Understanding Synchronous and Asynchronous Replication in PostgreSQL – What is Best for You?

In the world of database replication, choosing between synchronous and asynchronous methods can have a big impact on how reliable, consistent, and fast your data is.

This blog dives into what these methods are, how they work, and when you might want to use one over the other. Whether you're trying to keep your data super safe or just want it to move quickly, we'll break down everything you need to know about synchronous and asynchronous replication in PostgreSQL.

What is Streaming Replication?

Streaming replication enables real-time data movement from a primary server to one or more standby servers. Here's how it works

- The primary server continuously writes changes (updates, inserts, deletes) to a transaction log known as the Write-Ahead Log (WAL).
- Standby servers establish a connection to the primary and continuously read the WAL as it's generated.
- The changes are then applied to the standby server's database, keeping it synchronized with the primary in near real-time.
  This continuous flow of data ensures that standby servers are always up-to-date with the latest information on the primary server. This forms the foundation for both synchronous and asynchronous replication, which differ in how they handle the application of changes on the standby server.

Asynchronous Replication – Advantages and Drawbacks

Asynchronous replication in PostgreSQL prioritizes performance and scalability over strict data consistency at every millisecond. Here's the concept: changes (inserts, updates, deletes) are applied on the primary server first and then streamed asynchronously (meaning at a later time) to the standby server(s). This allows the primary server to handle transactions swiftly without waiting for confirmation from the standby. While the standby server eventually reflects the changes, there might be a slight delay, potentially leading to a small window of inconsistency between the primary and standby data. However, this trade-off translates to faster overall performance and better scalability for high-volume databases.

NOTE:
By default, PostgreSQL's streaming replication operates asynchronously.

Advantage

Flexibility: In case of Standby node fails we will still be able to perform read/write operations on the primary node as it won't wait for data to be replicated to the standby lets see this behaviour with the help of the example

We initialised two Postgres instances within separate Docker containers on the same network. Our plan involves establishing streaming replication between these instances, designating one as primary and the other as standby. Subsequently, we will conduct replication testing. Finally, we'll halt the standby server and attempt data insertion on the primary server to observe its behavior. We expect that the primary server should seamlessly manage various insert, update, and select operations without encountering any problems.

Install PostgreSQL by following community guidelines

https://www.postgresql.org/download/

Initialise the new data directory since we are doing this for testing so we will specify no additional parameters

```
postgres@docker-desktop:~$ /usr/lib/postgresql/16/bin/initdb -D primary
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
The database cluster will be initialized with locale "C".
The default database encoding has accordingly been set to "SQL_ASCII".
The default text search configuration will be set to "english".
Data page checksums are disabled.
creating directory primary ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Asia/Karachi
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok
initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or --auth-local
and --auth-host, the next time you run initdb.Copy to Clipboard
```

Success. You can now start the database server using:
    /usr/lib/postgresql/16/bin/pg_ctl -D primary -l logfile start

On primary server inside postgresql.conf file update following parameters in order to setup streaming asynchronous replication

```
listen_addresses = ‘*’
wal_level = replica
max_wal_senders = 10
wal_keep_size = 1GBCopy to Clipboard
```

Update pg_hba.conf file to allow replication trafic from standy server

```
host    replication     all             <IP_ADDR_OF_STANDBY>/32          trustCopy to Clipboard
```

Start the primary server

```
postgres@docker-desktop:~$ /usr/lib/postgresql/16/bin/pg_ctl -D primary -l logfile start
waiting for server to start.... done
server startedCopy to Clipboard
```

Run pg_basebackup on stanby node for streaming replication it will copy the data directory from the primary and place it on the standby node

```
postgres@docker-desktop:~$ /usr//lib/postgresql//16/bin/pg_basebackup -h 172.18.0.1 -p 5432 -U
postgres -X stream -v -R -w -D standby
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/2000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created temporary replication slot "pg_basebackup_8766"
pg_basebackup: write-ahead log end point: 0/2000100
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: syncing data to disk ...
pg_basebackup: renaming backup_manifest.tmp to backup_manifest
pg_basebackup: base backup completedCopy to Clipboard
```

Because both docker containers are running on the same network and we started the primary on default 5432 port so we have to change the port of standby inside standby/postgresql.conf file we wil use 5433 port for standby server

```
postgres@docker-desktop:~$ cat  standby/postgresql.conf | grep "port = 5433"
port = 5433 # (change requires restart)Copy to Clipboard
```

Start the standby server

```
postgres@docker-desktop:~$ /usr/lib/postgresql/16/bin/pg_ctl -D standby -l logfile start
waiting for server to start.... done
server startedCopy to Clipboard
```

Check standby logs to verify it is replica and read only

```
postgres@docker-desktop:~$ tail logfile
2024-05-16 12:44:33.974 PKT [7057] LOG:  listening on IPv6 address "::", port 5433
2024-05-16 12:44:33.976 PKT [7057] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5433"
2024-05-16 12:44:33.980 PKT [7060] LOG:  database system was interrupted; last known up at 2024-
05-16 12:43:30 PKT
2024-05-16 12:44:34.046 PKT [7060] LOG:  entering standby mode
2024-05-16 12:44:34.046 PKT [7060] LOG:  starting backup recovery with redo LSN 0/2000028,
checkpoint LSN 0/2000060, on timeline ID 1
2024-05-16 12:44:34.049 PKT [7060] LOG:  redo starts at 0/2000028
2024-05-16 12:44:34.050 PKT [7060] LOG:  completed backup recovery with redo LSN 0/2000028 and
end LSN 0/2000100
2024-05-16 12:44:34.050 PKT [7060] LOG:  consistent recovery state reached at 0/2000100
2024-05-16 12:44:34.050 PKT [7057] LOG:  database system is ready to accept read-only connections
2024-05-16 12:44:34.058 PKT [7061] LOG:  started streaming WAL from primary at 0/3000000 on
timeline 1Copy to Clipboard
```

We can see standby is expecting read only connections. Verify replication status from primary lets run this query

```
postgres=# SELECT pid,client_addr,usename,application_name,state,sync_state FROM
pg_stat_replication;
 pid  | client_addr  | usename  | application_name |   state   | sync_state
------+--------------+----------+------------------+-----------+------------
 8768 | 192.168.65.3 | postgres | walreceiver      | streaming | async
(1 row)
Copy to Clipboard
```

We can see our replication is setup properly it is streaming replication with async mode. Let’s create a table on the primary node

```
postgres=# CREATE TABLE numbers (ID SERIAL PRIMARY KEY);
CREATE TABLE
postgres=#
```

```
# Go to standby node connect with psql on port 5433 and see if our table is replicated our not
```

```
postgres@docker-desktop:~$ psql -p 5433
psql (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
Type "help" for help.
postgres=#
postgres=#
postgres=# \dt
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
```

```
 public | numbers | table | postgres
(1 row)
```

```
# Yes our table is replicated now lets stop the standby node and perform some inserts on primary 
inside newly created table
postgres@docker-desktop:~$ /usr/lib/postgresql/16/bin/pg_ctl -D standby -l logfile stop
waiting for server to shut down.... done
server stopped

# Replication status has stoped can be verified from primary
postgres=# SELECT pid,client_addr,usename,application_name,state,sync_state FROM 
pg_stat_replication;
 pid | client_addr | usename | application_name | state | sync_state
-----+-------------+---------+------------------+-------+------------
(0 rows)
```

```
# Standby is stopped but we can still able to perform inserts on primary node
postgres=# INSERT INTO numbers (ID) SELECT generate_series(1, 50000000);
INSERT 0 50000000
postgres=# select count(*) from numbers;
  count
----------
 50000000
(1 row)
```

```
Copy to Clipboard
```

That means asynchronous replication gives us the flexibility that in case of standby failure or primary will not stop working it will handle all the requests without any interruption

Drawback

Data loss: The lack of awareness on the primary node regarding standby node failures or disconnections poses a risk of data loss in streaming replication. This risk stems from the primary node’s deletion of WAL files necessary for the standby node, as it removes these files after applying changes locally.

In the example mentioned earlier, we created a table that was replicated to the standby node. After halting the standby node, we proceeded to insert nearly 50 million rows. Now, as we restart the standby node, we anticipate no data loss, but we will verify this outcome.

Staring the standby node again

```
postgres@docker-desktop:~$ /usr/lib/postgresql/16/bin/pg_ctl -D standby -l logfile start
waiting for server to start.... done
server startedCopy to Clipboard
```

Lets check logs for standby node

```
postgres@docker-desktop:~$ tail logfile
2024-05-16 13:00:51.352 PKT [7072] LOG:  waiting for WAL to become available at 0/3002000
2024-05-16 13:00:56.352 PKT [7076] LOG:  started streaming WAL from primary at 0/3000000 on 
timeline 1
2024-05-16 13:00:56.352 PKT [7076] FATAL:  could not receive data from WAL stream: 
ERROR:  requested WAL segment 000000010000000000000003 has already been removed
2024-05-16 13:00:56.353 PKT [7072] LOG:  waiting for WAL to become available at 0/3002000
2024-05-16 13:01:01.357 PKT [7077] LOG:  started streaming WAL from primary at 0/3000000 on
```

```
timeline 1
2024-05-16 13:01:01.357 PKT [7077] FATAL:  could not receive data from WAL stream:
ERROR:  requested WAL segment 000000010000000000000003 has already been removed
2024-05-16 13:01:01.358 PKT [7072] LOG:  waiting for WAL to become available at 0/3002000
2024-05-16 13:01:06.359 PKT [7078] LOG:  started streaming WAL from primary at 0/3000000 on
timeline 1
2024-05-16 13:01:06.359 PKT [7078] FATAL:  could not receive data from WAL stream:
ERROR:  requested WAL segment 000000010000000000000003 has already been removed
2024-05-16 13:01:06.359 PKT [7072] LOG:  waiting for WAL to become available at 0/3002000Copy to
Clipboard
```

Oh! Standby node is looking for some WAL files but these WAL files are already been removed by the primary means **data loss**

If we run the same count query on standby we will not see anything

```
postgres@docker-desktop:~$ psql -p 5433
psql (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
Type "help" for help.
postgres=# select count(*) from numbers;
 count
-------
     0
(1 row)Copy to Clipboard
```

This is expected behaviour from the primary as it wasnt aware about standby being stopped so it removed the all the WAL files. So we encountered data loss here

Where to use Asynchronous Replication?

E-commerce Websites: Asynchronous replication offers a good balance between performance and availability for e-commerce websites. Users can browse products and add items to their carts without delays, while purchase confirmations and inventory updates are replicated asynchronously to secondary servers. This ensures high responsiveness for user interactions while handling large volumes of data efficiently.

Content Delivery Networks (CDNs): Replicating content across geographically dispersed servers in a CDN is a prime example for asynchronous replication. Asynchronous updates ensure new content eventually reaches all edge servers, improving website loading times for users in different locations. Waiting for confirmation from every edge server would significantly slow down content delivery.

Synchronous Replication – Advantage and Drawback

Synchronous replication in PostgreSQL prioritizes ironclad data consistency at the expense of some performance. Here’s how it works: changes made on the primary server (inserts, updates, deletes) are not considered fully committed until they’ve been successfully replicated and acknowledged by all designated standby servers. This meticulous approach ensures that all replicas possess an identical copy of the data at any given moment. While this guarantees the highest level of consistency, it can introduce some performance overhead as the primary server waits for confirmation before completing the transaction. This method is ideal for scenarios where even the slightest data discrepancy is unacceptable.

Advantages

Guaranteed Data Consistency: Synchronous replication ensures immediate consistency between the primary and replica databases by waiting for confirmation from the replica before committing transactions. This guarantees that both databases are always in sync, reducing the risk of data inconsistencies.

Disadvantages

Performance Impact: Synchronous replication can introduce performance overhead on the primary database due to the need to wait for confirmation from the replica before committing transactions. This waiting time can increase transaction latency and reduce the overall throughput of write operations, impacting the performance of the primary database, especially in high-traffic environments.

Let’s explore how synchronous replication can impact of performance. We’ll configure synchronous replication between two nodes: one designated as the primary and the other as the standby. By halting the standby node and attempting to insert data into the primary, we’ll observe how the primary node responds in such a scenario.

- Initialize the primary node
- Update postgresql.conf and pg_hba.conf for primary
- Run pg_basebackup on standby node
- Update port inside postgresql.conf file for standby node
- Start the standby node
  Now in order to make replication synchronous update following parameter accordingly on the primary side. We can get application name from **pg_stat_replication** query

```
synchronous_standby_names = ‘walreceiver’Copy to Clipboard
```

Restart both primary and standby nodes. Connect to the primary node and run this query

```
postgres@docker-desktop:~$ psql
psql (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
Type "help" for help.
postgres=# SELECT pid,client_addr,usename,application_name,state,sync_state FROM
pg_stat_replication;
 pid  | client_addr  | usename  | application_name |   state   | sync_state
------+--------------+----------+------------------+-----------+------------
 8927 | 192.168.65.3 | postgres | walreceiver      | streaming | sync
(1 row)Copy to Clipboard
```

We can see our replication is now synchronous

Let’s proceed by creating a table on the primary node. Afterward, we’ll halt the standby node and attempt to perform an insert operation on the primary node. This will allow us to observe the behavior of the primary node in response to this action.

```
# Creating table on primary
postgres=# CREATE TABLE numbers (ID SERIAL PRIMARY KEY);
CREATE TABLE
```

```
# Testing replication on standby node
postgres@docker-desktop:~$ psql -p 5433
psql (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
Type "help" for help.
postgres=# \d
               List of relations
 Schema |      Name      |   Type   |  Owner
--------+----------------+----------+----------
 public | numbers        | table    | postgres
 public | numbers_id_seq | sequence | postgres
(2 rows)
```

```
# Stopping the standby node
postgres@docker-desktop:~$ /usr/lib/postgresql/16/bin/pg_ctl -D standby -l logfile stop
```

```
waiting for server to shut down.... done
server stopped
```

```
# Checking replication status - Replication has stopped
postgres=# SELECT pid,client_addr,usename,application_name,state,sync_state FROM 
pg_stat_replication;
 pid | client_addr | usename | application_name | state | sync_state
-----+-------------+---------+------------------+-------+------------
(0 rows)
```

```
# Issue insert query from primary node and it will halt forever
postgres=# INSERT INTO numbers (ID) SELECT generate_series(1, 50000000);
```

Copy to Clipboard

This means that primary is now waiting for a response from standby node before committing changes locally since standby node is down so primary will hang forever

Where to use Synchronous Replication?

High-Frequency Trading Platforms: In these fast-paced environments, data consistency is paramount. Synchronous replication ensures all trades are reflected identically across geographically dispersed servers, preventing discrepancies that could lead to financial losses. Even a slight delay could have significant consequences.

Mission-Critical Applications (e.g., Healthcare): For applications handling sensitive patient data, synchronous replication guarantees consistency. This ensures all updates to patient records, medication administration, or other critical information are immediately reflected on all standby servers. Even a small inconsistency could potentially harm patient care.

NOTE: Use physical replication slots in streaming replication to avoid Wal files being deleted and in this case we will not loss our data.