Step by Step Guide on Setting Up Physical Streaming Replication in PostgreSQL

Physical streaming replication in PostgreSQL allows you to maintain a live copy of your database on a standby server, which continuously receives updates from the primary server's WAL (Write-Ahead Log). This standby (or hot standby) can handle read-only queries and be quickly promoted to primary in case of failover, providing high availability and disaster recovery.

In this guide, I will walk through provisioning a primary PostgreSQL 16 server and a standby server on Linux, configuring them for streaming replication, and verifying that everything works. I assume you are an experienced engineer familiar with Linux, but new to PostgreSQL replication, so I will keep it friendly and straightforward.

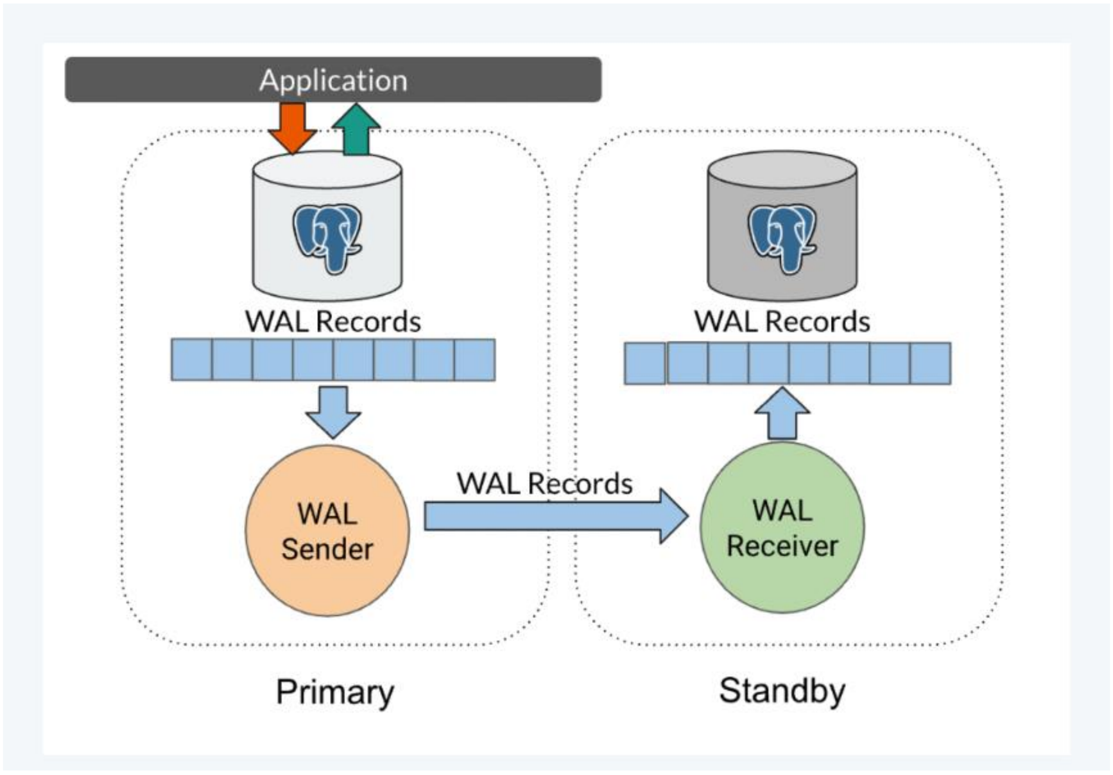

Figure: Real-time data streaming from a primary PostgreSQL server (left) to a standby server (right). The standby constantly applies WAL records received from the primary over a network connection, keeping an up-to-date copy of the database ready for failover.

Table of Contents

Step 1: Prepare Two Linux Servers and Install PostgreSQL 16

Before diving into PostgreSQL settings, set up two Linux servers (virtual or physical). One will act as the **primary** database server, and the other as the **standby** (read replica). For a smooth replication setup, both servers should be as similar as possible in OS, hardware, and PostgreSQL version. In particular, ensure the following prerequisites:

- **PostgreSQL 16 is installed** on both servers via the official PostgreSQL repositories. Both servers must run the same major PostgreSQL version and architecture (mixing different versions won't work for physical replication). If you haven't installed PostgreSQL yet, do so now (e.g., on Ubuntu: sudo apt install postgresql-16, or on RHEL/CentOS: use the PostgreSQL Yum repository). Make sure the PostgreSQL service is running on the primary server.
-
- **Network connectivity:** The standby must be able to reach the primary on the PostgreSQL port (default **5432**). If the servers are in a cloud environment like AWS EC2, configure the security group or firewall to allow the standby's IP to connect to the primary on port 5432. For example, in AWS you'd add an inbound rule permitting the standby's private IP address (or subnet) access to port 5432 on the primary. It is best to use private network interfaces for replication to reduce latency and avoid exposing the database publicly.
-
- **System settings:** Ensure your servers have the necessary OS user and permissions for PostgreSQL. The installation usually creates a postgres UNIX user that owns the data directories. You will run many commands as this postgres user. Also, verify that important prerequisites like consistent time sync (NTP) are in place, as it is generally good practice for database servers (though not specific to replication).
With the infrastructure ready, let's proceed to configure the primary PostgreSQL server to accept replication connections.

Step 2: Configure the Primary Server for Replication

On the primary server, we need to adjust PostgreSQL settings to enable streaming replication and allow the standby to connect. This involves editing the primary's postgresql.conf file to set appropriate parameters. The location of this file varies by installation; on Debian/Ubuntu it might be /etc/postgresql/16/main/postgresql.conf, and on CentOS/Red Hat it could be /var/lib/pgsql/16/data/postgresql.conf. You can confirm the path by connecting to PostgreSQL and running SHOW config_file;.

Open the primary's postgresql.conf in your editor (as root or the postgres user) and enable listening on the network and WAL streaming:

```
# Example: edit postgresql.conf on primaryCopy to Clipboard
```

```
sudo nano /etc/postgresql/16/main/postgresql.conf   # path may differ
```

Find and set the following parameters (remove the # comment prefix if present):

listen_addresses – change this from localhost to an address that allows your standby to connect. For simplicity, you can use '*' to listen on all network interfaces, or specify the primary server's IP address. For example:

```
listen_addresses = '*'Copy to Clipboard
```

This ensures the primary will accept external TCP connections (not just local ones).

- **wal_level** – set to replica. PostgreSQL 16's default is already replica, but it is good to double-check. This level produces enough information in WAL to support standby servers. For physical streaming replication, replica is the appropriate setting (it was known as hot_standby in older versions).
- **max_wal_senders** – increase this if needed to accommodate the number of standbys. This setting is the number of concurrent WAL sender processes the primary can have. Each standby will use one WAL sender connection. The default is typically 10, which is plenty for one or two standbys, but if you plan on more, set accordingly.
- **wal_keep_size** – (optional) set a WAL retention size (in megabytes) to prevent old WAL segments from being recycled too soon. This is useful as a safety net in asynchronous replication. For example:

  ```
  wal_keep_size = 64MBCopy to Clipboard
  ```

  This would keep at least 64 MB of WAL (which is 4 segments of 16MB each) available for the standby to catch up if it falls behind. If you omit this and are not using replication slots (discussed later), the standby could fall too far behind and require a full resynchronization if WAL segments it needs are recycled. Adjust this based on how much WAL your system can generate during potential downtime of the standby.

- **archive_mode** – (optional) you can set archive_mode = on if you plan to archive WAL files to a separate location (for example, to S3 or NFS) as an additional safeguard or for point-in-time recovery. If enabling this, you must also set an archive_command (e.g., a shell command to copy WAL files to your archive storage). Archiving is not strictly required for streaming replication, but it can complement it. (For simplicity, we won't deeply configure WAL archiving in this guide, but be aware it's an option for robust setups.)
- **hot_standby** – this parameter allows the standby to accept read-only queries. It **must** be enabled on the standby server (it has no effect on the primary). By default, PostgreSQL 16 sets hot_standby = on for standby mode, so you typically don't need to change it on the primary's config. We will verify it on the standby later. After making these changes, save postgresql.conf and exit the editor. The primary server needs to apply these new settings. Most of them (like listen_addresses and wal_level) require a restart of PostgreSQL to take effect. Restart the primary PostgreSQL service now to load the new configuration:

  ```
  sudo systemctl restart postgresql@16-main   # Debian/Ubuntu service name example

  # or on Red Hat/CentOS:

  sudo systemctl restart postgresql-16Copy to Clipboard
  ```

  Note: If your primary server is already in production with data, restarting will cause a brief outage. Plan accordingly. Alternatively, some parameters can be set with a pg_reload_conf(), but wal_level **does require a full restart**. It is best to configure replication early in the setup of a new server when possible.

  At this stage, your primary is configured to produce the necessary WAL and listen for a standby connection. Next, we will set up a user for replication and open up authentication.

Step 3: Create a Replication User and Configure Access (pg_hba.conf)

For security and clarity, it is good practice to use a dedicated role for replication. On the primary server, create a replication user that the standby will use to connect.

- **Create the replication role on primary:** Connect to the PostgreSQL prompt as a superuser (usually the postgres user) and create a role with the REPLICATION attribute. For example:

  ```
  sudo -u postgres psql -c "CREATE ROLE replicator WITH REPLICATION LOGIN PASSWORD 'strongpassword';"Copy to Clipboard
  ```

This creates a user named “replicator” with the ability to initiate replication, and with a login password (replace ‘strongpassword’ with a secure password of your choice). You should see CREATE ROLE as the output if successful. Remember this username and password, as the standby will use them to authenticate to the primary.

- **Authorize the standby to connect:** We need to allow our new replicator user to connect from the standby server’s address. PostgreSQL uses the pg_hba.conf file (Host-Based Authentication) to control connection permissions. Open the primary’s pg_hba.conf (it is typically in the same directory as postgresql.conf). Add a line to permit the standby’s IP to connect for replication using the replicator role. For example, add at the end of the file:

```
host    replication    replicator    <standby_ip>/32    md5
```

Let’s break down this rule: host means it is for TCP/IP connections, replication specifies the special replication database (so this isn’t for a specific normal database, but for WAL streaming), replicator is the role allowed, <standby_ip>/32 should be replaced with your standby server’s IP address (in CIDR notation /32 means an exact match to one IP), and md5 is the authentication method (md5 here covers password auth). If you prefer, you could use scram-sha-256 explicitly instead of md5 in that line, since new PostgreSQL installs default to SCRAM encryption for passwords. The result is the same: the standby will present a password and the primary will verify it.

If you expect multiple standbys, you would add multiple lines (or a single line with a wider CIDR range) to cover each standby’s IP. Keep security in mind – only allow the necessary addresses. It is also possible to use a replication user without a password by setting method to trust for a trusted network, or configure certificate-based auth, but those are beyond our scope. I will stick with password auth for now.

- **Reload the configuration:** Once pg_hba.conf is updated, instruct PostgreSQL to reload its configuration to
- apply changes. You can restart the service again, or more minimally send a reload signal. For example:

```
sudo systemctl reload postgresql@16-mainCopy to Clipboard
```

A reload is sufficient for pg_hba changes (no full restart needed). Alternatively, from the SQL prompt, you can run: SELECT pg_reload_conf();.

At this point, the primary server is fully configured to accept a replication connection from the standby. We have wal_level set to replica, a replication user created, and pg_hba.conf allowing that user from the standby’s address. The primary is ready – now we move to setting up the standby server.

Step 4: Initialize the Standby Server with a Base Backup

The standby needs to start with a copy of the primary's data. We'll accomplish this by taking a **base backup** of the primary and restoring it to the standby. PostgreSQL provides the convenient pg_basebackup tool to do this in one step over the network, using the replication connection. This will copy the entire data directory from the primary to the standby.

On the standby server:

- **Install PostgreSQL 16** if you haven't already. Ensure the PostgreSQL service is **stopped** on the standby for now – we don't want the standby's PostgreSQL running until we have the data copied. If the packaging system already initialized a database cluster on the standby (common on Debian/Ubuntu), we need to remove or empty that data directory first. For example, on Debian-based systems the default data directory might be /var/lib/postgresql/16/main/. Make sure the service is stopped, then clear out the directory:

```
sudo systemctl stop postgresql@16-main

sudo -u postgres rm -rf /var/lib/postgresql/16/main/*Copy to Clipboard
```

The above will delete all files in the standby's data directory, so it is empty and ready to receive the base backup. We run it as the postgres user (sudo -u postgres) because that user owns the data files. Ensure the directory exists and has the correct permissions (should be owned by postgres and mode 700). If it doesn't exist or was removed, recreate it:

```
sudo -u postgres mkdir /var/lib/postgresql/16/main && sudo chmod 700 /var/lib/postgresql/16/main.Copy to Clipboard
```

- **Run pg_basebackup to clone the primary:** Use the pg_basebackup utility on the standby to pull data from the primary. This requires the primary is up (which it is, from previous steps) and our replication user credentials. Execute:

```
sudo -u postgres pg_basebackup -h <primary_ip> -p 5432 -U replicator -D /var/lib/postgresql/16/main/ -Fp -Xs -P -R
```

-

Let's break down this command:

- -h <primary_ip> specifies the host of the primary server. Use the primary's private IP or hostname that the standby can reach.
- -p 5432 specifies the port (5432 is default; you can omit this if you didn't change it).
- -U replicator is the replication username we created.
- -D /var/lib/postgresql/16/main/ is the destination directory for the backup (the standby's data directory we emptied).
- -F p (or -Fp) means output in plain format (i.e., as a data directory ready to use, not a tar archive).
- -X s (or -Xs) means include WAL files by streaming them along with the base backup. This ensures the backup is consistent and includes all WAL needed to start replication.
- -P shows progress output, so you can see the backup progress.
- -R tells pg_basebackup to automatically configure this data directory as a replica. This is very handy: it creates a standby.signal file and a postgresql.auto.conf entry for replication settings. In older PostgreSQL versions, you had to create a recovery.conf file manually; in PostgreSQL 16, the presence of standby.signal triggers standby mode. The -R option essentially does this for you, plus it writes the connection info

to postgresql.auto.conf as primary_conninfo. This includes the host, port, user, and password so the standby knows how to connect to the primary.
When you run this command, it will prompt for the replicator user's password (the one you set in Step 3). After authentication, the base backup will begin. It may take some time, depending on the database size. You'll see progress if you used -P. Once finished, you should have a copy of the primary's data in the standby's data directory.

If the command completes successfully, your standby's data directory now contains all the data from the primary, and importantly, a file named standby.signal indicating this instance should start as a standby. There will also be a postgresql.auto.conf (or the main postgresql.conf updated) containing a primary_conninfo setting with the details to connect to the primary. You can inspect it to confirm, but *do not* share the contents of primary_conninfo casually since it likely contains the password in plain text. (PostgreSQL does store the replication password here by default for convenience; if you prefer not to have it in the file, you could instead configure a .pgpass file for the postgres user on the standby. That's an advanced detail— for now, having it in primary_conninfo is fine on a secure server.)

Tip: If you want to use a replication slot for a more robust replication (to ensure the primary doesn't remove WAL that the standby still needs), you can have pg_basebackup create one during this step. For example, add –slot=standby1 –create-slot to the command. This will register a physical replication slot named "standby1" on the primary while taking the backup. The primary will then retain WAL segments until they are applied by this standby, preventing WAL from being recycled too early. This is useful to avoid falling behind; however, be aware that if the standby is down for a long time, the primary's disk could fill with accumulated WAL because the slot prevents it from discarding them. Always monitor disk space if using replication slots. (You can see slot information on the primary via pg_replication_slots view.) If you use –slot with pg_basebackup -R, it will also record the slot name in the standby's configuration (primary_slot_name in postgresql.auto.conf). In our simple setup, we did not use a slot, instead, we might rely on wal_keep_size or WAL archiving as mentioned earlier.

Tune the standby's configuration (if needed): The base backup copied over the primary's configuration files. We should adjust a couple of settings on the standby before starting it:

- Ensure hot_standby is enabled in the standby's postgresql.conf. It should be on by default in PostgreSQL 16, but double-check:

```
hot_standby = on
```

  This allows the standby to accept read-only queries during recovery (i.e., while it's continuously replicating). Without this, the standby would not allow connections until promoted.

- Set the standby's listen_addresses to its own IP or '*' if you want to allow connections (for example, from application servers for read queries, or for an admin to connect and monitor). This wasn't automatically changed by pg_basebackup (which copied the primary's config where listen_addresses was the primary's IP or *). So update it to reflect the standby's network. This will be important both for allowing client reads on the standby and for after a failover (when the standby becomes primary, it should already be configured to listen appropriately).

- You do **not** need to set wal_level or max_wal_senders on the standby; those settings are relevant only for a primary. (If this standby later becomes a primary after failover, it would inherit those settings from the config we copied, so they're effectively already in place for the new primary.)
- The standby will be in continuous recovery, so you generally shouldn't modify other settings yet.
  Save any changes you made to the standby's postgresql.conf. Now we're ready to turn on the standby.

Start the standby PostgreSQL service: Start PostgreSQL on the standby server:

```
sudo systemctl start postgresql@16-mainCopy to Clipboard
```

PostgreSQL will see the standby.signal file and enter recovery/replication mode. The standby will immediately try to connect to the primary using the connection info we set. If all is well, it will start streaming WAL. There's no need to run pg_basebackup again – that was a one-time step to set up the initial data.

Step 5: Verify Streaming Replication is Working

With the standby started, it should be connecting to the primary and replicating any new transactions. Let's verify the replication status.

On the **primary server**, connect to PostgreSQL (sudo -u postgres psql) and run a query to check the replication statistics:

```
SELECT client_addr, state, sync_state
FROM pg_stat_replication;Copy to Clipboard
```

This will show an entry for each standby connected. You should see the IP of your standby (client_addr), and the state should be streaming. For example:

```
client_addr  |   state   | sync_state
-------------+-----------+------------
10.0.0.2     | streaming | asyncCopy to Clipboard
```

In the above, 10.0.0.2 is the standby's address. The state being streaming confirms that the standby is actively receiving WAL data. The sync_state column indicates if this is asynchronous or synchronous; by default, it will be async (as we haven't configured synchronous replication). If you see a row here with state streaming, congratulations – you have a working replication setup!

You can also check on the **standby server** itself. Connect to the standby's PostgreSQL (sudo -u postgres psql) and run:

```
SELECT pg_is_in_recovery();Copy to Clipboard
```

This should return t (true), meaning the standby is in recovery mode (i.e., not running as a normal read-write server). On the standby, you can also query pg_stat_wal_receiver to see details about the WAL receiver process (such as the primary host it's connected to, and the last received LSN). For a quick check, pg_stat_wal_receiver should show a status streaming as well. For example:

```
SELECT status, received_lsn, last_msg_send_time, last_msg_receipt_time Copy to Clipboard

FROM pg_stat_wal_receiver;Copy to Clipboard
```

This will show the WAL receiver status and some timing info. As long as status is streaming and the timestamps are updating, things are good. You can even test by creating a table or inserting data on the primary and ensuring it appears on the standby (remember, you'll only be able to **read** it on the standby, since it's not writable).

Another important view on the primary is pg_replication_slots (if you set up a slot). It will show the slot's name and how much WAL it is retaining. On the primary, pg_stat_replication has more columns (like WAL flush and replay LSN positions) that can help measure lag. In a simple async setup, a small lag is normal; if the standby falls behind significantly, those LSN differences grow.

For basic monitoring, the single query on pg_stat_replication on the primary is often enough: if the standby is listed and state = streaming, your replication link is healthy. In a script, one might periodically check that this view returns expected results or use an external monitoring tool to alert on replication delays.

Step 6: Monitoring Replication Health and Performance

With the replication in place, ongoing monitoring is essential to ensure it stays healthy:

- **pg_stat_replication (on primary):** This is your primary view for monitoring. It shows each connected standby and various metrics. You can monitor the lag by looking at the difference between sent_lsn vs replay_lsn for each standby. Large differences might indicate the standby is behind. There are also timestamp columns (write_lag, etc.) that can help estimate time lag. Consider setting up an alert if the standby's state becomes something other than streaming or if lag exceeds a threshold.
- **Logs:** Keep an eye on PostgreSQL logs on both servers. On the primary, a disconnection of a standby or any errors in sending WAL will be logged. On the standby, connection issues or replication delays will be logged. For example, if the standby falls too far behind, you might see messages about missing WAL segments. Regular log review or monitoring with a log analyzer can catch issues early.
- **pg_stat_wal_receiver (on standby):** This view on the standby shows the status of the WAL receiver. If the standby is having trouble connecting, this view might show a status like connecting or errors.
- **Replication slots:** If using a slot, monitor pg_replication_slots on the primary. Particularly, note the restart_lsn for the slot – if it doesn't advance for a long time while WAL accumulates, your standby might be stuck or far behind. Also watch disk space for the pg_wal directory if a slot is in use.
- **System monitoring:** Ensure the network link between primary and standby is reliable. High latency or packet loss can slow replication. Using tools like netcat or other network monitoring between the servers can be useful.
- **Backups:** Even with replication, you should still take regular backups (logical or physical snapshots) of your data, especially before major changes. Replication is not a backup by itself (a mistaken operation on primary can replicate to standby!). A combination of replication for high availability and backups for recovery from errors is ideal.
  PostgreSQL's native replication is quite robust, but it is not set-and-forget. Plan to monitor your replication and set up alerts for important thresholds.

Tips for Cloud Deployments (AWS and Others)

Deploying PostgreSQL replication in cloud environments introduces a few extra considerations:

- **Firewall and Security Groups:** As mentioned, ensure your cloud firewall rules (security groups in AWS, network security groups in Azure, etc.) allow the necessary traffic. For AWS EC2, for example, the primary's security group should permit the standby's IP on port 5432. It is wise to restrict this to the standby's address or VPC security group rather than open it to the world. Also, if using a cloud provider's managed firewall, double-check both inbound and outbound rules (some clouds might need egress rules as well, though AWS security groups by default allow all egress).
- **Instance roles and permissions:** If you plan to use WAL archiving to cloud storage (like uploading WAL files to Amazon S3 or Google Cloud Storage for safekeeping or for PITR), consider using the cloud's instance role mechanism. For example, on AWS you can assign an IAM Role to your EC2 instance that has permissions to write to a specific S3 bucket. Then your archive_command can use AWS CLI or another tool without embedding AWS keys in plaintext. For example:

```
archive_command = 'aws s3 cp %p s3://your-bucket-name/%f'Copy to Clipboard
```

This makes your setup more secure and manageable. PostgreSQL doesn't directly know about AWS; you would script the archive command (e.g., calling aws s3 cp …). The instance's role (if configured with a proper IAM policy) handles authentication to AWS. This avoids the need to store cloud credentials on the server.

- **Network topology:** If your primary and standby are in different Availability Zones or Regions (for cross-site redundancy), latency will be higher. Streaming replication can work over WAN distances, but be mindful of the performance impact. Asynchronous replication is usually preferred in cross-region scenarios to avoid latency affecting the primary's transactions. If using synchronous replication across regions, expect transaction commit times to increase due to network round-trip delays.
- **Performance and instance type:** Ensure both primary and standby have sufficient resources (CPU, RAM, disk I/O) for your workload. In AWS, database servers benefit from using EBS volumes with provisioned IOPS for consistent performance, especially for WAL writes. If the standby will be used for heavy read queries, make sure it's sized accordingly as well.
- **Maintenance:** In cloud setups, it is easy to spin up a new standby if needed. For example, you can take a snapshot of your primary's volume and launch a new instance from it, then use it as a new standby (though you'd still need to configure it as a standby). However, the easiest way remains using pg_basebackup as we did. Automate the steps where possible (cloud-init scripts, Ansible playbooks, etc., can embed the setup so that you can quickly replace a standby if one fails).
- **Backup strategy:** Consider integrating cloud storage for backups. Even though you have a standby, a separate backup (e.g., daily pg_dump or periodic basebackup to S3) is useful. Cloud object storage is great for keeping such backups. If using continuous archiving to S3, you can achieve point-in-time recovery if needed by replaying WAL from S3. Tools like wal-e, wal-g or custom scripts can help manage this, but again, those are add-ons beyond core PostgreSQL.

Conclusion

By following this guide, you have set up a PostgreSQL 16 primary-standby pair with physical streaming replication on Linux. The primary is streaming WAL records to the standby in real-time, and the standby is applying them to stay in sync. We configured the primary to allow replication, created a replication user, used pg_basebackup to clone the data, and started the standby in hot standby mode. We also verified that the replication is working via pg_stat_replication on the primary, and discussed how to promote the standby in case of failover.

This setup provides a foundation for high availability: the standby can take over if the primary fails, and in the meantime, it can serve read-only queries (offloading some read workload from the primary). The solution uses only built-in PostgreSQL features and tools.

Going forward, you might consider **refining the setup** by tuning performance (e.g., WAL segment size, network compression, etc.), setting up **synchronous replication** if zero data loss is required (this involves configuring synchronous_commit and synchronous_standby_names on the primary so that transactions wait for the standby to confirm WAL apply – see the PostgreSQL docs for details), or adding more standbys to scale out reads