Chapter 11

Streaming Replication

Synchronous streaming replication was implemented in version 9.1. It is a single-master-multi-slaves type of replication, where the terms "master" and "slaves" are usually referred to as **primary** and **standbys** respectively.

This native replication feature is based on log shipping, a general replication technique in which the primary server continuously sends **WAL (Write-Ahead Log) data** to the standby servers, which then replay the received data immediately.

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

- How streaming replication starts up
- How data is transferred between the primary and standby servers
- How the primary server manages multiple standby servers
- How the primary server detects failures of standby servers

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

11.1. Starting the Streaming Replication

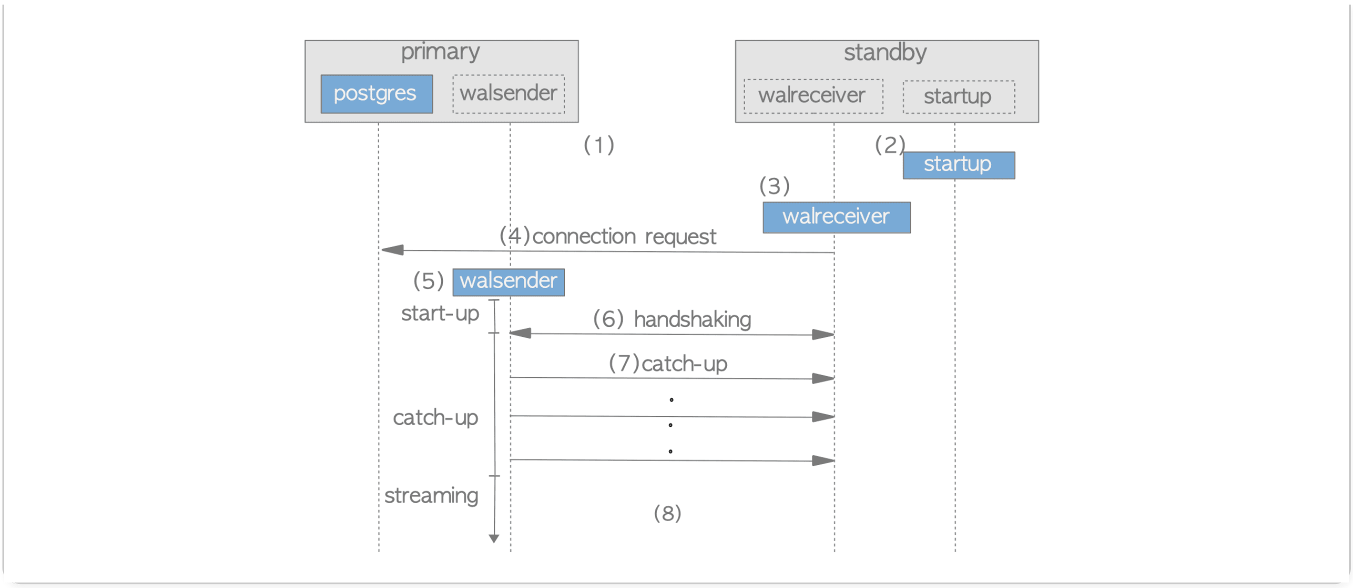
In streaming replication, three types of processes work cooperatively:

- A **walsender** process on the primary server sends WAL (Write-Ahead Log) data to the standby server.
- A **walreceiver** process on the standby server receives and replays the WAL data.
- A **startup** process on the standby server starts the walreceiver process.

The walsender and walreceiver communicate using a single TCP connection.

The startup sequence of streaming replication is shown in Figure 11.1:

Fig. 11.1. SR startup sequence.



(1) Start the primary and standby servers.
(2) The standby server starts the 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 (Log Sequence Number) of standby's database cluster. This 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. These WAL data are provided by WAL segments stored in the primary's pg_wal subdirectory (in versions 9.6 or earlier, pg_xlog). The standby server then 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 that is appropriate for the working phase of the connected walreceiver or application. The following are the possible states of a walsender process:

- 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 pg_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 versions 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. This is a stopgap solution.

In versions 9.4 or later, this problem can be prevented by using *replication slot*. A 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 that contain unsent data under the *pg_wal* (or *pg_xlog* if versions 9.6 or earlier) 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 the main aspect of streaming replication, as the primary server sends WAL (Write-Ahead Log) data to the connected standby servers whenever they are written. Database synchronization is required for synchronous replication, where the primary server communicates with each standby server to synchronize their database clusters.

To accurately understand how streaming replication works, we need to understand how one primary server manages multiple standby servers. We will start with the simple case (i.e., single-primary single-standby system) in this section, and then discuss the general case (single-primary multi-standby system) in the next section.

11.2.1. Communication Between a Primary and a Synchronous Standby

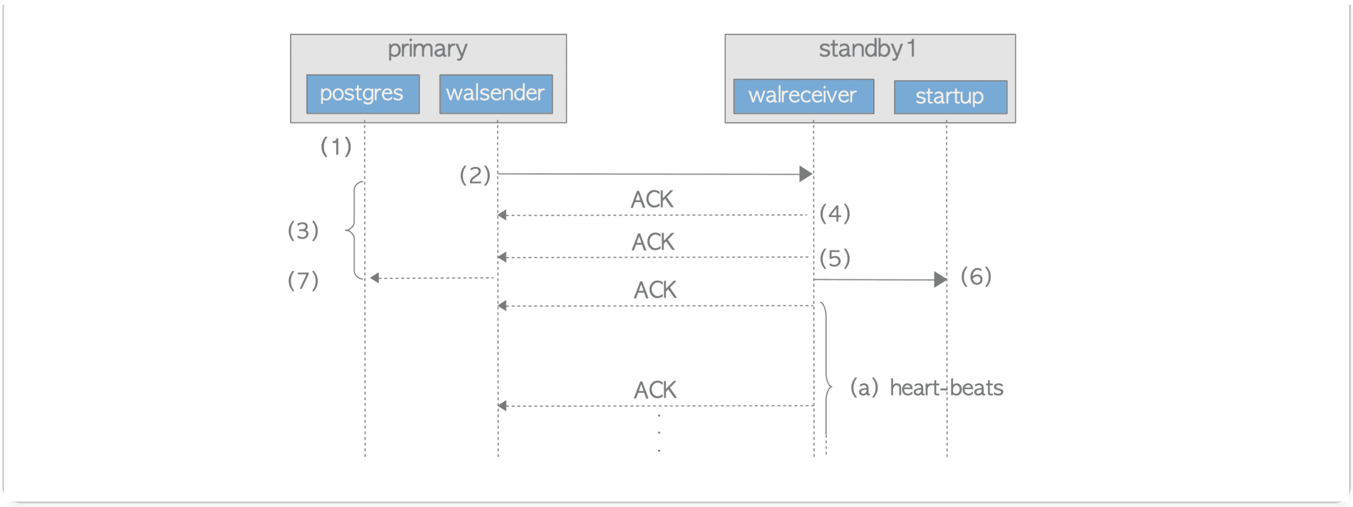
Assume that the standby server is in the synchronous replication mode, but the configuration parameter *hot_standby* is disabled and *wal_level* is '*replica*'. The main parameter of the primary server is shown below:

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

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 the 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, the 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'*, the latch is released when the ACK of step (4) is received.

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

```
XLogWalRcvSendReply(void)@src/backend/replication/walreceiver.c
        /* Construct a new message */
        writePtr = LogstreamResult.Write;
        flushPtr = LogstreamResult.Flush;
        applyPtr = GetXLogReplayRecPtr(NULL);

        resetStringInfo(&reply_message);
        pq_sendbyte(&reply_message, 'r');
        pq_sendint64(&reply_message, writePtr);
        pq_sendint64(&reply_message, flushPtr);
        pq_sendint64(&reply_message, applyPtr);
        pq_sendint64(&reply_message, GetCurrentTimestamp());
        pq_sendbyte(&reply_message, requestReply ? 1 : 0);
```

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

The walreceiver returns ACK responses not only when WAL data have been written and flushed, but also periodically as a heartbeat from the standby server. The primary server therefore always has an accurate understanding of the status of all connected standby servers.

The LSN-related information of the connected standby servers can be displayed by issuing the queries shown below:

```
testdb=# SELECT application_name AS host,
        write_location AS write_LSN, flush_location AS flush_LSN,
        replay_location AS replay_LSN FROM pg_stat_replication;

   host    | write_lsn | flush_lsn | replay_lsn
----------+-----------+-----------+------------
 standby1 | 0/5000280 | 0/5000280 | 0/5000280
 standby2 | 0/5000280 | 0/5000280 | 0/5000280
(2 rows)
```

The heartbeat 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, I describe how the primary server behaves when a synchronous standby server fails, and how to deal with the situation.

Even if a synchronous standby server fails and is no longer able to return an ACK response, the primary server will continue to wait for responses forever. This means that running transactions cannot commit and subsequent query processing cannot be started. In other words, all primary server operations are effectively stopped. (Streaming replication does not support a function to automatically revert to asynchronous mode after a timeout.)

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

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

```
synchronous_standby_names = ''
```

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

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

This procedure does not affect connected clients. The primary server will continue to process transactions and all sessions between clients and their respective backend processes will be maintained.

11.3. Managing Multiple-Standby Servers

In this section, I describe the way streaming replication works with multiple standby servers.

11.3.1. sync_priority and sync_state

The primary server assigns the *sync_priority* and *sync_state* attributes to all managed standby servers, and treats each standby server according to its respective values. (The primary server assigns these values even if it manages just one standby server; this was not mentioned in the previous section.)

The *sync_priority* attribute indicates the priority of the standby server in synchronous mode. The lower the value, the higher the priority. The special value 0 means that the standby server is '*in asynchronous mode*'. The priorities of the standby servers are assigned in the order listed in the primary server's configuration parameter *synchronous_standby_names*. For example, in the following configuration, the priorities of standby1 and standby2 are 1 and 2, respectively.

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

(Standby servers that are not listed in this parameter are in asynchronous mode and have a priority of 0.)

sync_state is the state of the standby server. The *sync_state* attribute indicates the state of the standby server. It can be one of the following values:

- **sync:** The standby server is in synchronous mode and is the highest priority standby server that is currently working.
- **potential:** The standby server is in synchronous mode and is a lower priority standby server that is currently working. If the current sync standby server fails, this standby server will be promoted to sync state.
- **async:** The standby server is in asynchronous mode. (It will never be in 'sync' or 'potential' mode.)

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

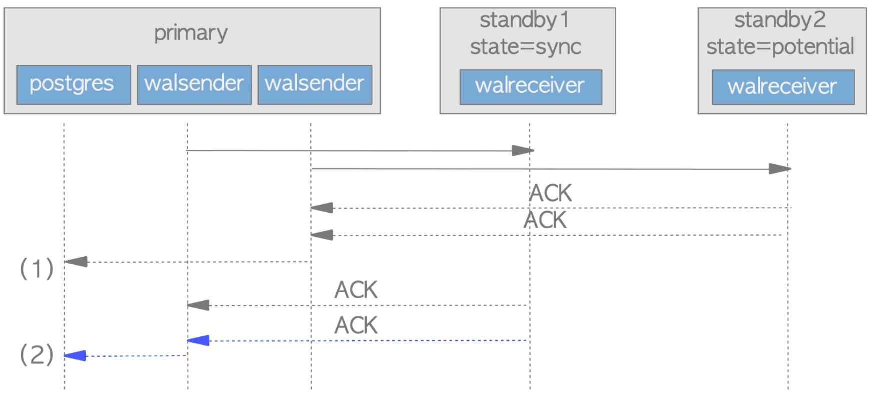
```
testdb=# SELECT application_name AS host,
         sync_priority, sync_state FROM pg_stat_replication;
   host   | sync_priority | sync_state
----------+---------------+------------
 standby1 |             1 | sync
 standby2 |             2 | potential
(2 rows)
```

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 the synchronous standby's writing and flushing of WAL data. Streaming replication, therefore, ensures that only the synchronous standby is in a consistent and synchronous state with the primary.

Figure 11.3 shows the case in which the ACK response of the potential standby has been returned earlier than that of the primary standby. In this case, the primary server does not complete the commit action of the current transaction and continues to wait for the primary's ACK response. 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) The primary's backend process continues to wait for an ACK response from the synchronous standby server, even though it has received an ACK response from the potential standby server. (2) After receiving the ACK response from the synchronous standby server, the primary's backend process releases the latch and completes the current transaction processing.

In the opposite case (i.e., the primary's ACK response has been returned earlier than the potential's), 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.

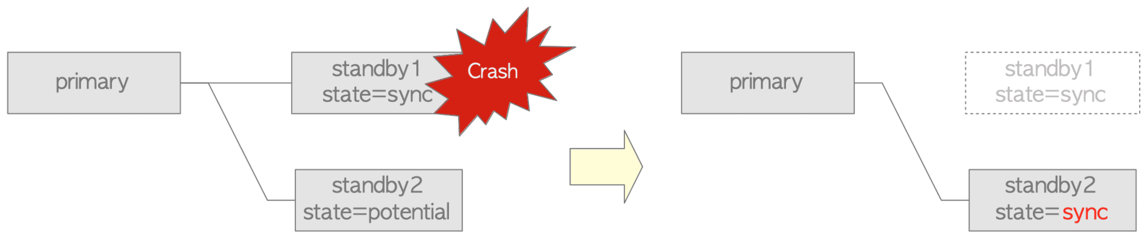
11.3.3. Behavior When a Failure Occurs

Once again, let's see how the primary server behaves when a 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 on 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 the 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 the synchronous standby. (Therefore, failure detection of the standby server is a very important function to increase the availability of the 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 servers are running in synchronous 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 do not require any special hardware.

1. Failure detection of standby server process:
   - When a connection drop between the walsender and walreceiver is 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 read the socket interface of the walreceiver, the primary server also *immediately* determines its failure.
2. Failure detection of hardware and networks:
   - If a walreceiver does not return anything 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, up to *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 due to some failures (e.g., standby server's hardware failure, network failure, etc.).

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