24 Dec 2023

Relational DB Backup and Restore

Point-in-Time Recovery (PITR) allows a database administrator to restore or recover a set of data from a backup from a particular time in the past, using a tool or a system. Once PITR starts logging a database, the administrator can then restore the database backup from a respective time. PITR is also considered an additional method of data protection, as it safeguards loss of important information.

**What is a Point-in-Time (PIT) Snapshot?**

The set of data logged by PITR is known as the Point-in-Time (PIT) snapshot. A point-in-time snapshot is a copy of data, files, or databases at a point in time. PIT is useful if there is a need to recover and restore data from a specific time or the last recent PIT snapshot.

PIT snapshots are usually ready only and can be kept updated either by mapping a recent snapshot with the earlier copy or only copy data that has been changed will be copied again.

**What is Database Point-In-Time Recovery (PITR)?**

PITR is important when someone has accidentally deleted a table or records in a database or if something has gone wrong which has corrupted the existing database. The fastest solution to this would be to retrieve the transaction logs and recover the database from the last “known good” point. PITR is done differently for individual databases.

**PITR in PostgreSQL Using Timelines**

PITR for PostgreSQL requires backing up live database files and archiving the Write Ahead Log (WAL) files. A WAL file logs any transactions like updates, deletes, and creation done to the database. If you want to restore the state of a database to a point in time, the WAL file segment can be used.

However, once you have used it and the WAL has the latest changes, rolling back to the earlier changes before PITR would not be possible. To avoid this, the series of WAL files created after the PITR should be isolated from the original database.

PostgreSQL solves this problem using timelines. After every successful recovery, a timeline is created to store the series of WAL records created after the recovery. These WAL segments have a unique timeline ID to make sure there is no override of previous WAL data. This is useful especially when you are not very sure which archive to use and allows you to test and see which point-in-time data is the best for recovery.

PostgreSQL also maintains a timeline history where it keeps a file to track which timeline is branched and when. These small text files are also archived together with the WAL files segment and come in handy when picking the right WAL from an archive.

**PITR in MySQL Using Binary Log**

Like the other two databases, MySQL also uses a log to record the operations done to the database. PITR uses the set of incremental backups of binary log files after a full backup operation.

To restore the binary log, you need to know the name and location of the current binary log. PITR for MySQL can be done using event time or event position. To use the event time option, you need to specify the ‘start date and time’, and ‘end date and time’ options for mysqlbinlog.

This is a good option if you need to restore the table and data from a specific time and date. PITR using event position is done by specifying the log positions instead of date and time. This method is precise, as it allows you to roll back accurately, especially if many transactions were done at the time of damage.

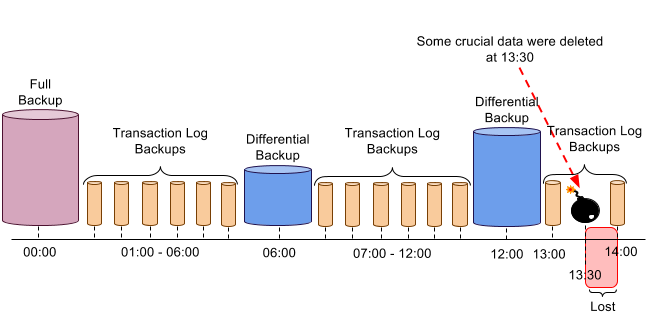
For this option, you can use mysqlbinlog to get the log positions and pass in the time range of the unwanted transaction. This will create a text file with the SQL statements during the damage and retrieve the log positions of the SQL statement causing the damage. Restore the last backup, and then use the log positions to process the binary log.

**Conclusion**

PITR is another method for data protection and a quick recovery plan for the recovery process during a disaster. The prerequisite for PITR would be the backup process needs to have a log enabled to have all database transactions recorded.

Based on the summary of the three databases mentioned above, two components are important for Point-in-Time Recovery; one being the base backup and the other is the transactions logged in the log files. The log files help to either restore a good transaction or eliminate bad ones like accidentally deleting a table or updates done to a large amount of data via SQL statements.

Point-in-time recovery has some disadvantages the database will be unavailable during the recovery process and if the base backups are located elsewhere it is going to be a time-consuming process to restore all data files together with the transaction logs. However, PITR can bring the database back in time with minimal data loss.



**PostgreSQL Backup and Point-in-Time Recovery Strategy**

**Backup Mechanisms:**

**1. Full Backups:**

Perform regular full backups of the entire PostgreSQL database using the **pg\_dump** utility. Full backups provide a comprehensive snapshot of the database.

**pg\_dump -U <username> -h <hostname> -d <database\_name> -f /path/to/full\_backup\_2023-01-01.sql**

**2. Base Backup (Optional but recommended for large databases):**

Periodically take base backups using **pg\_basebackup** to create a consistent snapshot of the entire database cluster.

**pg\_basebackup -U <username> -h <hostname> -D /path/to/base\_backup\_2023-01-02 -Ft -Xs -P**

**3. Continuous Archiving (WAL-based Incremental Backups):**

Enable continuous archiving of Write-Ahead Logs (WAL) to capture incremental changes for point-in-time recovery.

Edit **postgresql.conf**:

**wal\_level = replica**

**archive\_mode = on**

**archive\_command = 'cp %p /path/to/archive/%f'**

**Timeline for Backups:**

* **Day 1:**
  + Full backup (**full\_backup\_2023-01-01.sql**).
* **Day 2:**
  + Base backup (**base\_backup\_2023-01-02.tar**) in addition to continuous archiving.
* **Subsequent Days:**
  + Regularly archive WAL segments.

**Point-in-Time Recovery:**

**1. Restore Full Backup:**

Restore the most recent full backup using **psql**.

**psql -U <username> -h <hostname> -d postgres -f /path/to/full\_backup\_2023-01-01.sql**

**2. Apply Base Backup:**

Apply the base backup created on Day 2.

**pg\_basebackup -U <username> -h <hostname> -D /path/to/data -Ft -Xs -P**

**3. Apply WAL Segments to Point of Failure:**

Copy archived WAL segments to the restoration location.

**cp /path/to/archive/\* /path/to/restore\_location/pg\_wal/**

**4. Create recovery.conf File:**

Use **pg\_waldump** to generate a recovery configuration file.

**pg\_waldump /path/to/archive/0000000100000001000000A3 > recovery.conf**

Edit **recovery.conf**:

**restore\_command = 'cp /path/to/archive/%f %p'**

**recovery\_target\_time = '2023-01-03 14:30:00'**

**5. Start PostgreSQL in Recovery Mode:**

Start PostgreSQL in recovery mode using the restored data directory.

**postgres -D /path/to/data**

This process will apply the base backup and subsequent WAL segments until it reaches the specified recovery target time.