Point in Time Recovery

Fanavaran Anisa
Iran Linux House

Linux & Open Source Training Center

www.anisa.co.ir

Point-in-time Recovery

PostgreSQL has the ability to restore the database to any point in history following the **previous (full) backup**. This is called **point in time recovery (PITR)**. It does this by keeping files called transaction logs.

The PostgreSQL server records all users' data modification transaction like insert, update or delete and write it into a file call **write-ahead (WAL) log file**. This mechanism use the history records stored in WAL file to do roll-forward changes made since last database **full backup**.

PITR – First Step: Backup

- Modify postgresql.conf to support archive log
- Make a base backup (full database backup)
- Backup base backup to remote storage. (optional)
- Backup WAL (archive log files) to remote storage (continuous process) (optional)

```
wal_level = replica
archive_mode = on
archive_command = 'test ! -f /archive/%f && cp %p /archive/%f'  # Unix

archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

PITR – Second Step: Recover(Restore)

- Extract files from base backup & copy into **Data** folder
- Move WAL files from **Archive** folder into **pg_wal** folder(Remove any files present in pg_wal)
- Add Recovery Setting into **postgres.conf** file
- Create the **recovery.signal** file
- Start Recover (Start Postgres)

```
archive_mode = off
restore_command = 'cp /archive/%f %p’
recovery_target_timeline = 'latest'
```

Recovery Targets

```
recovery_target = 'immediate'

recovery_target_time = '2023-04-14 12:54:23'

recovery_target_lsn = '0/2000060'

recovery_target_xid = 569865

recovery_target_timeline = 'latest'

recovery_target_timeline = 2

recovery_target_name = '1402-Bahman-30'
```

This parameter specifies the named restore point (created with `pg_create_restore_point()`) to which recovery will proceed.

```
recovery_target_inclusive = true
```

Recovery Target Action

`recovery_target_action:`

- pause: The default behavior. Recovery will be paused when the recovery target is reached.

- promote: The recovery process will finish, and the server will start to accept connections. This effectively promotes the server from a standby state to a primary state. The server will stop after reaching the recovery target.

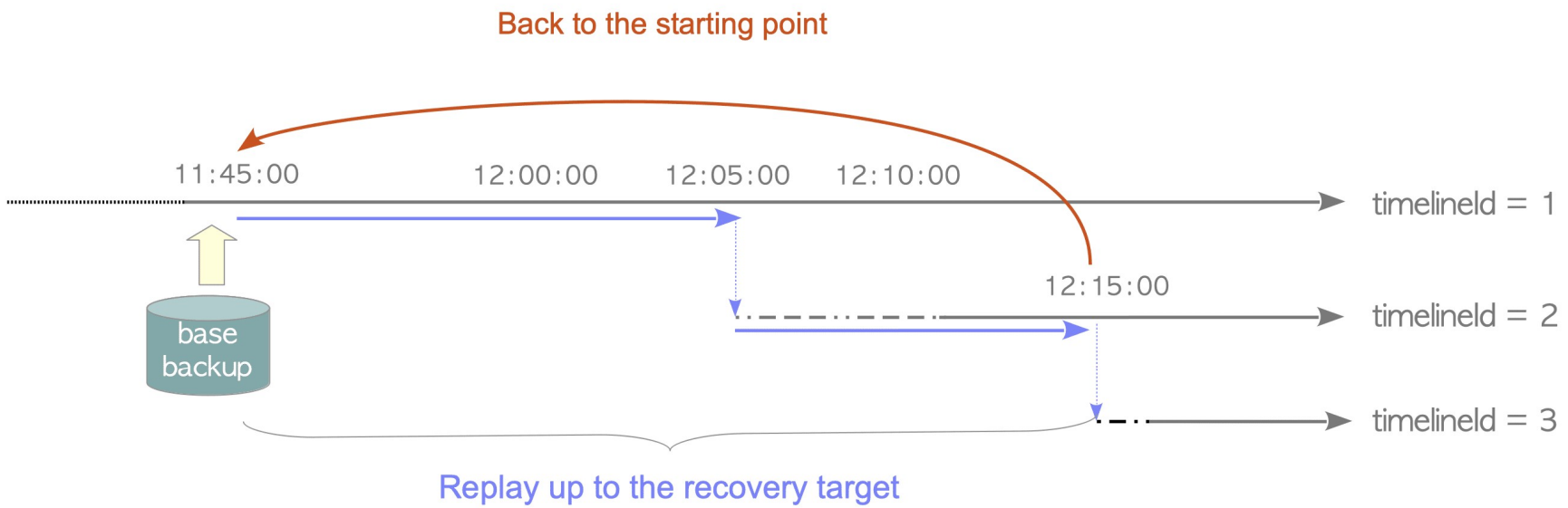
- shutdown: The server will stop after reaching the recovery target.

Upon completion of the recovery process, the server will remove `recovery.signal`

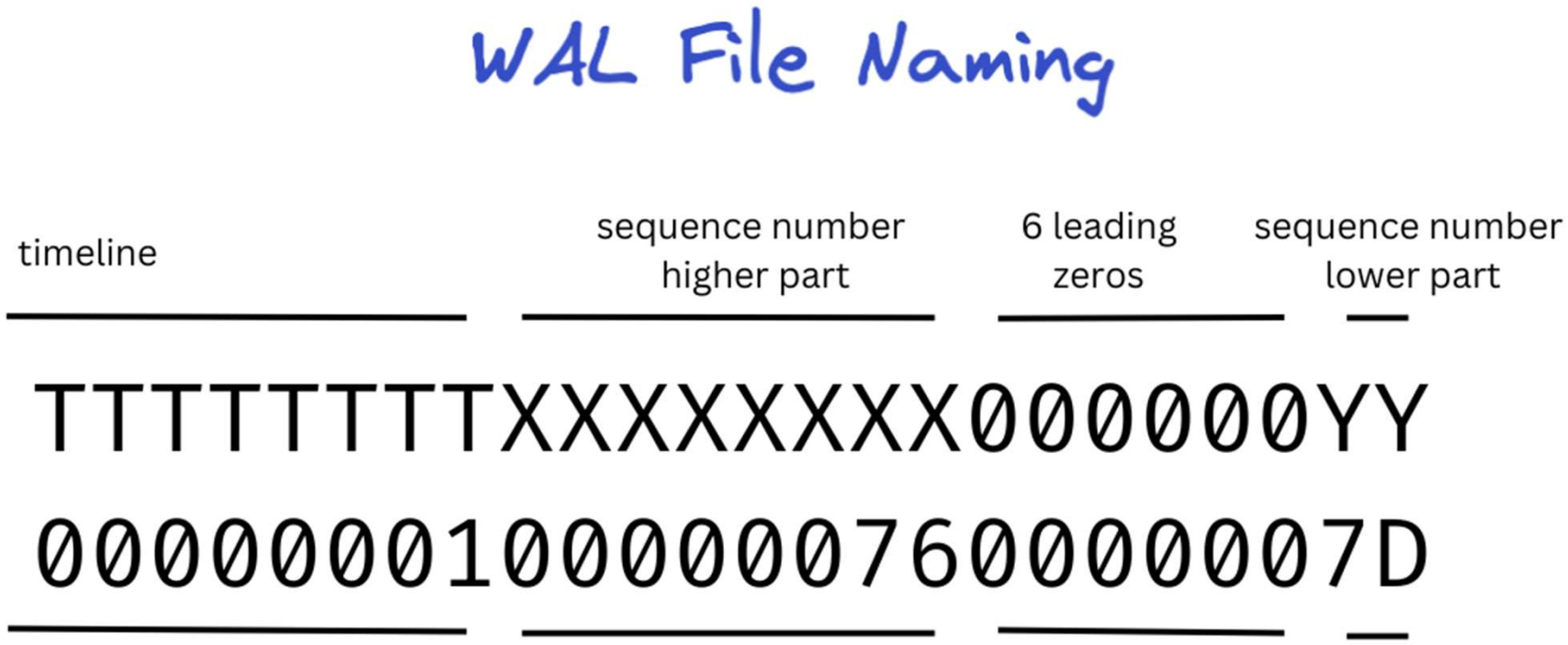
Timeline Incrementation

Whenever an **archive recovery completes**, a new timeline is **created to identify the series of WAL records generated after that recovery**.
The timeline ID number is part of WAL segment file names so a new timeline does not overwrite the WAL data generated by previous timelines.

For example, in the WAL file name 000000010000012340000055CD, the leading 00000001 is the timeline ID in hexadecimal.
a new timeline history file named '00000002?.history' is created in the pg_wal subdirectory (pg_xlog if versions 9.6 or earlier) and the archival directory.



WAL File Naming



Wal File Name in PSQL/ Switch WAL

[postgres] # select pg_walfile_name('76/7D000028');

```
 pg_walfile_name
 --------------------------
000000010000007600000007D
(1 row)
```

`pg_switch_wal()` is a system function which forces PostgreSQL to switch to a new WAL file. (if current WAL is not empty)

```
postgres=# SELECT pg_switch_wal();
 pg_switch_wal
---------------
 0/161F710
(1 row)
```

Log Sequence Numbers (LSN)

The LSN is a 64-bit integer, representing a position in the write-ahead log stream.

This 64-bit integer is split into two segments (high 32 bits and low 32 bits).

It is printed as two hexadecimal numbers separated by a slash (XXXXXXXX/YYZZZZZZ).

- The 'X' represents the high 32-bits of the LSN
- ‘Y’ is the high 8 bits of the lower 32-bits section.
- The 'Z' represents the offset position in the file.

Each element is a hexadecimal number. The 'X' and 'Y' values are used in the second part of the WAL file on a default PostgreSQL deployment.

LSN Related Functions

The **`pg_current_wal_lsn`** is the location of the last write.
The **`pg_current_wal_insert_lsn`** is the logical location and reflects data in the buffer that has not been written to disk.
There is also a flush value that shows what has been written to durable storage.

```
[postgres] # select pg_current_wal_lsn(), pg_current_wal_insert_lsn();

 pg_current_wal_lsn | pg_current_wal_insert_lsn
--------------------+--------------------------
 76/7D000000        | 76/7D000028
(1 row)
```

LSN Related Functions – A Quick Sample

```
postgres] # select pg_current_wal_lsn(), pg_current_wal_insert_lsn();
 pg_current_wal_lsn | pg_current_wal_insert_lsn
 -------------------+--------------------------
76/7E000060 | 76/7E000060
(1 row)

[postgres] # insert into test (a) values ('a');
INSERT 0 1
[postgres] # select pg_current_wal_lsn(), pg_current_wal_insert_lsn();
 pg_current_wal_lsn | pg_current_wal_insert_lsn
-------------------+--------------------------
76/7E000108 | 76/7E000108

[postgres] # select '76/7E000108'::pg_lsn - '76/7E000060'::pg_lsn size_bytes;
 size_bytes
------------
168
(1 row)
```

pg_waldump

We use **pg_waldump** to get a human readable summary of the WAL segment contents.
In the following command the starting position (-s) and ending position (-e) are specified along with the WAL file name

```
$ pg_waldump -s 76/7E000060 -e 76/7E000108 000000010000000760000007E
rmgr: Heap len (rec/tot): 57/ 57, tx: 59555584, lsn: 76/7E000060, prev 76/7E000028, desc:
INSERT+INIT off 1 flags 0x08, blkref #0: rel 1663/5/53434 blk 0
rmgr: Transaction len (rec/tot): 46/ 46, tx: 59555584, lsn: 76/7E0000A0, prev 76/7E000060,
desc: COMMIT 2023-02-13 16:25:19.441483 EST
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 76/7E0000D0, prev 76/7E0000A0, desc:
RUNNING_XACTS nextXid 59555585 latestCompletedXid 59555584 oldestRunningXid
59555585
```

Workshop Time!