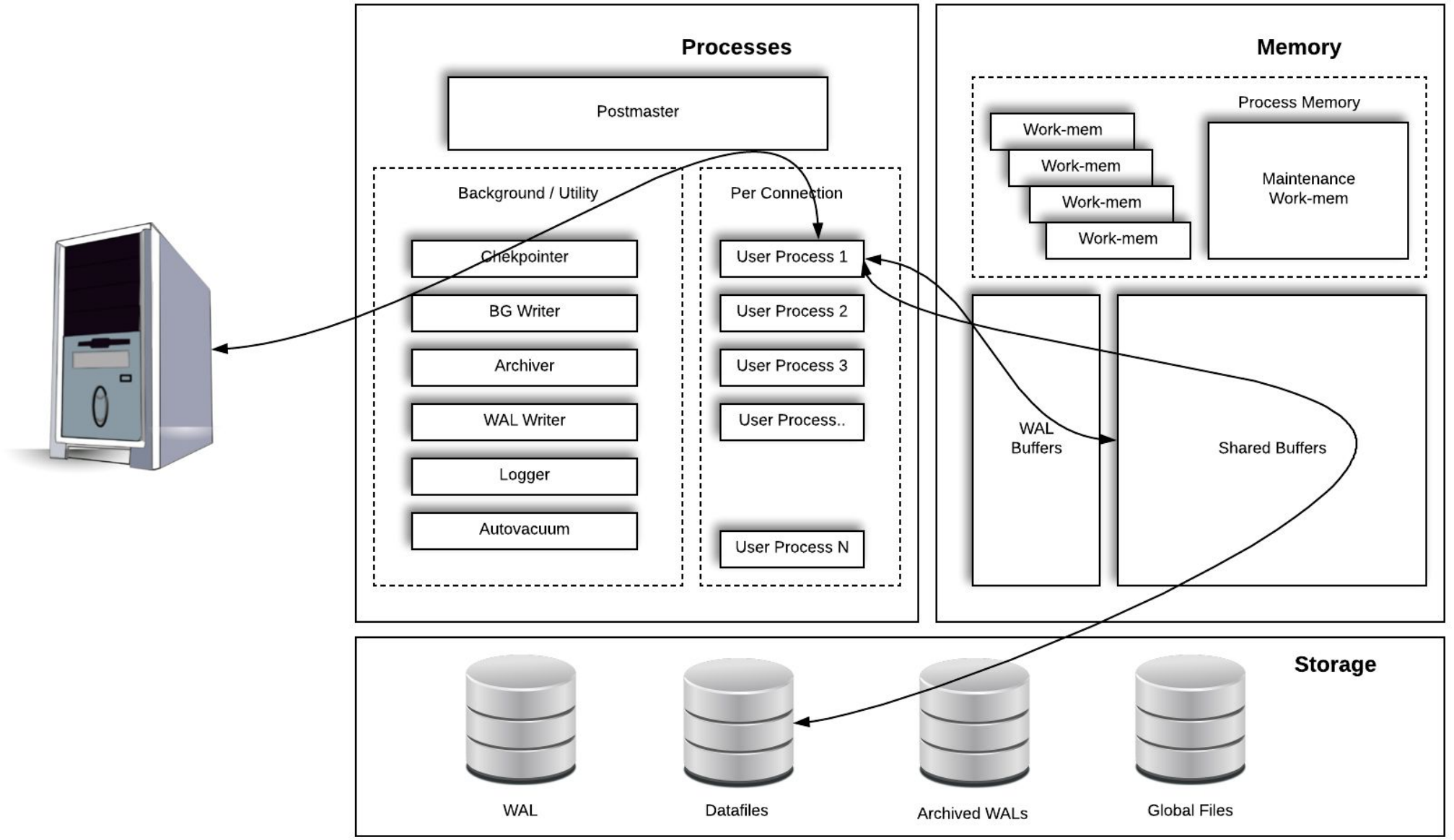
$ psql -c "select pg_switch_wal()"

```
avi@percona:~ $ls -l $PGDATA/pg_wal
total 32768
-rw-------. 1 postgres postgres 16777216 Oct 30 21:09 000000010000000000000001
-rw-------. 1 postgres postgres 16777216 Oct 30 21:09 000000010000000000000002
drwx------. 2 postgres postgres       43 Oct 30 21:09 archive_status
avi@percona:~ $ls -l /archive/
total 16384
-rw-------. 1 postgres postgres 16777216 Oct 30 21:09 000000010000000000000001
avi@percona:~ $
```

What if Archiving Failed?

If archiving has been enabled and the **archive_command** failed,

- the WAL segment for which the archiving failed will not be removed from pg_wal or pg_xlog
- an empty **wal_file_name.ready** file is generated in the **archive_status** directory
- the background process **archiver** attempts to archive the failed WAL segment until it succeeds
- there is a chance that the **pg_wal** directory can get filled and doesn't allow any more connections to database


Processes
Postmaster
Background / Utility
Chekpointer
BG Writer
Archiver
WAL Writer
Logger
Autovacuum
Per Connection
User Process 1
User Process 2
User Process 3
User Process..
User Process N
Memory
Process Memory
Work-mem
Work-mem
Work-mem
Work-mem
Maintenance
Work-mem
WAL
Buffers
Shared Buffers
Storage
WAL
Datafiles
Archived WALs
Global Files

Users and Roles in PostgreSQL

- Database users are different from Operating System users
- Users can be created in SQL using CREATE USER command or using the createuser utility
- Database users are common for all the databases that exists in a cluster
- Roles are created to segregate privileges for access control

Users and Roles in PostgreSQL - Demo

- Let us consider creating a read_only and a read_write role in database - percona
- A **read_only** Role that only has SELECT, USAGE privileges on Schema: percona
  - *CREATE ROLE scott_read_only;*
    GRANT SELECT ON ALL TABLES IN SCHEMA scott TO scott_read_only;
    GRANT USAGE ON SCHEMA scott TO scott_read_only;
- A **read_write** Role that only has SELECT, INSERT, UPDATE, DELETE privileges on Schema: percona
  - *CREATE ROLE scott_read_write;*
    GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA scott TO scott_read_write;
    GRANT USAGE ON SCHEMA scott TO scott_read_write;
- Create a User and assign either read_only or read_write role
  - *CREATE USER pguser WITH LOGIN ENCRYPTED PASSWORD 'pg123pass';*
    GRANT scott_read_only to pguser;

    ALTER USER pguser WITH CONNECTION LIMIT 20;

Backups in PostgreSQL

- PostgreSQL provides native backup tools for both Logical and Physical backups.
- Backups similar to mysqldump and Xtrabackup are automatically included with Community PostgreSQL
- Backups like RMAN in Oracle may be achieved using Open Source tools like pgBackRest and pgBarman
  - Logical Backups
    - **pg_dump** (Both Custom(Compressed and non human-readable) and Plain Backups)
    - **pg_restore** (To restore the custom backups taken using pg_dump)
    - **pg_dumpall** (To backup Globals - Users and Roles)
    - Logical Backups cannot be used to setup Replication and perform a PITR
    - You cannot apply WAL's after restoring a Backup taken using pg_dump
  - Physical Backups
    - **pg_basebackup** : File System Level & Online Backup, similar to Xtrabackup for MySQL
    - Useful to build Replication and perform PITR
    - This Backup can only use one process and cannot run in parallel
    - Explore Open Source Backup tools like : pgBackRest, pgBarman and WAL-e for more features like Xtrabackup

Logical Backup - Demo

- Let’s use pgbench to create some sample tables

  $ pgbench -i percona (Initialize)
  $ pgbench -T 10 -c 10 -j 2 percona (load some data)

- Use pg_dump to backup the DDL (schema-only) of database: percona

  $ pg_dump -s percona -f /tmp/percona_ddl.sql

- Use pg_dump to backup a table (with data) using custom and plain text format

  $ pg_dump -Fc —t public.pgbench_history -d percona -f /tmp/pgbench_history
  $ pg_dump -t public.pgbench_branches -d percona -f /tmp/pgbench_branches

- Create an another database and restore both the tables using pg_restore and psql

  $ psql -c "CREATE DATABASE testdb"
  $ pg_restore -t pgbench_history -d testdb /tmp/pgbench_history
  $ psql -d testdb -f /tmp/pgbench_branches

Globals Backup - pg_dumpall

- **pg_dumpall**
  - Can dump all the databases of a cluster into a script file
  - Use psql to restore the backup taken using pg_dumpall
  - Can be used to dump global objects such as ROLES and TABLESPACES
- To dump only Globals using pg_dumpall, use the following syntax

  $ pg_dumpall -g > /tmp/globals.sql
- To dump all databases (or entire Cluster), use the following syntax

  $ pg_dumpall > /tmp/globals.sql

Physical Backup - pg_basebackup

- Command line options for pg_basebackup

  $ pg_basebackup --help

  -D --> Target Location of Backup
  -cfast -—> Issues a fast checkpoint to start the backup earlier
  -Ft -—> Tar format. Use -Fp for plain
  -v --> Print the Backup statistics/progress.
  -U --> A User who has Replication Privilege.
  -W --> forcefully ask for password of replication User above. (Not mandatory)
  -z --> Compresses the Backup
  -R --> Creates a recovery.conf file that can be used to setup replication
  -P --> Shows the progress of the backup
  -l --> Creates a backup_label file

Full backup using pg_basebackup

- Run pg_basebackup now ….

  $ pg_basebackup -U postgres -p 5432 -h 127.0.0.1 -D /tmp/backup_11052018 -Ft -z -Xs -P -R -l backup_label

```
avi@percona:~ $pg_basebackup -U postgres -p 5432 -h 127.0.0.1 -D /tmp/backup_11052018 -Ft -z -Xs -P -R -l backup_label
58549/58549 kB (100%), 1/1 tablespace
avi@percona:~ $
avi@percona:~ $ls -l /tmp/backup_11052018
total 6428
-rw-------. 1 postgres postgres 6560306 Oct 31 02:35 base.tar.gz
-rw-------. 1 postgres postgres   17667 Oct 31 02:35 pg_wal.tar.gz
avi@percona:~ $
avi@percona:~ $tar -xzf /tmp/backup_11052018/base.tar.gz
avi@percona:~ $
avi@percona:~ $cat backup_label
START WAL LOCATION: 0/6000028 (file 000000010000000000000006)
CHECKPOINT LOCATION: 0/6000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2018-10-31 02:35:24 EDT
LABEL: backup_label
START TIMELINE: 1
avi@percona:~ $_
```

MVCC

Topics Being Discussed Under MVCC…

- UNDO Management
- Transaction ID’s and PostgreSQL hidden columns
- MVCC and how different is it from other RDBMS
- Why Autovacuum?
- Autovacuum settings
- Tuning Autovacuum

UNDO Management - Oracle and PostgreSQL

- Oracle and MySQL have separate storage for UNDO
  - May be limited space
  - ORA-01555 - Snapshot too old
  - ORA-30036: unable to extend segment by 8 in undo tablespace
  - Requires no special care to cleanup bloat
- PostgreSQL
  - Maintains UNDO within a table through versions - old and new row versions
  - Transaction ID’s are used to identify a version a query can use
  - A background process to delete old row versions explicitly
  - No additional writes to a separate UNDO storage in the event of writes
  - Row locks stored on tuple itself and no separate lock table

MVCC

- MVCC: Multi-Version Concurrency Control
- Data consistency
- Prevents viewing inconsistent data
- Readers and Writers do not block each other
- No Rollback segments for UNDO
- UNDO management is within tables
- A tuple contains the minimum and maximum transaction ids that are permitted to see it
- Just like SELECT statements executing WHERE
  xmin <= txid_current() AND (xmax = 0 OR txid_current() < xmax)

Transaction IDs in PostgreSQL

- Each transaction is allocated a transaction ID (txid)
- txid is a 32-bit unsigned integer
- 4.2 Billion (4,294,967,296) ID's
  - 2.1 Billion in the past are visible and
  - 2.1 Billion in the future are not visible
- ID's - 0, 1 and 2 are reserved

  0 - INVALID txid
  1 - Used in initialization of Cluster
  2 - Frozen txid

- txid is circular

Hidden Columns of a Table in PostgreSQL

```
[avi@percona:]$psql -d percona -c "SELECT attname, format_type (atttypid, atttypmod) \
FROM pg_attribute WHERE attrelid::regclass::text='foo.bar' \
ORDER BY attnum"
 attname  |      format_type
----------+-----------------------
 tableoid | oid
 cmax     | cid
 xmax     | xid
 cmin     | cid
 xmin     | xid
 ctid     | tid
 id       | integer
 name     | character varying(20)
(8 rows)
```

Hidden Columns - xmin and xmax

- xmin: Transaction ID that inserted the tuple
- xmax: txid of the transaction that issued an update/delete on this tuple and not committed yet
  or
  when the delete/update has been rolled back
  and 0 when nothing happened

```
[avi@percona:]$psql -d percona -c "CREATE TABLE foo.bar (id int, name varchar (20) )"
CREATE TABLE
[avi@percona:]$psql -d percona -c "select txid_current()"
 txid_current
--------------
         1603
(1 row)

[avi@percona:]$psql -d percona -c "INSERT INTO foo.bar VALUES (generate_series(1,10), 'avi')"
INSERT 0 10
[avi@percona:]$psql -d percona -c "select xmin, xmax, id, name from foo.bar"
 xmin | xmax | id | name
------+------+----+------
 1604 |    0 |  1 | avi
 1604 |    0 |  2 | avi
 1604 |    0 |  3 | avi
 1604 |    0 |  4 | avi
 1604 |    0 |  5 | avi
 1604 |    0 |  6 | avi
 1604 |    0 |  7 | avi
 1604 |    0 |  8 | avi
 1604 |    0 |  9 | avi
 1604 |    0 | 10 | avi
(10 rows)
```

Extension: pg_freespacemap

- PostgreSQL uses **FSM** to choose the page where a tuple can be inserted
- **FSM** stores free space information of each page
- Using the extension **pg_freespacemap**, we can see the freespace available inside each page of a table

```
percona=# CREATE EXTENSION pg_freespacemap;
CREATE EXTENSION
```

```
[avi@percona:]$ psql -d percona
psql (10.6)
Type "help" for help.

percona=# \x
Expanded display is on.
percona=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
FROM pg_freespace('foo.bar');
-[ RECORD 1 ]---+------
blkno           | 0
avail           | 7776
freespace ratio | 94.00
```

Delete a Record and See What Happens...

Session 1

```
[avi@percona:]$psql -d percona
psql (10.6)
Type "help" for help.

percona=# BEGIN;
BEGIN
percona=# DELETE FROM foo.bar WHERE id = 9;
DELETE 1
percona=#
```

Session 2

```
percona=# BEGIN ;
BEGIN
percona=# select xmin, xmax, id, name from foo.bar;
 xmin | xmax | id | name
------+------+----+------
 1604 |    0 |  1 | avi
 1604 |    0 |  2 | avi
 1604 |    0 |  3 | avi
 1604 |    0 |  4 | avi
 1604 |    0 |  5 | avi
 1604 |    0 |  6 | avi
 1604 |    0 |  7 | avi
 1604 |    0 |  8 | avi
 1604 | 1605 |  9 | avi
 1604 |    0 | 10 | avi
(10 rows)
```

Now COMMIT the DELETE and See...

Session 1

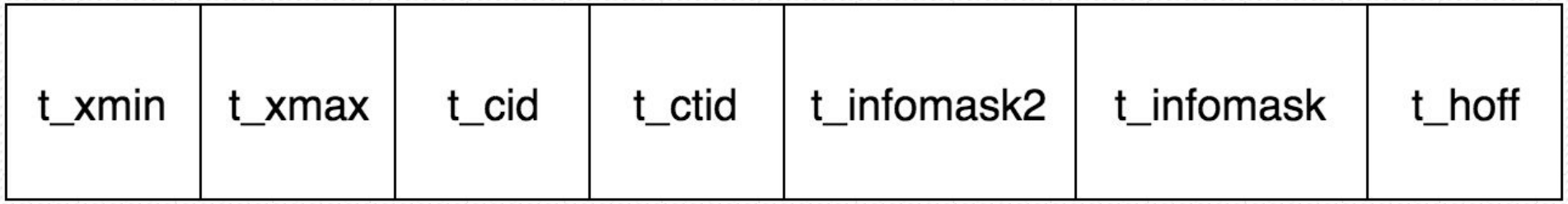
```
[avi@percona:]$psql -d percona
psql (10.6)
Type "help" for help.

percona=# BEGIN;
BEGIN
percona=# DELETE FROM foo.bar WHERE id = 9;
DELETE 1
percona=# COMMIT;
COMMIT
percona=#
```

Session 2

```
percona=# select xmin, xmax, id, name from foo.bar;
 xmin | xmax | id | name
------+------+----+------
 1604 |    0 |  1 | avi
 1604 |    0 |  2 | avi
 1604 |    0 |  3 | avi
 1604 |    0 |  4 | avi
 1604 |    0 |  5 | avi
 1604 |    0 |  6 | avi
 1604 |    0 |  7 | avi
 1604 |    0 |  8 | avi
 1604 |    0 | 10 | avi
(9 rows)
```

Heap Tuples

- Each Heap tuple in a table contains a HeapTupleHeaderData structure



HeapTupleHeaderData Structure

t_xmin: txid of the transaction that inserted this tuple

t_xmax: txid of the transaction that issued an update/delete on this tuple and not committed yet
or
when the delete/update has been rolled back.
and 0 when nothing happened.

t_cid: The position of the SQL command within a transaction that has inserted this tuple, starting from 0. If 5th command of transaction inserted this tuple, cid is set to 4

t_ctid: Contains the block number of the page and offset number of line pointer that points to the tuple

Extension: pageinspect

- Included with the contrib module
- Show the contents of a page/block
- 2 functions we could use to get tuple level metadata and data
  - **get_raw_page**: reads the specified 8KB block
  - **heap_page_item_attrs**: shows metadata and data of each tuple

- Create the Extension pageinspect

```
postgres=# CREATE EXTENSION pageinspect ;
CREATE EXTENSION
```

```
[avi@percona:]$ psql -d percona -c "\dt+ foo.bar"
                    List of relations
 Schema | Name | Type  |  Owner   |    Size    | Description
--------+------+-------+----------+------------+-------------
 foo    | bar  | table | postgres | 8192 bytes |
(1 row)
```

```
[avi@percona:]$psql -d percona -c "SELECT t_xmin, t_xmax, t_field3 as t_cid, t_ctid \
> FROM heap_page_items(get_raw_page('foo.bar', 0))"
 t_xmin | t_xmax | t_cid | t_ctid
--------+--------+-------+--------
   1604 |      0 |     0 | (0,1)
   1604 |      0 |     0 | (0,2)
   1604 |      0 |     0 | (0,3)
   1604 |      0 |     0 | (0,4)
   1604 |      0 |     0 | (0,5)
   1604 |      0 |     0 | (0,6)
   1604 |      0 |     0 | (0,7)
   1604 |      0 |     0 | (0,8)
   1604 |   1605 |     0 | (0,9)
   1604 |      0 |     0 | (0,10)
(10 rows)
```

```
percona=# SELECT lp,
       t_ctid AS ctid,
       t_xmin AS xmin,
       t_xmax AS xmax,
       (t_infomask & 128)::boolean AS xmax_is_lock,
       (t_infomask & 1024)::boolean AS xmax_committed,
       (t_infomask & 2048)::boolean AS xmax_rolled_back,
       (t_infomask & 4096)::boolean AS xmax_multixact,
       t_attrs[1] AS p_id,
       t_attrs[2] AS p_val
FROM heap_page_item_attrs(
        get_raw_page('foo.bar', 0),
        'foo.bar'
     ) where lp = 9;
-[ RECORD 1 ]----+-----------
lp               | 9
ctid             | (0,9)
xmin             | 1604
xmax             | 1605
xmax_is_lock     | f
xmax_committed   | t
xmax_rolled_back | f
xmax_multixact   | f
p_id             | \x09000000
p_val            | \x09617669
```

Delete a Record and Rollback...

SELECT sometimes a Write IO ?

Perform a select that sets the hint bits, after reading the commit log. It is an IO in fact :(

```
percona=# BEGIN;
BEGIN
percona=# DELETE FROM foo.bar WHERE id = 6;
DELETE 1
percona=# ROLLBACK;
ROLLBACK
```

```
percona=# select * from foo.bar where id = 6;
 id | name
----+------
  6 | avi
(1 row)
```

```
percona=# SELECT lp,
       t_ctid AS ctid,
       t_xmin AS xmin,
       t_xmax AS xmax,
       (t_infomask & 128)::boolean AS xmax_is_lock,
       (t_infomask & 1024)::boolean AS xmax_committed,
       (t_infomask & 2048)::boolean AS xmax_rolled_back,
       (t_infomask & 4096)::boolean AS xmax_multixact,
       t_attrs[1] AS p_id,
       t_attrs[2] AS p_val
FROM heap_page_item_attrs(
        get_raw_page('foo.bar', 0),
        'foo.bar'
     ) where lp = 6;
-[ RECORD 1 ]----+-----------
lp               | 6
ctid             | (0,6)
xmin             | 1604
xmax             | 1606
xmax_is_lock     | f
xmax_committed   | f
xmax_rolled_back | t
xmax_multixact   | f
p_id             | \x06000000
p_val            | \x09617669
```

Conclusion

- Just like SELECT statements executing
  WHERE xmin <= txid_current() AND (xmax = 0 OR txid_current() < xmax)

The above statement must be understandable by now…

Space Occupied by the DELETED Tuple?

VACUUM / AUTOVACUUM

- **Live Tuples**: Tuples that are Inserted or up-to-date or can be read or modified
- **Dead Tuples**: Tuples that are changed (Updated/Deleted) and unavailable to be used for any future transactions
- Continuous transactions may lead to a number of dead rows. A lot of space can be rather re-used by future transactions
- **VACUUM** in PostgreSQL would cleanup the dead tuples and mark it to free space map
- Transaction ID (**xmax**) of the deleting transaction must be older than the oldest transaction still active in PostgreSQL Server for vacuum to delete that tuple (i.e. xmax < oldest_active_txid)
- If xmax of a tuple is 100 and xact_committed = true and the oldest transaction id that is still active is 99, then vacuum cannot delete that tuple.
- **Autovacuum** in PostgreSQL automatically runs VACUUM on tables as a background process
- Autovacuum is also responsible to run **ANALYZE** that updates the statistics of a Table.

Background Processes in PostgreSQL

```
[avi@percona:]$ps -eaf | grep postgres
postgres 13532     1  0 Feb12 ?        00:00:00 /usr/pgsql-10/bin/postgres -D /var/lib/pgsql/10/data
postgres 13533 13532  0 Feb12 ?        00:00:00 postgres: logger process
postgres 13535 13532  0 Feb12 ?        00:00:00 postgres: checkpointer process
postgres 13536 13532  0 Feb12 ?        00:00:00 postgres: writer process
postgres 13537 13532  0 Feb12 ?        00:00:00 postgres: wal writer process
postgres 13538 13532  0 Feb12 ?        00:00:00 postgres: autovacuum launcher process
postgres 13539 13532  0 Feb12 ?        00:00:00 postgres: archiver process
postgres 13540 13532  0 Feb12 ?        00:00:01 postgres: stats collector process
postgres 13541 13532  0 Feb12 ?        00:00:00 postgres: bgworker: logical replication launcher
```

Let us Run a VACUUM and See Now…

```
[avi@percona:]$psql -d percona -c "VACUUM foo.bar"
VACUUM
```

```
[avi@percona:]$psql -d percona -c "SELECT t_xmin, t_xmax, t_field3 as t_cid, t_ctid \
FROM heap_page_items(get_raw_page('foo.bar', 0))"
 t_xmin | t_xmax | t_cid | t_ctid
--------+--------+-------+--------
   1604 |      0 |     0 | (0,1)
   1604 |      0 |     0 | (0,2)
   1604 |      0 |     0 | (0,3)
   1604 |      0 |     0 | (0,4)
   1604 |      0 |     0 | (0,5)
   1604 |   1606 |     0 | (0,6)
   1604 |      0 |     0 | (0,7)
   1604 |      0 |     0 | (0,8)
        |        |       |
   1604 |      0 |     0 | (0,10)
(10 rows)
```

Does it Show Some Extra Free Space in the Page Now?

Use pg_freespacemap Again...

```
percona=# \x
Expanded display is on.
percona=#
percona=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
FROM pg_freespace('foo.bar');
-[ RECORD 1 ]---+------
blkno           | 0
avail           | 7808
freespace ratio | 95.00
```

When Does Autovacuum Run?

Autovacuum

- To start **autovacuum**, you must have the parameter **autovacuum** set to ON
- Background Process : **Stats Collector** tracks the usage and activity information
- PostgreSQL identifies the tables needing vacuum or analyze depending on certain parameters
- Parameters needed to enable autovacuum in PostgreSQL are:
  autovacuum = on # (ON by default)
  track_counts = on # (ON by default)
- An automatic vacuum or analyze runs on a table depending on a certain mathematical equations

- **Autovacuum VACUUM**
  - Autovacuum VACUUM threshold for a table = **autovacuum_vacuum_scale_factor** * number of tuples + **autovacuum_vacuum_threshold**
  - If the actual number of dead tuples in a table exceeds this effective threshold, due to updates and deletes, that table becomes a candidate for autovacuum vacuum
- **Autovacuum ANALYZE**
  - Autovacuum ANALYZE threshold for a table = **autovacuum_analyze_scale_factor** * number of tuples + **autovacuum_analyze_threshold**
  - Any table with a total number of inserts/deletes/updates exceeding this threshold since last analyze is eligible for an autovacuum analyze

- **autovacuum_vacuum_scale_factor** or **autovacuum_analyze_scale_factor**: Fraction of the table records that will be added to the formula. For example, a value of 0.2 equals to 20% of the table records
- **autovacuum_vacuum_threshold** or **autovacuum_analyze_threshold**: Minimum number of obsolete records or dml's needed to trigger an autovacuum
- Let's consider a table: foo.bar with 1000 records and the following autovacuum parameters

  autovacuum_vacuum_scale_factor = 0.2
  autovacuum_vacuum_threshold = 50
  autovacuum_analyze_scale_factor = 0.1
  autovacuum_analyze_threshold = 50

- Table : foo.bar becomes a candidate for autovacuum VACUUM when,
  Total number of Obsolete records = (0.2 * 1000) + 50 = 250
- Table : foo.bar becomes a candidate for autovacuum ANALYZE when,
  Total number of Inserts/Deletes/Updates = (0.1 * 1000) + 50 = 150

Tuning Autovacuum in PostgreSQL

- Setting global parameters alone may not be appropriate, all the time
- Regardless of the table size, if the condition for autovacuum is reached, a table is eligible for autovacuum vacuum or analyze
- Consider 2 tables with ten records and a million records
- Frequency at which a vacuum or an analyze runs automatically could be greater for the table with just ten records
- Use table level autovacuum settings instead

  ALTER TABLE foo.bar SET (autovacuum_vacuum_scale_factor = 0, autovacuum_vacuum_threshold = 100);

- There cannot be more then **autovacuum_max_workers** number of auto vacuum processes running at a time. Default is 3
- Each autovacuum runs with a gap of **autovacuum_naptime**, default is 1 min

Can I Increase autovacuum_max_workers?
Is VACUUM IO Intensive?

- Autovacuum reads 8KB (default **block_size**) pages of a table from disk and modifies/writes to the pages containing dead tuples
- Involves both read and write IO and may be heavy on big tables with huge amount of dead tuples
- Autovacuum IO Parameters:

  autovacuum_vacuum_cost_limit: total cost limit autovacuum could reach (combined by all autovacuum jobs)

  autovacuum_vacuum_cost_delay: autovacuum will sleep for these many milliseconds when a cleanup reaching *autovacuum_vacuum_cost_limit* cost is done

  vacuum_cost_page_hit: Cost of reading a page that is already in shared buffers and doesn't need a disk read

  vacuum_cost_page_miss: Cost of fetching a page that is not in shared buffers

  vacuum_cost_page_dirty: Cost of writing to each page when dead tuples are found in it

- Default Values for the Autovacuum IO parameters

  autovacuum_vacuum_cost_limit = -1 (Defaults to vacuum_cost_limit) = 200
  autovacuum_vacuum_cost_delay = 20ms
  vacuum_cost_page_hit = 1
  vacuum_cost_page_miss = 10
  vacuum_cost_page_dirty = 20

- Let's imagine what can happen in 1 second. (1 second = 1000 milliseconds)

- In a best case scenario where read latency is 0 milliseconds, autovacuum can wake up and go for sleep 50 times (1000 milliseconds / 20 ms) because the delay between wake-ups needs to be 20 milliseconds.

  1 second = 1000 milliseconds = 50 * **autovacuum_vacuum_cost_delay**

- **Read IO limitations with default parameters**
  - If all the pages with dead tuples are found in shared buffers, in every wake up 200 pages can be read Cost associated per reading a page in shared_buffers is 1

    So, in 1 second, (50 * 200/vacuum_cost_page_hit * 8 KB) = **78.13 MB** can be read by autovacuum
  - If the pages are not in shared buffers and need to fetched from disk, an autovacuum can read: 50 * ((200 / vacuum_cost_page_miss) * 8) KB = **7.81 MB** per second
- **Write IO limitations with default parameters**
  - To delete dead tuples from a page/block, the cost of a write operation is : vacuum_cost_page_dirty, set to 20 by default
  - At the most, an autovacuum can write/dirty : 50 * ((200 / vacuum_cost_page_dirty) * 8) KB = **3.9 MB** per second

Transaction ID Wraparound

- 4.2 Billion (4,294,967,296) ID's
- 2.1 Billion in the past are visible
- 2.1 Billion in the future are not visible

- Transaction with txid:= n, inserted a record
  t_xmin := n
- After some time, we are now at a txid := (2.1 billion + n)
  Tuple is visible to a SELECT now. (Because it is still 2.1 Billionth transaction in the past)
- Now let us say that the txid is:= (2.1 billion + n + 1). The same SELECT fails as the txid:= n is now considered to be the future.
- This is usually referred to as: Transaction ID Wraparound in PostgreSQL
- Vacuum in PostgreSQL re-writes the t_xmin to the frozen txid when the t_xmin is older than (current txid - vacuum_freeze_min_age)
- Until 9.3, xmin used to be updated with an invalid and visible txid : 2, upon FREEZE
- Starting from 9.4, the XMIN_FROZEN bit is set to the t_infomask field of tuples and avoids re-writing the tuples

Best Strategy

- Do not just add more autovacuum workers. See if you are fine for more IO caused by autovacuum and tune all the IO settings
- Busy OLTP systems require your thorough supervision for automation of manual vacuum
- Perform routine manual vacuum in low peak or non-business hours to ensure a less bloated database at all times
- A database with finely tuned autovacuum settings and routine maintenance tasks is always healthy

Tablespaces in PostgreSQL

Tablespaces in PostgreSQL

- Tablespaces
  - Can be used to move Table and Indexes to different disks/locations
  - Helps distributing IO
- Steps to create tablespace in PostgreSQL
- Step 1: Create a directory for the tablespace
  - **$ mkdir -p /tmp/tblspc_1**
    $ chown postgres:postgres /tmp/tblspc_1
    $ chmod 700 /tmp/tblspc_1
- Step 2: Create tablespace using the new directory
  - **$ psql -c "CREATE TABLESPACE tblspc_1 LOCATION '/tmp/tblspc_1'"**
- Step 3: Create a table in the new table-space
  - **$ psql -d percona -c "CREATE TABLE scott.foo (id int) TABLESPACE tblspc_1"**

PostgreSQL Partitioning

Partitioning in PostgreSQL

- **Partitioning until PostgreSQL 9.6**
  - PostgreSQL supported Partitioning via Table Inheritance
  - CHECK Constraints and Trigger Functions to redirect data to appropriate CHILD Tables
  - Supports both RANGE and LIST Partitioning
- **Declarative Partitioning since PostgreSQL 10 (Oracle and MySQL like Syntax)**
  - Avoid the trigger based Partitioning and makes it easy and faster
  - Uses internal C Functions instead of PostgreSQL Triggers
  - Supports both RANGE and LIST Partitioning
- **Advanced Partitioning from PostgreSQL 11**
  - Supports default partitions
  - Hash Partitions
  - Parallel Partition scans
  - Foreign Keys
  - Optimizer Partition elimination, etc

Declarative Partitioning in PostgreSQL

- **Create a table and partition by RANGE**

```
CREATE TABLE scott.orders (id INT, order_time TIMESTAMP WITH TIME ZONE, description TEXT)
PARTITION BY RANGE (order_time);

ALTER TABLE scott.orders ADD PRIMARY KEY (id, order_time);

CREATE TABLE scott.order_2018_01_04 PARTITION OF scott.orders
    FOR VALUES FROM ('2018-01-01') TO ('2018-05-01');

CREATE TABLE scott.order_2018_05_08 PARTITION OF scott.orders
    FOR VALUES FROM ('2018-05-01') TO ('2018-09-01');

CREATE TABLE scott.order_2018_09_12 PARTITION OF scott.orders
    FOR VALUES FROM ('2018-09-01') TO ('2019-01-01');
```

- **Insert values to the table**

```
INSERT INTO scott.orders (id, order_time, description)
SELECT random() * 6, order_time, md5(order_time::text)
FROM generate_series('2018-01-01'::date, CURRENT_TIMESTAMP, '1 hour') as order_time;
```

```
percona=# \d+ scott.orders
                                        Table "scott.orders"
   Column    |           Type           | Collation | Nullable | Default | Storage  | Stats target | Description
-------------+--------------------------+-----------+----------+---------+----------+--------------+-------------
 id          | integer                  |           | not null |         | plain    |              |
 order_time  | timestamp with time zone |           | not null |         | plain    |              |
 description | text                     |           |          |         | extended |              |
Partition key: RANGE (order_time)
Indexes:
    "orders_pkey" PRIMARY KEY, btree (id, order_time)
Partitions: scott.order_2018_01_04 FOR VALUES FROM ('2018-01-01 00:00:00-05') TO ('2018-05-01 00:00:00-04'),
            scott.order_2018_05_08 FOR VALUES FROM ('2018-05-01 00:00:00-04') TO ('2018-09-01 00:00:00-04'),
            scott.order_2018_09_12 FOR VALUES FROM ('2018-09-01 00:00:00-04') TO ('2019-01-01 00:00:00-05')

percona=# INSERT INTO scott.orders (id, order_time, description)
percona-# SELECT random() * 6, order_time, md5(order_time::text)
percona-# FROM generate_series('2018-01-01'::date, CURRENT_TIMESTAMP, '1 hour') as order_time;
INSERT 0 7283
percona=# _
```

EXPLAIN Plan on Partitioned Table

- Use EXPLAIN to see the Execution Plan of the following SELECT statement

```
EXPLAIN SELECT id, order_time, description
        FROM scott.orders
        WHERE order_time between '2018-05-22 02:00:00' and '2018-05-28 02:00:00';
```

- Create Indexes on Partition Keys to ensure optimal performance

```
CREATE INDEX order_idx_2018_01_04 ON scott.order_2018_01_04 (order_time);
CREATE INDEX order_idx_2018_05_08 ON scott.order_2018_05_08 (order_time);
CREATE INDEX order_idx_2018_09_12 ON scott.order_2018_09_12 (order_time);
```

EXPLAIN - Before and After Indexes

- Before

```
percona=# EXPLAIN SELECT id, order_time, description
FROM scott.orders
WHERE order_time between '2018-05-22 02:00:00' and '2018-05-28 02:00:00';
                                                                         QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------
 Append  (cost=0.00..76.00 rows=145 width=45)
   ->  Seq Scan on order_2018_05_08  (cost=0.00..75.28 rows=145 width=45)
         Filter: ((order_time >= '2018-05-22 02:00:00-04'::timestamp with time zone) AND (order_time <= '2018-05-28 02:00:00-04'::timestamp with time zone))
(3 rows)
```

- After

```
percona=# EXPLAIN SELECT id, order_time, description
FROM scott.orders
WHERE order_time between '2018-05-22 02:00:00' and '2018-05-28 02:00:00';
                                                                         QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------------------
---
 Append  (cost=0.28..6.91 rows=145 width=45)
   ->  Index Scan using order_idx_2018_05_08 on order_2018_05_08  (cost=0.28..6.18 rows=145 width=45)
         Index Cond: ((order_time >= '2018-05-22 02:00:00-04'::timestamp with time zone) AND (order_time <= '2018-05-28 02:00:00-04'::timestamp with time zone
))
(3 rows)
```

High Availability in PostgreSQL

- **Streaming Replication for PostgreSQL 9.x and above**
  - WAL Segments are streamed to Standby/Slave and replayed on Slave
  - Not a Statement/Row/Mixed Replication like MySQL
  - This can be referred to as a byte-by-byte or Storage Level Replication
  - Slaves are always Open for Read-Only SQLs but not Writes
  - You cannot have different Schema or Data in a Master and a Slave in Streaming Replication
  - Allows Cascading Replication
  - Supports both Synchronous and Asynchronous Replication
  - Supports a Delayed Standby for faster PITR

- **Logical Replication and Logical Decoding for PostgreSQL 10 and above**
  - Allows for Replication of selected Tables using Publisher and Subscriber Model
  - Similar to binlog_do_db in MySQL, but no DDL Changes are replicated
  - Subscribers are also open for Writes automatically
  - Used in Data Warehouse environments that stores Data fetched from multiple OLTP Databases for Reporting, etc

PostgreSQL Streaming Replication

- Step 1: Create a user in Master with REPLICATION ROLE

```
CREATE USER replicator
    WITH REPLICATION
    ENCRYPTED PASSWORD 'replicator';
```

- Step 2: Parameters you should know while setting up SR

archive_mode: Must be set to ON to enable Archiving of WALs
wal_level: Must be set to "hot_standy" until 9.5 and "replica" in the later versions.
max_wal_senders: Must be set to 3 if you are starting with 1 Slave. For every Slave, you may add 2 wal senders.
wal_keep_segments: Number of WALs always retained in pg_xlog (Until PostgreSQL 9.6) or pg_wal (From PostgreSQL 10)
archive_command: This parameter takes a shell command. It can be a simple copy command to copy the WAL segments to another location or a Script that has the logic to archive the WALs to S3 or a remote Backup Server.
hot_standby: Must be set to ON on Standby/Replica and has no effect on the Master. However, when you setup your Replication, parameters set on Master are automatically copied. This parameter is important to enable READS on Slave. Else, you cannot run your SELECTS on Slave.

Streaming Replication (Cont.d)

- Step 3: Set the parameters that are not set already

```
ALTER SYSTEM SET wal_keep_segments TO '50';
select pg_reload_conf();
```

- Step 4: Add an entry to pg_hba.conf of Master to allow Replication connections from Slave
  Default location of pg_hba.conf is the Data Directory

```
$ vi pg_hba.conf
```

Add the following line between >>>>> and <<<<<< to the end of the pg_hba.conf file

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
host replication replicator 192.168.0.28/32 md5
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

Replace the IP address(192.168.0.28) with your Slave IP address

- Step 5: Give a SIGHUP or RELOAD

```
$ pg_ctl -D $PGDATA reload
```

Streaming Replication (Cont.d)

- Step 6: Use pg_basebackup to backup of your Master data directory to the Slave data directory

  $ pg_basebackup -U replicator -p 5432 -D /tmp/slave -Fp -Xs -P -R

- Step 7: Change the port number of your slave if you are creating the replication in the same server for demo

  $ echo "port = 5433" >> /tmp/slave/postgresql.auto.conf

- Step 8: Start your Slave

  $ pg_ctl -D /tmp/slave start

- Step 9: Check the replication status from Master using the view : pg_stat_replication

  select * from pg_stat_replication;

Failover to Slave/Standby

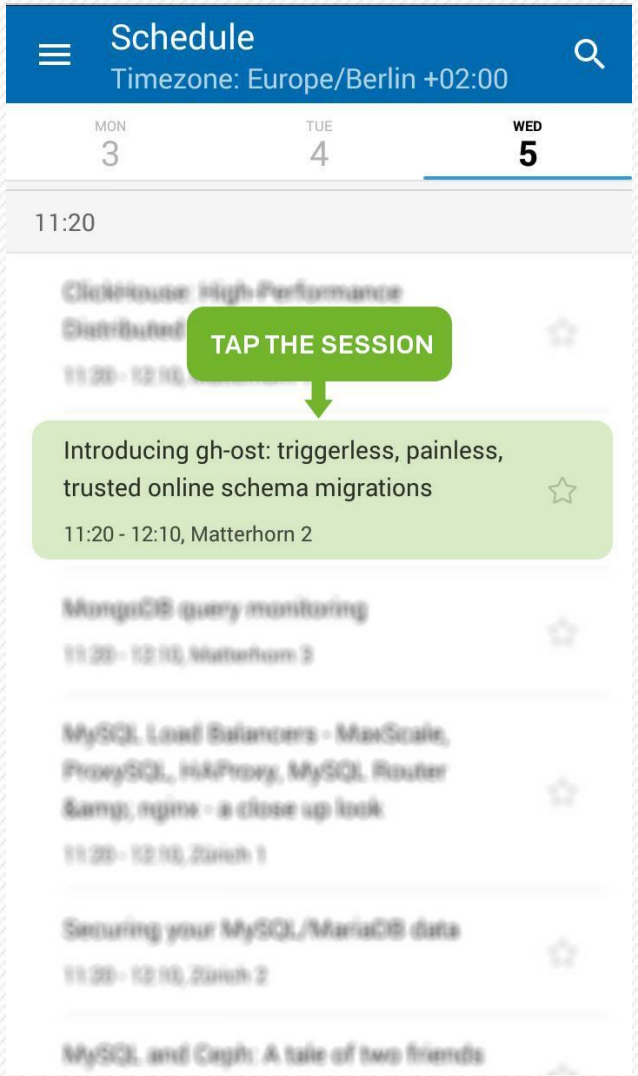
- PostgreSQL databases do not allow any direct writes to be sent to their replicas/slaves, when configured using Streaming Replication.
- Thus, a slave needs to be explicitly promoted in order to allow writes to it
- Run the following command to perform a failover

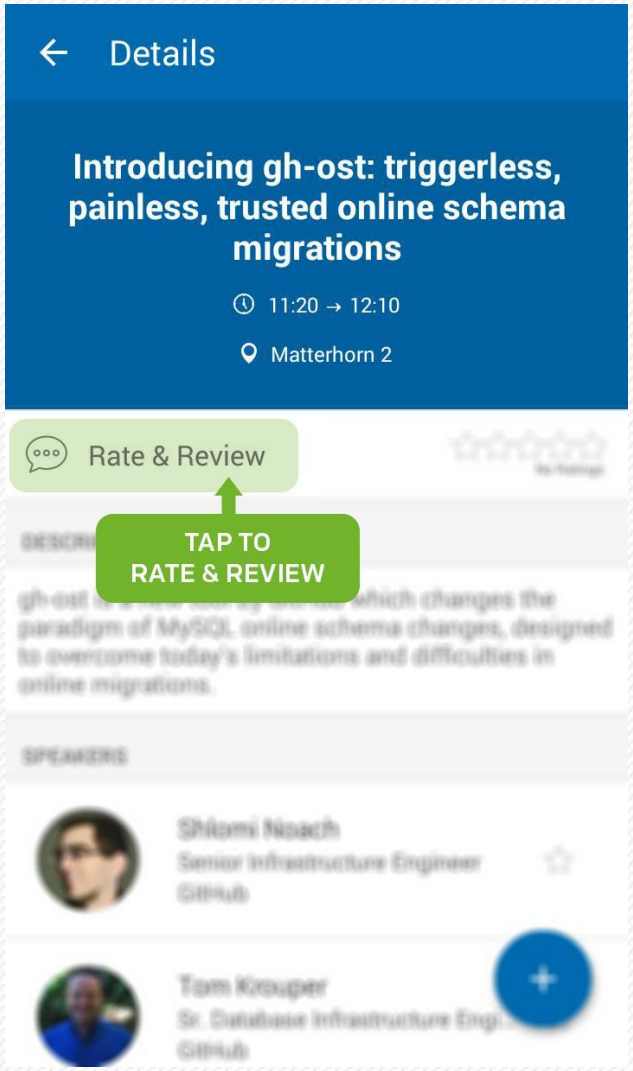
  $ pg_ctl -D $PGDATA promote

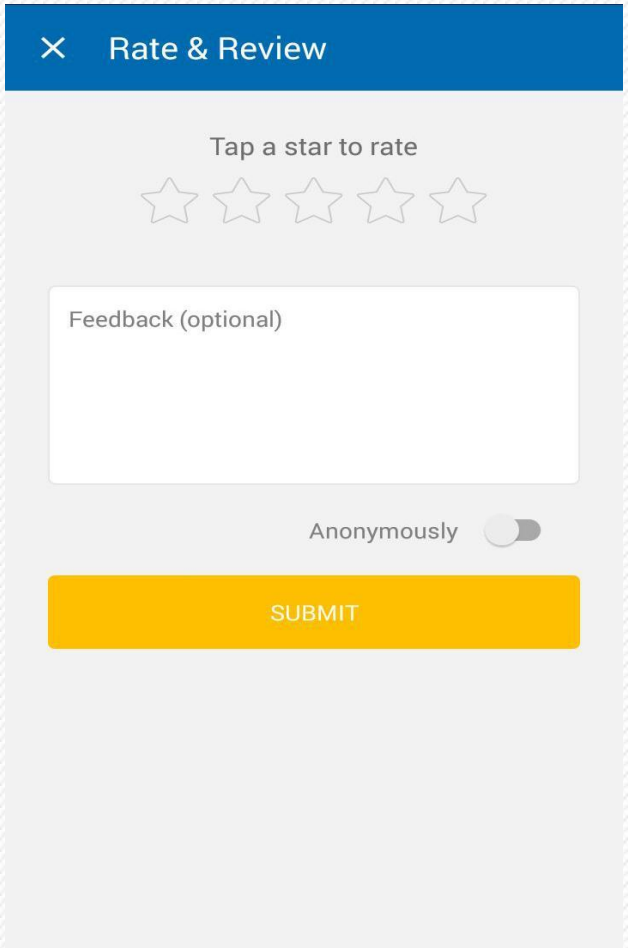
- Promote happens real quick with almost no impact to the already existing connections on slave

Thank You to Our Sponsors

Rate My Session







Any Questions?