Chapter 10

Base Backup & Point-in-Time Recovery

Online database backup can be roughly classified into two categories: logical and physical backups. While both have advantages and disadvantages, one disadvantage of logical backups is that they can be very time-consuming. In particular, it can take a long time to make a backup of a large database, and even longer to restore the database from the backup data. On the other hand, physical backups can be made and restored much more quickly, making them a very important and useful feature in practical systems.

In PostgreSQL, online physical full backups have been available since version 8.0. A snapshot of a running whole database cluster (i.e., physical backup data) is known as a **base backup**.

Point-in-Time Recovery (PITR), which has also been available since version 8.0, is the feature to restore a database cluster to any point in time using a *base backup* and *archive logs* created by the continuous archiving feature. For example, if you make a critical mistake (such as truncating all tables), this feature can be used to restore the database to the point just before the mistake was made.

In this chapter, following topics are described:

- What is a base backup?
- How doed PITR work?
- What is a timelineId?
- What is a timeline history file?

In versions 7.4 or earlier, PostgreSQL had supported only logical backups (logical full and partial backups, and data exports).

10.1. Base Backup

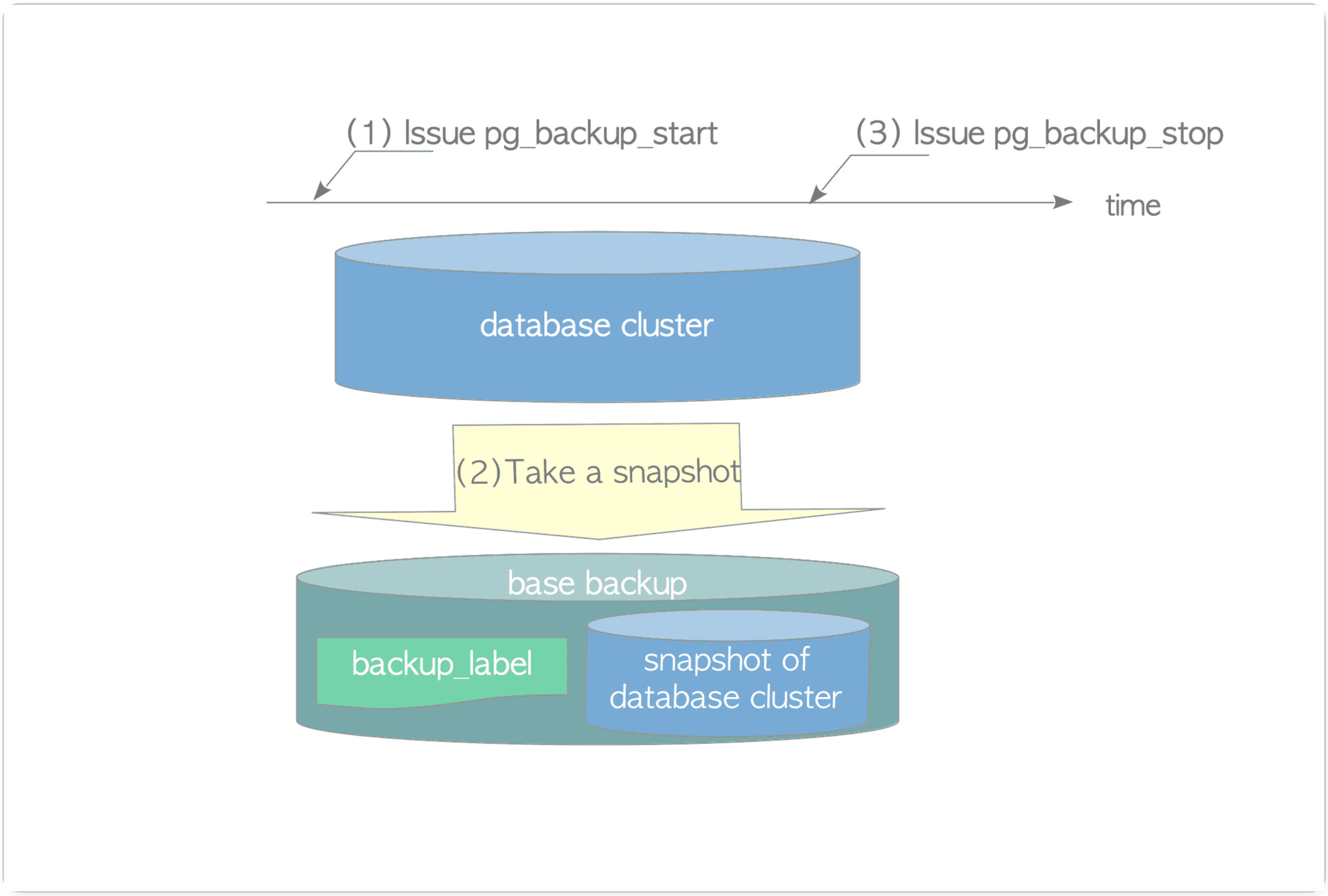
First of all, the standard procedure to make a base backup using the low-level commands is as follows:

(1) Issue the *pg_backup_start* command (versions 14 or earlier, *pg_start_backup*).
(2) Take a snapshot of the database cluster using the archiving command of your choice.
(3) Issue the *pg_backup_stop* command (versions 14 or earlier, *pg_stop_backup*).

This simple procedure is easy for database administrators to use because it requires no special tools other than common tools such as the cp command or a similar archiving tool. In addition, this procedure does not require table locks, so all users can continue to issue queries without being affected by the backup operation. This is a significant advantage over other major open source RDBMSs.

A simpler way to make a base backup is to use the *pg_basebackup* utility, which internally issues the low-level commands described above.

Fig. 10.1. Making a base backup.



Because the pg_backup_start and pg_backup_stop commands are so important to understanding PITR, we will explore them in more detail in the following subsections.

The *pg_backup_start* and *pg_backup_stop* commands are defined here: src/backend/access/transam/xlogfuncs.c.

10.1.1. pg_backup_start (Ver.14 or earlier, pg_start_backup)

The *pg_backup_start* command prepares for making a base backup. As discussed in Section 9.8, the recovery process starts from a REDO point, so the *pg_backup_start* command must do a checkpoint to explicitly create a REDO point at the start of making a base backup. Moreover, the checkpoint location of its checkpoint must be saved in a file other than pg_control because regular checkpoints might be done a number of times during the backup. Therefore, the *pg_backup_start* performs the following four operations:

1. Force the database into full-page wirte mode.
2. Switch to the current WAL segment file (versions 8.4 or later).
3. Do a checkpoint.
4. Create a *backup_label file* – This file, created in the top level of the base directory, contains essential information about base backup itself, such as the checkpoint location of this checkpoint.

The third and fourth operations are the heart of this command. The first and second operations are performed to recover a database cluster more reliably.

A *backup_label* file contains the following six items (versions 11 or later, seven items):

- CHECKPOINT LOCATION – This is the LSN location where the checkpoint created by this command has been recorded.
- START WAL LOCATION – This is **not** used with PITR, but used with the streaming replication, which is described in Chapter 11. It is named 'START WAL LOCATION' because the standby server in replication-mode reads this value only once at initial startup.
- BACKUP METHOD – This is the method used to make this base backup.
- BACKUP FROM – This shows whether this backup is taken from the primary or standby server.
- START TIME – This is the timestamp when the *pg_backup_start* command was executed.
- LABEL – This is the label specified at the *pg_backup_start* command.
- START TIMELINE – This is the timeline that the backup started. This is for a sanity check and has been introduced in version 11.

backup_label

An actual example of a backup_label file in version 16, which is taken by using pg_basebackup, is shown below:

```
postgres> cat /usr/local/pgsql/data/backup_label
START WAL LOCATION: 0/1B000028 (file 00000001000000000000001B)
CHECKPOINT LOCATION: 0/1B000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-10-26 11:45:19 GMT
LABEL: pg_basebackup base backup
START TIMELINE: 1
```

As you may imagine, when you recover a database using this base backup, PostgreSQL takes the 'CHECKPOINT LOCATION' from the backup_label file to read the checkpoint record from the appropriate archive log. It then gets the REDO point from the record and starts the recovery process. (The details will be described in the next section.)

10.1.2. pg_backup_stop (Ver.14 or earlier, pg_stop_backup)

The *pg_backup_stop* command performs the following five operations to complete the backup:

1. Reset to *non-full-page writes* mode if it has been forcibly changed by the *pg_backup_start* command.
2. Write a XLOG record of backup end.

3. Switch the WAL segment file.
4. Create a *backup history file*. This file contains the contents of the *backup_label* file and the timestamp that the *pg_backup_stop* command was executed.
5. Delete the backup_label file. The backup_label file is required for recovery from the base backup, but once copied, it is not necessary in the original database cluster.

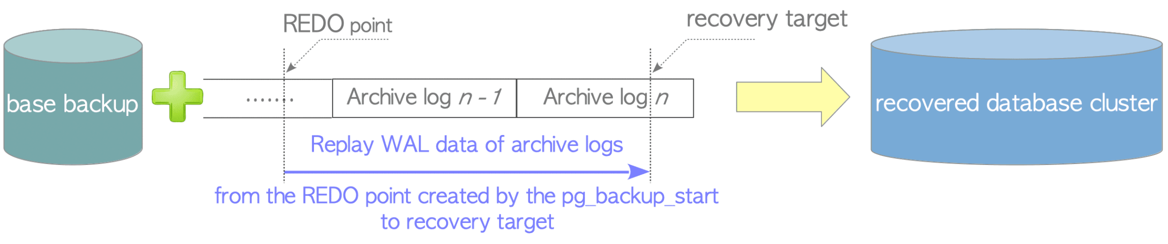
The *naming method for backup history file* is shown below.

```
{WAL segment}.{offset value at the time the base backup was started}.backup
```

10.2. How Point-in-Time Recovery Works

Figure 10.2 shows the basic concept of PITR. In PITR mode, PostgreSQL replays the WAL data of the archive logs on the base backup, from the REDO point created by the *pg_backup_start* up to the point you want to recover. In PostgreSQL, the point to be recovered is referred to as a **recovery target**.

Fig. 10.2. Basic concept of PITR.


REDO point
recovery target
base backup
Archive log n - 1
Archive log n
recovered database cluster
Replay WAL data of archive logs
from the REDO point created by the pg_backup_start
to recovery target

Here is the description of how PITR works.

Suppose that you made a mistake at 12:05 GMT of 26 October, 2023. You should remove the database cluster and restore the new one using the base backup you made before that.

At first, you need to set the command of the *restore_command* parameter, and also set the time of the *recovery_target_time* parameter to the point you made the mistake (in this case, 12:05 GMT) in a postgresql.conf (versions 12 or later) or recovery.conf (versions 11 or earlier).

```
# Place archive logs under /mnt/server/archivedir directory.
restore_command = 'cp /mnt/server/archivedir/%f %p'
recovery_target_time = "2023-10-26 12:05 GMT"
```

When PostgreSQL starts up, it enters into PITR mode if there is a *recovery.signal*(versions 12 or later) or a *recovery.conf* (versions 11 or earlier), and a *backup_label* in the database cluster.

recovery.conf

The *recovery.conf* file has been abolished in version 12, and all recovery-related parameters should be written in postgresql.conf. See the official document in detail.

In versions 12 or later, when you restore your server from a basebackup, you need to create an empty file called **recovery.signal** in the database cluster directory.

```
$ touch /usr/local/pgsql/data/recovery.signal
```

The PITR (Point-in-Time Recovery) process is almost the same as the normal recovery process described in Chapter 9. The only differences are:

1. Where are WAL segments/Archive logs read from?
   - Normal recovery mode – from the pg_wal subdirectory (in versions 9.6 or earlier, pg_xlog subdirectory) under the base directory.
   - PITR mode – from an archival directory set in the configuration parameter archive_command.
2. Where is the checkpoint location read from?
   - Normal recovery mode – from the *pg_control file*.
   - PITR mode – from a *backup_label file*.

The outline of PITR process is as follows:

(1) PostgreSQL reads the value of *'CHECKPOINT LOCATION'* from the backup_label file using the internal function *read_backup_label* to find the REDO point.
(2) PostgreSQL reads some values of parameters from the postgresql.conf (versions 12 or later) or recovery.conf (versions 11 or earlier), such as *restore_command* and *recovery_target_time*.
(3) PostgreSQL starts replaying WAL data from the REDO point, which can be easily obtained from the value of *'CHECKPOINT LOCATION'*. The WAL data are read from archive logs that are copied from the archival area to a temporary area by executing the command written in the *restore_command* parameter. (The copied log files in the temporary area are removed after use.)
In this example, PostgreSQL reads and replays WAL data from the REDO point to the one before the timestamp '2023-10-26 12:05:00' because the *recovery_target_time* parameter is set to this timestamp. If a recovery target is not set to the postgresql.conf (versions 12 or later) or recovery.conf (versions 11 or earlier), PostgreSQL will replay until the end of the archiving logs.
(4) When the recovery process completes, a **timeline history file**, such as '00000002.history', is created in the pg_wal subdirectory (in versions 9.6 or earlier, pg_xlog subdirectory). If archiving log feature is enabled, the same named file is also created in the archival directory. The contents and role of this file are described in the following sections.

The records of commit and abort actions contain the timestamp at which each action has done (XLOG data portion of both actions are defined in xl_xact_commit and xl_xact_abort respectively). Therefore, if a target time is set to the parameter *recovery_target_time*, PostgreSQL may select whether to continue recovery or not, whenever it replays XLOG record of either commit or abort action. When XLOG record of each action is replayed, PostgreSQL compares the target time and each timestamp written in the record; and if the timestamp exceed the target time, PITR process will be finished.

The function *read_backup_label* is defined in src/backend/access/transam/xlog.c.
The structure *xl_xact_commit* and *xl_xact_abort* are defined in src/include/access/xact.h.

Why can we use common archiving tools to make a base backup?

The recovery process is a process of restoring a database cluster to a consistent state, even if the cluster is inconsistent. PITR is based on the recovery process, so it can recover a database cluster even if the base backup is a bunch of inconsistent files. This is why we can use common archiving tools without the need for a file system snapshot capability or a special tool.

10.3. timelineId and Timeline History File

A **timeline** in PostgreSQL is used to distinguish between the original database cluster and the recovered ones. It is a central concept of PITR. In this section, two things associated with the timeline are described: *timelineId* and *timeline history files*.

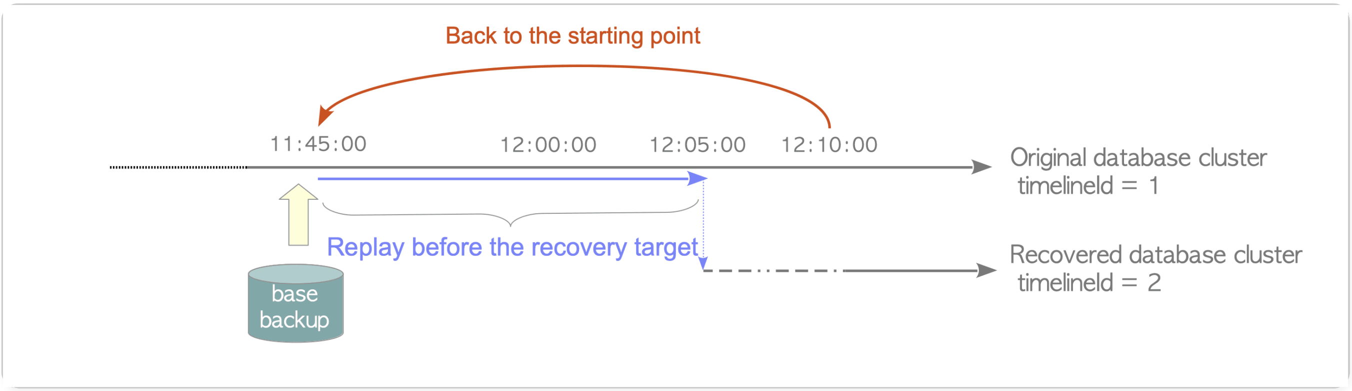
10.3.1. timelineId

Each timeline is given a corresponding **timelineId**, a 4-byte unsigned integer starting at 1.

An individual timelineId is assigned to each database cluster. The timelineId of the original database cluster created by the initdb utility is 1. Whenever a database cluster recovers, the timelineId is increased by 1. For example, in the example of the previous section, the timelineId of the cluster recovered from the original one is 2.

Figure 10.3 illustrates the PITR process from the viewpoint of the timelineId. First, we remove our current database cluster and restore the base backup made in the past, in order to go back to the starting point of recovery. This situation is represented by the red arrow curve in the figure. Next, we start the PostgreSQL server, which replays WAL data in the archive logs from the REDO point created by the *pg_backup_start* until the recovery target by tracing along the initial timeline (timelineId 1). This situation is represented by the blue arrow line in the figure. Then, a new timelineId 2 is assigned to the recovered database cluster and PostgreSQL runs on the new timeline.

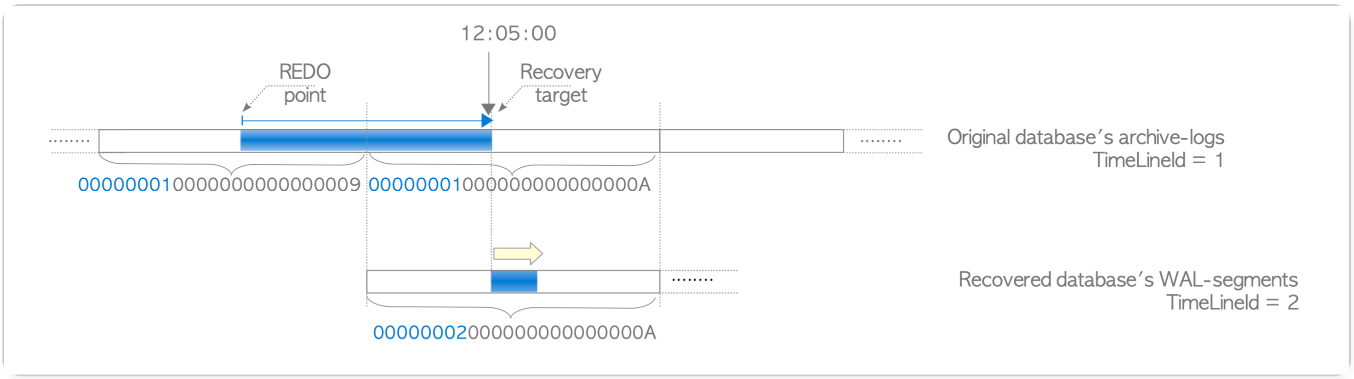
Fig. 10.3. Relation of timelineId between an original and a recovered database clusters.


Back to the starting point
11:45:00
12:00:00
12:05:00
12:10:00
Original database cluster timelineId = 1
Replay before the recovery target
Recovered database cluster timelineId = 2
base backup

As briefly mentioned in Chapter 9, the first 8 digits of the WAL segment filename are equal to the timelineId of the database cluster that created the segment. When the timelineId is changed, the WAL segment filename will also be changed.

Focusing on WAL segment files, the recovery process can be described again. Suppose that we recover the database cluster using two archive logs '000000010000000000000009' and '00000001000000000000000A'. The newly recovered database cluster is assigned the timelineId 2, and PostgreSQL creates the WAL segment from '00000002000000000000000A'. Figure 10.4 shows this situation.

Fig. 10.4. Relation of WAL segment files between an original and a recovered database clusters.



10.3.2. Timeline History File

When a PITR process completes, a timeline history file with names like '00000002.history' is created under the archival directory and the pg_xlog subdirectory (in versions 10 or later, pg_wal subdirectory). This file records which timeline it branched off from and when.

The naming rule of this file is shown below:

```
"8-digit new timelineId".history
```

The timeline history file contains at least one line, and each line is composed of the following three items:

- timelineId – The timelineId of the archive logs used to recover.
- LSN – The LSN location where the WAL segment switches happened.
- reason – A human-readable explanation of why the timeline was changed.

A specific example is shown below:

```
postgres> cat /home/postgres/archivelogs/00000002.history
1          0/A000198     before 2023-10-26 12:05:00.861324+00
```

Meaning as follows:

> The database cluster (timelineId=2) is based on the base backup whose timelineId is *1*, and is recovered in the time just *before '2023-10-26 12:05:00.861324+00*' by replaying the archive logs until the *0/A000198*.

In this way, each timeline history file tells us a complete history of the individual recovered database cluster. Moreover, it is also used in the PITR process itself. The details are explained in the next section.

ℹ

The timeline history file format is changed in version 9.3. The formats of versions 9.3 or later and earlier are shown below, but not in detail.

Later version 9.3:

```
timelineId  LSN   "reason"
```

Until version 9.2:

```
timelineId  WAL_segment    "reason"
```

10.4. Point-in-Time Recovery with Timeline History File

The timeline history file plays an important role in the second and subsequent PITR processes. By trying a second time recovery, we will explore how it is used.

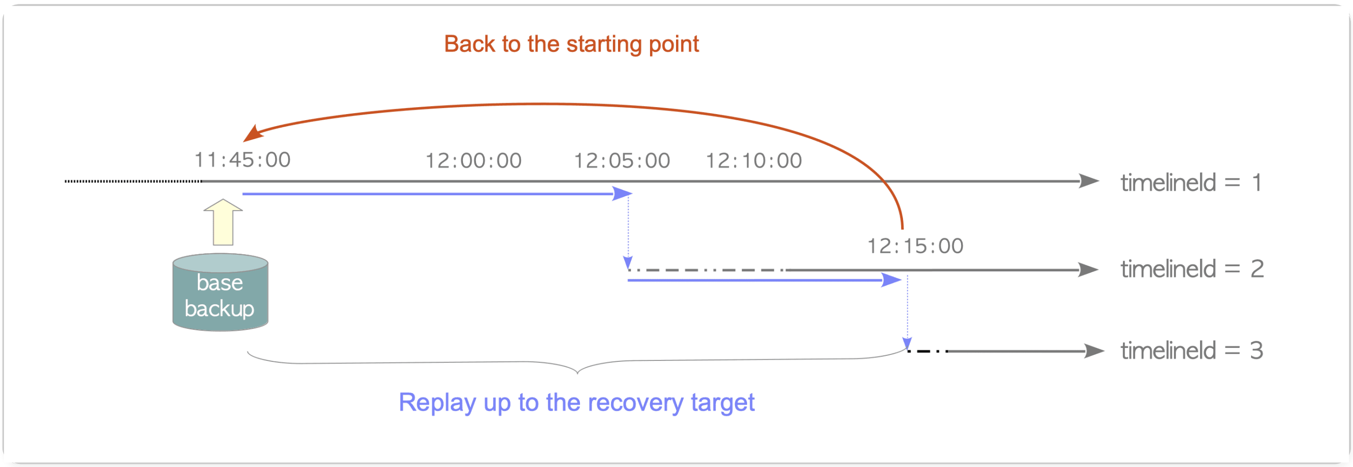
Again, suppose that you made a mistake at 12:15:00 in the recovered database cluster whose timelineId is 2. In this case, to recover the database cluster, you should create a new recovery.conf file as shown below:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
recovery_target_time = "2023-10-26 12:15:00 GMT"
recovery_target_timeline = 2
```

The parameter *recovery_target_time* sets the time you made the new mistake, and the parameter *recovery_target_timeline* is set at '2' in order to recover along its timeline.

Restart the PostgreSQL server and enter PITR mode to recover the database at the target time along the timelineId 2. See Fig. 10.5.

Fig. 10.5. Recover the database at 12:15:00 along the timelineId 2.



(1) PostgreSQL reads the value of '*CHECKPOINT LOCATION*' from the backup_label file.
(2) Some values of parameters are read from the recovery.conf; in this example, *restore_command*, *recovery_target_time*, and *recovery_target_timeline*.
(3) PostgreSQL reads the timeline history file '00000002.history' which is corresponding to the value of the parameter *recovery_target_timeline*.
(4) PostgreSQL does replaying WAL data by the following steps:
   1. From the REDO point to the LSN '0/A000198', which is written in the 00000002.history file, PostgreSQL reads and replays WAL data of appropriate archive logs whose timelineId is

1.
2. From the one after LSN '0/A000198' to the one before the timestamp '2023-10-26 12:15:00', PostgreSQL reads and replays WAL data (of appropriate archive logs) whose timelineId is 2.

(5) When the recovery process completes, the current timelineId will advance to 3, and a new timeline history file named *00000003.history* is created in the pg_wal subdirectory (pg_xlog if versions 9.6 or earlier) and the archival directory.

```
postgres> cat /home/postgres/archivelogs/00000003.history
1         0/A000198     before 2023-10-26 12:05:00.861324+00

2         0/B000078     before 2023-10-26 12:15:00.927133+00
```

When you do PITR more than once, you should explicitly set a timelineId for using the appropriate timeline history file.

In this way, timeline history files are not only history logs of database cluster, but also the recovery instruction documents for PITR process.