The Internals of PostgreSQL

for database administrators and system developers

Chapter 11

Streaming Replication

The synchronous streaming replication was implemented in version 9.1. It is a so-called single-master-multi-slaves type replication, and those two terms – master and slave(s) – are usually referred to as **primary** and **standby(s)** respectively in PostgreSQL.

This native replication feature is based on the log shipping, one of the general replication techniques, in which a primary server continues to send **WAL data** and then, each standby server replays the received data immediately.

This chapter covers the following topics focusing on how streaming replication works:

- How Streaming Replication starts up
- How the data are transferred between primary and standby servers
- How primary server manages multiple standby servers
- How primary server detects failures of standby servers

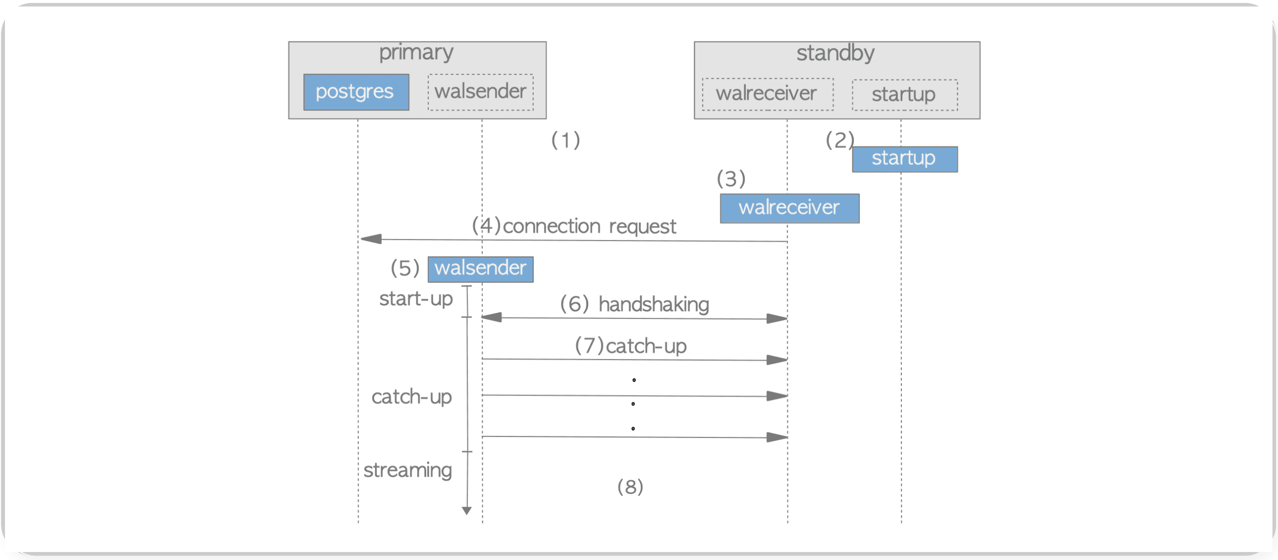
Though the first replication feature, only for asynchronous replication, had implemented in version 9.0, it had replaced with the new implementation (currently in use) for synchronous replication in 9.1.

11.1. Starting the Streaming Replication

In Streaming Replication, three kinds of processes work cooperatively. A **walsender** process on the primary server sends WAL data to standby server; and then, a **walreceiver** and a **startup** processes on standby server receives and replays these data. A walsender and a walreceiver communicate using a single TCP connection.

In this section, we will explore the start-up sequence of the streaming replication to understand how those processes are started and how the connection between them is established. Figure 11.1 shows the startup sequence diagram of streaming replication:

Fig. 11.1. SR startup sequence.



(1) Start primary and standby servers.
(2) The standby server starts a startup process.
(3) The standby server starts a walreceiver process.
(4) The walreceiver sends a connection request to the primary server. If the primary server is not running, the walreceiver sends these requests periodically.
(5) When the primary server receives a connection request, it starts a walsender process and a TCP connection is established between the walsender and walreceiver.
(6) The walreceiver sends the latest LSN of standby's database cluster. In general, this phase is known as **handshaking** in the field of information technology.
(7) If the standby's latest LSN is less than the primary's latest LSN (Standby's LSN < Primary's LSN), the walsender sends WAL data from the former LSN to the latter LSN. Such WAL data are provided by WAL segments stored in the primary's pg_xlog subdirectory (in version 10 or later, pg_wal subdirectory). Then, the standby server replays the received WAL data. In this phase, the standby catches up with the primary, so it is called **catch-up**.
(8) Streaming Replication begins to work.

Each walsender process keeps a state appropriate for the working phase of connected walreceiver or any application (Note that it is not the state of walreceiver or application connected to the walsender.) The following are the possible states of it:

- start-up – From starting the walsender to the end of handshaking. See Figs. 11.1(5)–(6).
- catch-up – During the catch-up phase. See Fig. 11.1(7).
- streaming – While Streaming Replication is working. See Fig. 11.1(8).
- backup – During sending the files of the whole database cluster for backup tools such as *pg_basebackup* utility.

The *pg_stat_replication* view shows the state of all running walsenders. An example is shown below:

```
testdb=# SELECT application_name,state FROM p
g_stat_replication;
 application_name |   state
------------------+-----------
 standby1         | streaming
 standby2         | streaming
 pg_basebackup    | backup
(3 rows)
```

As shown in the above result, two walsenders are running to send WAL data for the connected standby servers, and another one is running to send all files of the database cluster for *pg_basebackup* utility.

❓ What will happen if a standby server restarts after a long time in the stopped condition?

In version 9.3 or earlier, if the primary's WAL segments required by the standby server have already been recycled, the standby cannot catch up with the primary server. There is no reliable solution for this problem, but only to set a large value to the configuration parameter *wal_keep_segments* to reduce the possibility of the occurrence. It's a stopgap solution.

In version 9.4 or later, this problem can be prevented by using *replication slot*. The replication slot is a feature that expands the flexibility of the WAL data sending, mainly for the *logical replication*, which also provides the solution to this problem – the WAL segment files which contain unsent data under the *pg_xlog* (or *pg_wal* if version 10 or later) can be kept in the replication slot by pausing recycling process. Refer the official document for detail.

11.2. How to Conduct Streaming Replication

Streaming replication has two aspects: log shipping and database synchronization. Log shipping is obviously one of those aspects since the streaming replication is based on it – the primary server sends WAL data to the connected standby servers whenever the writing of them occurs. Database synchronization is required for synchronous replication – the primary server communicates with each multiple-standby server to synchronize its database clusters.

To accurately understand how streaming replication works, we should explore how one primary server manages multiple standby servers. To make it rather simple, the special case (i.e. single-primary single-standby system) is described in this section, while the general case (single-primary multi-standbys system) will be described in the next section.

11.2.1. Communication Between a Primary and a Synchronous Standby

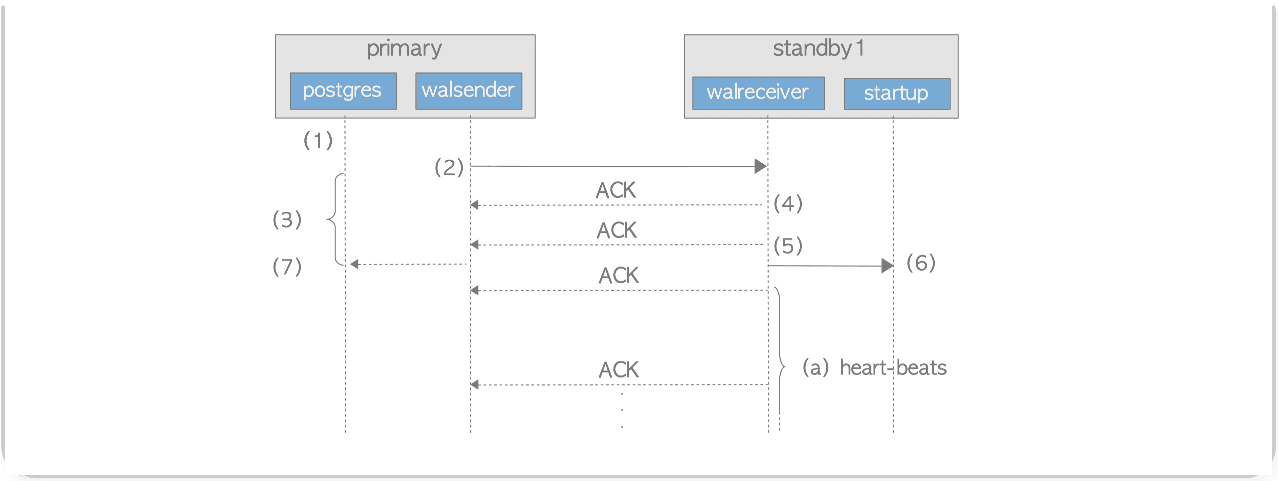
Assume that the standby server is in the synchronous replication mode, but the configuration parameter *hot-standby* is disabled and *wal_level* is '*archive*'. The main parameter of the primary server is shown below:

```
synchronous_standby_names = 'standby1'
hot_standby = off
wal_level = archive
```

Additionally, among the three triggers to write the WAL data mentioned in Section 9.5, we focus on the transaction commits here.

Suppose that one backend process on the primary server issues a simple INSERT statement in autocommit mode. The backend starts a transaction, issues an INSERT statement, and then commits the transaction immediately. Let's explore further how this commit action will be completed. See the following sequence diagram in Fig. 11.2:

Fig. 11.2. Streaming Replication's communication sequence diagram.



(1) The backend process writes and flushes WAL data to a WAL segment file by executing the functions *XLogInsert()* and *XLogFlush()*.
(2) The walsender process sends the WAL data written into the WAL segment to the walreceiver process.
(3) After sending the WAL data, the backend process continues to wait for an ACK response from the standby server. More precisely, the backend process gets a latch by executing the internal function *SyncRepWaitForLSN()*, and waits for it to be released.
(4) The walreceiver on standby server writes the received WAL data into the standby's WAL segment using the *write()* system call, and returns an ACK response to the walsender.
(5) The walreceiver flushes the WAL data to the WAL segment using the system call such as *fsync()*, returns another ACK response to the

walsender, and informs the startup process about WAL data updated.

(6) The startup process replays the WAL data, which has been written to the WAL segment.

(7) The walsender releases the latch of the backend process on receiving the ACK response from the walreceiver, and then, backend process's commit or abort action will be completed. The timing for latch-release depends on the parameter *synchronous_commit*. It is *'on'* (default), the latch is released when the ACK of step (5) received, whereas it is *'remote_write'*, when the ACK of step (4) received.

If the configuration parameter *wal_level* is *'hot_standby'* or *'logical'*, PostgreSQL writes a WAL record regarding the hot standby feature, following the records of a commit or abort action. (In this example, PostgreSQL does not write that record because it’s *'archive'*.)

Each ACK response informs the primary server of the internal information of standby server. It contains four items below:

- LSN location where the latest WAL data has been written.
- LSN location where the latest WAL data has been flushed.
- LSN location where the latest WAL data has been replayed in the startup process.
- The timestamp when this response has be sent.

Walreceiver returns ACK responses not only when WAL data have been written and flushed, but also periodically as the heartbeat of standby server. The primary server therefore always grasps the status of all connected standby servers.

By issuing the queries as shown below, the LSN related information of the connected standby servers can be displayed.

```
testdb=# SELECT application_name AS host,
        write_location AS write_LSN, flush_lo
cation AS flush_LSN,
        replay_location AS replay_LSN FROM pg
_stat_replication;

   host    | write_lsn | flush_lsn | replay_ls
n
```

```
----------+-----------+-----------+----------
--
  standby1 | 0/5000280 | 0/5000280 | 0/5000280
  standby2 | 0/5000280 | 0/5000280 | 0/5000280
(2 rows)
```

The heartbeat's interval is set to the parameter *wal_receiver_status_interval*, which is 10 seconds by default.

11.2.2. Behavior When a Failure Occurs

In this subsection, descriptions are made on how the primary server behaves when the synchronous standby server has failed, and how to deal with the situation.

Even if the synchronous standby server has failed and is no longer able to return an ACK response, the primary server continues to wait for responses forever. So, the running transactions cannot commit and subsequent query processing cannot be started. In other words, all of the primary server operations are practically stopped. (Streaming replication does not support a function to revert automatically to asynchronous-mode by timing out.)

There are two ways to avoid such situation. One of them is to use multiple standby servers to increase the system availability, and the other is to switch from synchronous to *asynchronous* mode by performing the following steps manually.

(1) Set empty string to the parameter *synchronous_standby_names*.

```
synchronous_standby_names = ''
```

(2) Execute the pg_ctl command with *reload* option.

```
postgres> pg_ctl -D $PGDATA reload
```

The above procedure does not affect the connected clients. The primary server continues the transaction processing as well as all sessions between clients and the respective backend processes are kept.

11.3. Managing Multiple-Standby Servers

In this section, the way streaming replication works with multiple standby servers is described.

11.3.1. sync_priority and sync_state

Primary server gives *sync_priority* and *sync_state* to all managed standby servers, and treats each standby server depending on its respective values. (The primary server gives those values even if it manages just one standby server; haven't mentioned this in the previous section.)

sync_priority indicates the priority of standby server in synchronous-mode and is a fixed value. The smaller value shows the higher priority, while 0 is the special value that means 'in asynchronous-mode'. Priorities of standby servers are given in the order listed in the primary's configuration parameter *synchronous_standby_names*. For example, in the following configuration, priorities of standby1 and standby2 are 1 and 2, respectively.

```
synchronous_standby_names = 'standby1, standby2'
```

(Standby servers not listed on this parameter are in asynchronous-mode, and their priority is 0.)

sync_state is the state of the standby server. It is variable according to the running status of all standby servers and the individual priority. The followings are the possible states:

- **Sync** is the state of synchronous-standby server of the highest priority among all working standbys (except asynchronous-servers).
- **Potential** is the state of spare synchronous-standby server of the second or lower priority among the all working standbys (except asynchronous-servers). If the synchronous-standby has failed, it will be replaced with the highest priority standby within the potential ones.
- **Async** is the state of asynchronous-standby server, and this state is fixed. The primary server treats asynchronous-standbys in the same way as potential standbys except that their *sync_state* never be *'sync'* or *'potential'*.

The priority and the state of the standby servers can be shown by issuing the following query:

```
testdb=# SELECT application_name AS host,
         sync_priority, sync_state FROM pg_st
at_replication;
   host     | sync_priority | sync_state
------------+---------------+------------
 standby1   |             1 | sync
 standby2   |             2 | potential
(2 rows)
```

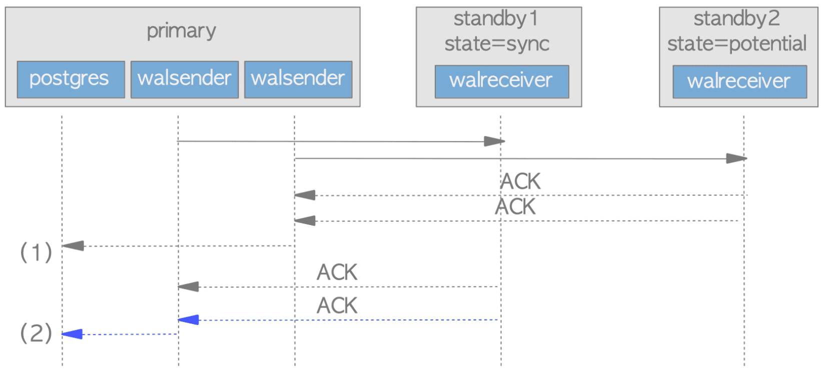
Several developers are recently trying to implement 'multiple synchronous-standby'. See here for details.

11.3.2. How the Primary Manages Multiple-standbys

The primary server waits for ACK responses from the synchronous standby server alone. In other words, the primary server confirms only synchronous standby's writing and flushing of WAL data. Streaming replication, therefore, ensures that only synchronous standby is in the consistent and synchronous state with the primary.

Figure 11.3 shows the case in which the ACK response of potential standby has been returned earlier than that of the primary standby. There, the primary server does not complete the commit action of the current transaction, and continues to wait for the primary's ACK response. And then, when the primary's response is received, the backend process releases the latch and completes the current transaction processing.

Fig. 11.3. Managing multiple standby servers.



The sync_state of standby1 and standby2 are *'sync'* and *'potential'* respectively. (1) In spite of receiving an ACK response from the potential standby server, the primary's backend process continues to wait for an ACK response from the synchronous-standby server. (2) The primary's backend process releases the latch, completes the current transaction processing.

In the opposite case (i.e. the primary's ACK response has been returned earlier than the potential's one), the primary server immediately completes the commit action of the current transaction without ensuring if the potential standby writes and flushes WAL data or not.

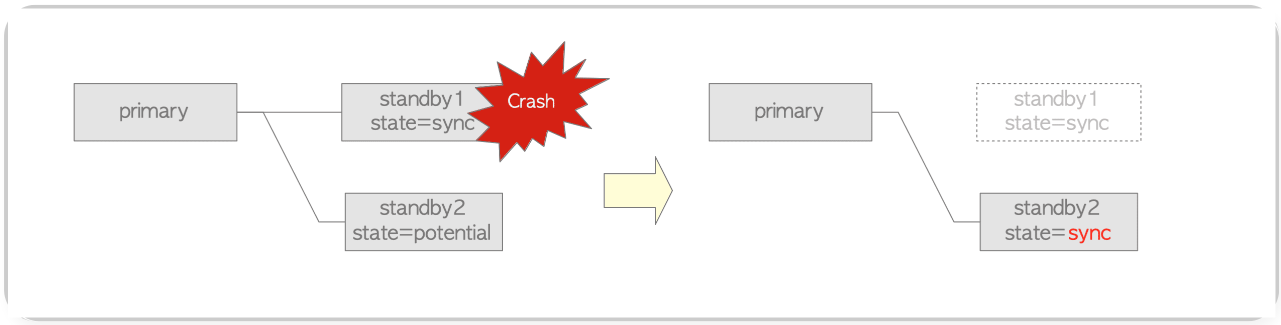
11.3.3. Behavior When a Failure Occurs

Once again, see how the primary server behaves when the standby server has failed.

When either a potential or an asynchronous standby server has failed, the primary server terminates the walsender process connected to the failed standby and continues all processing. In other words, transaction processing of the primary server would not be affected by the failure of either type of standby server.

When a synchronous standby server has failed, the primary server terminates the walsender process connected to the failed standby, and replaces synchronous standby with the highest priority potential standby. See Fig. 11.4. In contrast to the failure described above, query processing on the primary server will be paused from the point of failure to the replacement of synchronous standby. (Therefore, failure detection of standby server is a very important function to increase availability of replication system. Failure detection will be described in the next section.)

Fig. 11.4. Replacing of synchronous standby server.



In any case, if one or more standby server shall be running in syncrhonous-mode, the primary server keeps only one synchronous standby server at all times, and the synchronous standby server is always in a consistent and synchronous state with the primary.

11.4. Detecting Failures of Standby Servers

Streaming replication uses two common failure detection procedures that will not require any special hardware at all.

1. Failure detection of standby server process
   When a connection drop between walsender and walreceiver has been detected, the

primary server *immediately* determines that the standby server or walreceiver process is faulty. When a low level network function returns an error by failing to write or to read the socket interface of walreceiver, the primary also *immediately* determines its failure.

2. Failure detection of hardware and networks
   If a walreceiver returns nothing within the time set for the parameter *wal_sender_timeout* (default 60 seconds), the primary server determines that the standby server is faulty. In contrast to the failure described above, it takes a certain amount of time – maximum is *wal_sender_timeout* seconds – to confirm the standby's death on the primary server even if a standby server is no longer able to send any response by some failures (e.g. standby server's hardware failure, network failure, and so on).

Depending on the types of failures, it can usually be detected immediately after a failure occurs, while sometimes there might be a time lag between the occurrence of failure and the detection of it. In particular, if a latter type of failure occurs in synchronous standby server, all transaction processing on the primary server will be stopped until detecting the failure of standby, even though multiple potential standby servers may have been working.

The parameter *wal_sender_timeout* was called as *replication_timeout* in version 9.2 or earlier.