

The Internals of PostgreSQL

for database administrators and system
developers

Chapter 10

Base Backup & Point-in-Time Recovery

Online database backup can be roughly classified into two categories: logical and physical backups. While both of them have advantages and disadvantages, there is one disadvantage in logical backup; taking too much time for its performance. In particular, it requires a fairly long time to make the backup of a large database, and even more time to restore such a database from backup data. On the contrary, physical backup makes it possible to backup and to restore large databases in a relatively short time so that it is a very important and useful feature in practical systems.

In PostgreSQL, online physical full backup has been available since version 8.0, and 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 continuous archiving feature. For example, even if you made a critical mistake (e.g. truncating all tables), this feature enables you to restore the database of the point just before the mistake you have made.

In this chapter, following topics are described:

- What base backup is
- How PITR works
- What timelineId is
- What timeline history file is



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

10.1. Base Backup

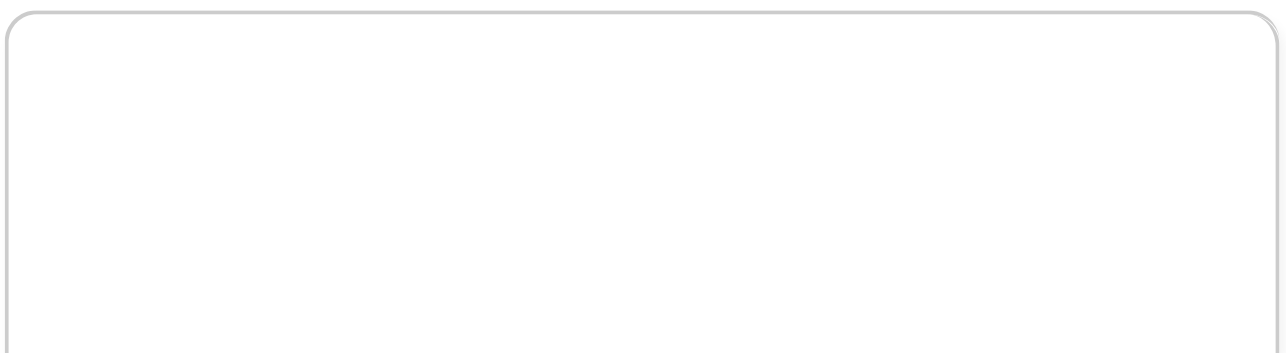
First of all, the standard procedure to make a base backup using the low-level commands is shown below:

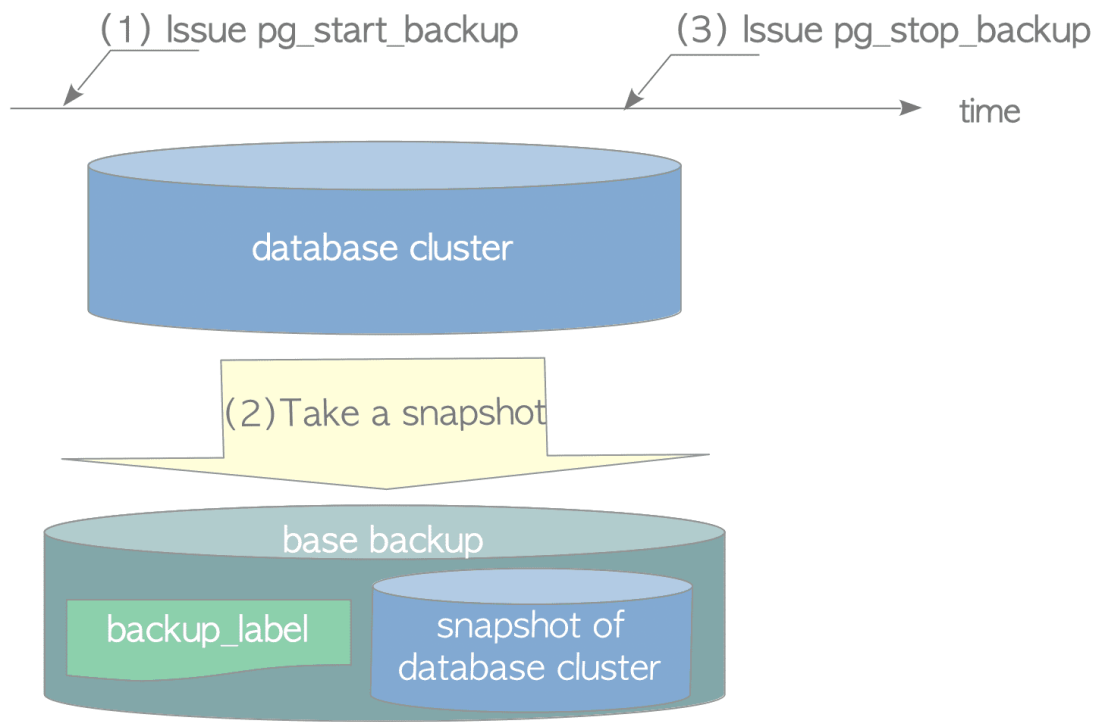
- (1) Issue the *pg_start_backup* command
- (2) Take a snapshot of the database cluster with the archiving command you want to use
- (3) Issue the *pg_stop_backup* command

This simple procedure is easy-to-use for database system administrators, because it requires no special tools but common tools such as copy command or a similar archiving tools to create a base backup. In addition, in this procedure, no table locks are required and all users can issue queries without being affected by backup operation. Those are big advantages over other major open source RDBMS.

More simple way to make a base backup is to use the *pg_basebackup* utility, but it internally issues those low-level commands.

Fig. 10.1. Making a base backup.





As the pair of those commands is clearly one of the key points for understanding the PITR, we will explore them in the following subsections.



The `pg_start_backup` and `pg_stop_backup` commands are defined here:
`src/backend/access/transam/xlogfuncs.c`.

10.1.1. `pg_start_backup`

The *pg_start_backup* prepares for making a base backup. As discussed in Section 9.8, recovery process starts from a REDO point, so the *pg_start_backup* must do 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 checkpoint might be done a number of times during backup. Hence the *pg_start_backup* performs the following four operations:

1. Force into the full-page write mode.
2. Switch to the current WAL segment file (version 8.4 or later).
3. Do 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 (version 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 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. (Either *'pg_start_backup'* or *'pg_basebackup'*.)
- BACKUP FROM – This shows whether this backup is taken from primary or standby.
- START TIME – This is the timestamp when the *pg_start_backup* has executed.
- LABEL – This is the label specified at the *pg_start_backup*.
- 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 backup_label file in version 9.6 is shown in the following:

```
postgres> cat /usr/local/pgsql/data/backup_label
START WAL LOCATION: 0/9000028 (file 0000000100000
000000000009)
CHECKPOINT LOCATION: 0/9000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2021-7-6 11:45:19 GMT
LABEL: Weekly Backup
```

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

10.1.2. pg_stop_backup

The *pg_stop_backup* 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_start_backup*.
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_stop_backup* has been executed.

5. Delete the backup_label file – The backup_label file is required for recovery from the base backup and once copied, it is not necessary in the original database cluster.



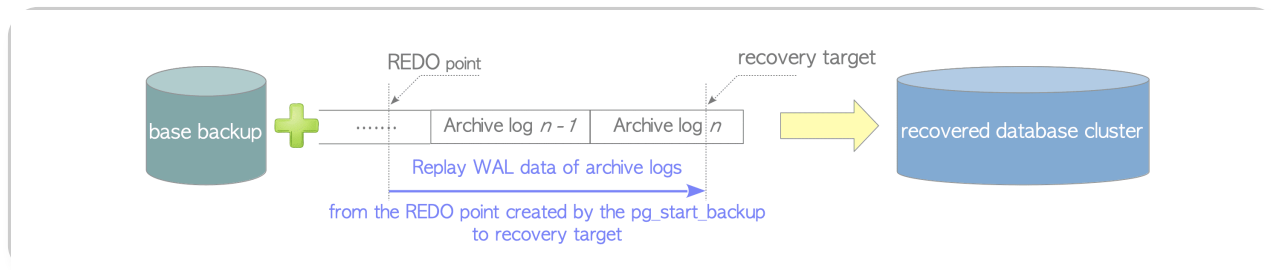
The *naming method for backup history file* is shown below.

`{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. PostgreSQL in PITR-mode replays the WAL data of the archive logs on the base backup, from the REDO point created by the *pg_start_backup* 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.



Here is the description of how PITR works. Suppose that you made a mistake at 12:05 GMT of 6 July, 2021. You should remove the database cluster and restore the new one using the base backup you made before that. Then, set the command of the parameter *restore_command*, and also set the time of the parameter *recovery_target_time* at the point you made the mistake (in this case, 12:05 GMT) in a *recovery.conf* (version 11 or earlier) or *postgresql.conf* (version 12 or later).

```
# Place archive logs under /mnt/server/archivedir directory.
```

```
restore_command = 'cp /mnt/server/archivedir/%f  
%p'
```

```
recovery_target_time = "2021-7-6 12:05 GMT"
```

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

recovery.conf

recovery.conf has been abolished in version 12, and all recovery-related parameters should be written in *postgresql.conf*. See the [official document](#) in detail.

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

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

If a recovery target is not set to the *recovery.conf* (version 11 or earlier) or the *postgresql.conf* (version 12 or later), PostgreSQL will replay until the end of archiving logs.

- (4) When the recovery process completes, a **timeline history file**, such as '00000002.history', is created in the *pg_xlog* subdirectory (in version 10 or later, *pg_wal* subdirectory); if archiving log feature is enabled, 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.

PITR process is almost the same as the normal recovery process described in Chapter 9; the only differences between them are the following two points:

1. Where are WAL segments/Archive logs read from?
 - Normal recovery mode – from the `pg_xlog` subdirectory (in version 10 or later, `pg_wal` 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 a *pg_control file*.

- PITR mode – from a *backup_label* file.

The outline of PITR process is described as follows:

- (1) In order to find the REDO point, PostgreSQL reads the value of '*CHECKPOINT LOCATION*' from the *backup_label* file with the internal function *read_backup_label*.
- (2) PostgreSQL reads some values of parameters from the *recovery.conf* (version 11 or earlier) or the *postgresql.conf* (version 12 or later) ; in this example, *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 which are copied from the archival area to a temporary area by executing the command written in the parameter *restore_command*. (The copied log files in the temporary area are removed after using.)

In this example, PostgreSQL reads and replays WAL data from the REDO point to the one before the timestamp '2021-7-6 12:05:00' because the parameter *recovery_target_time* is set to this timestamp. If a recovery target is not set to the *recovery.conf* (version 11 or earlier) or the

postgresql.conf (version 12 or later), PostgreSQL will replay until the end of archiving logs.

- (4) When the recovery process completes, a **timeline history file**, such as '00000002.history', is created in the `pg_xlog` subdirectory (in version 10 or later, `pg_wal` subdirectory); if archiving log feature is enabled, 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?

Recovery process is a process to make a database cluster in a consistent state, though the cluster is inconsistent. As PITR is based on the recovery process, it can recover the database cluster even if a base backup is a bunch of inconsistent files. This is the reason why we can use common archiving tools without a file system snapshot capability or a special tool.

10.3. `timelineld` and Timeline History File

Timeline in PostgreSQL is used to distinguish between the original database cluster and the recovered ones, and is central concept of PITR. In

this section, two things associated with the timeline are described: *timelineId* and *timeline history files*.

10.3.1. timelineId

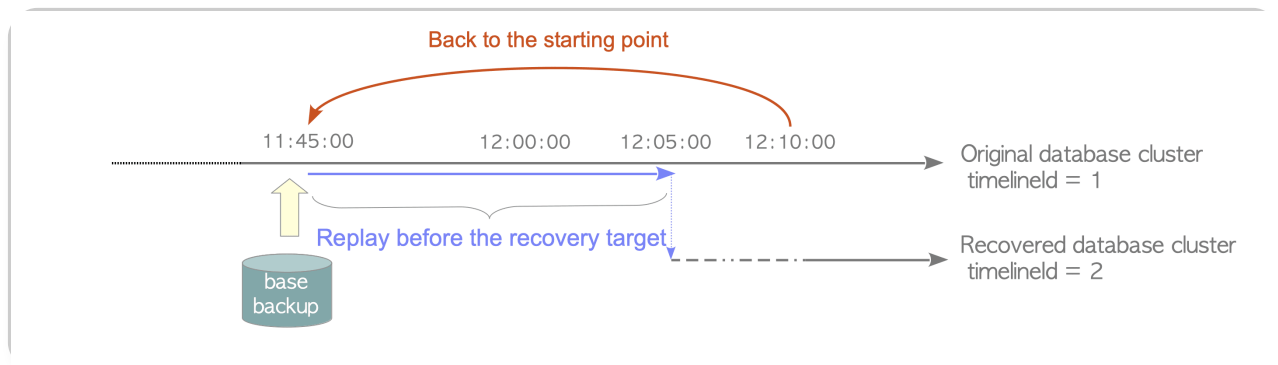
Each timeline is given a corresponding **timelineId**, a 4-byte unsigned integer starting at 1.

An individual timelineId is assigned to each database cluster. The timelineId of original database cluster created by the `initdb` utility is 1. Whenever database cluster recovers, timelineId will be 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, and such situation is represented in 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_start_backup* until the recovery target by tracing along the initial timeline (timelineId 1), and such situation is represented in

the blue arrow line in the figure. Then, a new timelineId 2 is assigned to recovered database cluster and PostgreSQL runs on the new timeline.

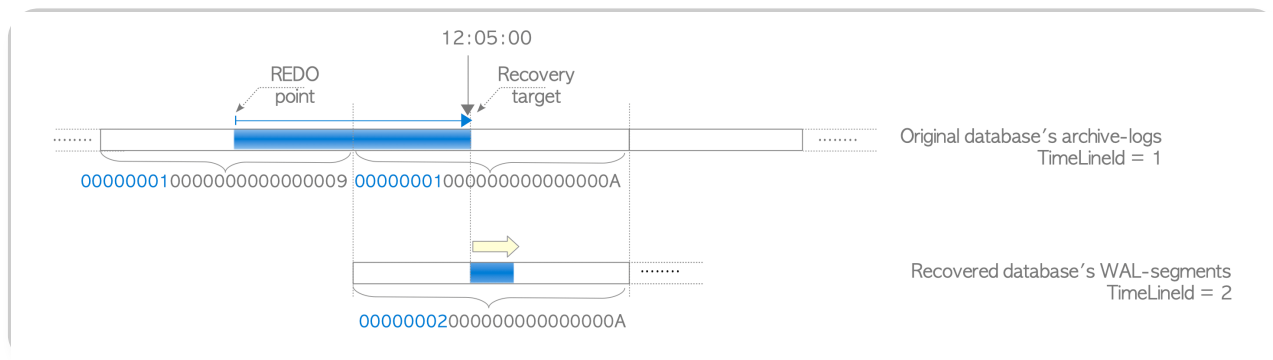
Fig. 10.3. Relation of timelineId between an original and a recovered database clusters.



As briefly mentioned in Chapter 9, the first 8-digit of WAL segment filename is equal to the timelineId of the database cluster created for each segment. When the timelineId is changed, WAL segment filename will also be changed.

Focusing on WAL segment files, the recovery process will be described again. Suppose that we recover the database cluster using two archive logs '00000001000000000000000009' and '0000000100000000000000000A'. The newly recovered database cluster is assigned the timelineId 2, and PostgreSQL creates the WAL segment from '0000000200000000000000000A'. 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 version 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 in the following:

"8-digit new timelineId".history

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

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

A specific example is shown below:

```
postgres> cat /home/postgres/archivelogs/0000  
0002.history  
1          0/A000198      before 2021-7-6 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* ‘2021-7-6 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 to PITR process itself. The detail is explained in the next section.



The timeline history file format is changed in version 9.3. Formats of versions 9.3 or later and earlier both 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 the 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

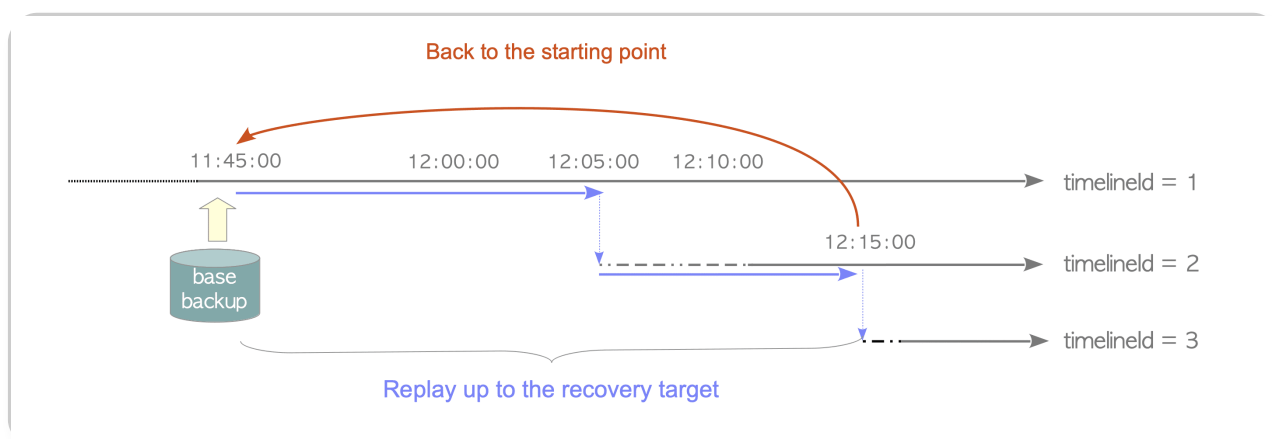
database cluster, you should create a new `recovery.conf` shown below:

```
restore_command = 'cp /mnt/server/archivedir/%f  
%p'  
recovery_target_time = "2021-7-6 12:15:00 GMT"  
recovery_target_timeline = 2
```

The parameter `recovery_target_time` sets the time you made new mistake, and the `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 `'2021-7-6 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 new timeline history file named *00000003.history* is created in the `pg_xlog` subdirectory (`pg_wal` subdirectory if version 10 or later) and the archival directory.

```
postgres> cat /home/postgres/archivelogs/0000003.history
1      0/A000198    before 2021-7-6 12:05:00.861324+00

2      0/B000078    before 2021-7-6 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.

© Copyright 2015-2021 Hironobu SUZUKI All Rights Reserved.