PostgreSQL Physical Replication, Easy Guide

Constantin Alexander
Cloud Data Architect | DeltaLake + Apache Iceberg | Apache Flink + Apache Fluss | Data Governance | Ai Ops | 🐘...

September 22, 2025

PostgreSQL Physical Replication:Complete Guide with Examples

Introduction

PostgreSQL physical replication aka streaming replication, is a method of creating exact binary copies of a PostgreSQL database cluster. Unlike logical replication, which replicates data changes at the SQL level, physical replication streams Write-Ahead Log (WAL) records from the primary server to one or more standby servers, creating identical copies at the storage level.

Physical replication is the center of PostgreSQL's high availability, disaster recovery, and read scaling solutions. It provides strong consistency guarantees and automatic failover capabilities, making it the preferred choice for mission-critical applications.

Key Concepts

Primary Server

The main database server that accepts write operations and generates WAL records. Also called the "master" server in older PostgreSQL versions.

Standby Server

A replica server that receives and applies WAL records from the primary. Standby servers can serve read-only queries when configured as "hot standby" servers.

WAL (Write-Ahead Log)

The transaction log that records all database changes before they're applied to the actual data files. WAL records are streamed to standby servers for replication.

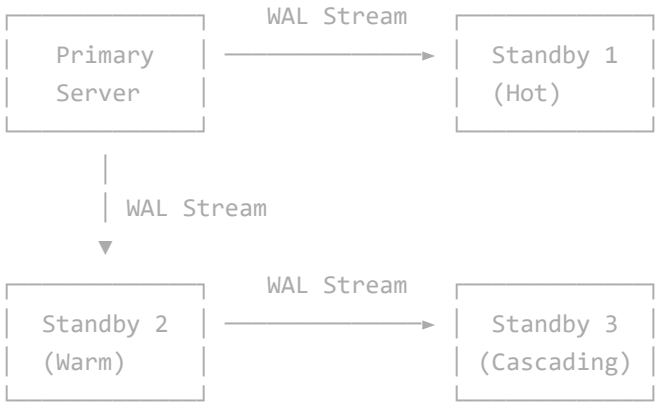
Streaming Replication

The method of continuously streaming WAL records from primary to standby servers over a network connection.

Hot Standby

A standby server configuration that allows read-only queries while continuously applying changes from the primary.

Architecture Overview



Prerequisites and Configuration

System Requirements

- PostgreSQL 9.0 or later (examples use PostgreSQL 12+)
- Network connectivity between servers
- Sufficient disk space for WAL files
- Same PostgreSQL major version on all servers
- Same CPU architecture (x86_64, ARM, etc.)

Basic Configuration Parameters

The following parameters must be configured on the primary server in postgresql.conf:

```
# Enable replication
wal_level = replica

# Number of concurrent connections from standby servers
max_wal_senders = 10

# Number of replication slots (optional but recommended)
max_replication_slots = 10

# WAL archiving (optional but recommended)
archive_mode = on
archive_command = 'cp %p
/var/lib/postgresql/wal_archive/%f'

# WAL retention (prevents WAL files from being deleted too
early)
wal_keep_size = 1GB

# Checkpoint settings (optional optimization)
checkpoint_completion_target = 0.9
```

Step-by-Step Setup Guide

Step 1: Configure the Primary Server

First, let's set up a primary server with sample data:

```
-- Connect to PostgreSQL as superuser
sudo -u postgres psql

-- Create a sample database
CREATE DATABASE company;
\c company

-- Create sample tables
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    department VARCHAR(100),
    salary DECIMAL(10,2),
    hire_date DATE DEFAULT CURRENT_DATE
);
```

```
CREATE TABLE departments (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) UNIQUE NOT NULL,
    budget DECIMAL(15,2),
    manager_id INTEGER REFERENCES employees(id)
);

-- Insert sample data
INSERT INTO departments (name, budget) VALUES
    ('Engineering', 5000000.00),
    ('Sales', 2000000.00),
    ('Marketing', 1500000.00);

INSERT INTO employees (name, department, salary) VALUES
    ('Alice Johnson', 'Engineering', 95000.00),
    ('Bob Smith', 'Engineering', 87000.00),
    ('Carol Davis', 'Sales', 65000.00),
    ('David Wilson', 'Marketing', 58000.00);
```

Configure PostgreSQL for replication in /etc/postgresql/14/main/postgresql.conf:

```
# Edit postgresql.conf
sudo nano /etc/postgresql/14/main/postgresql.conf

# Add or modify these lines:
listen_addresses = '*'
wal_level = replica
max_wal_senders = 10
max_replication_slots = 10
wal_keep_size = '1GB'
hot_standby = on
archive_mode = on
archive_command = 'cp %p
/var/lib/postgresql/14/main/wal_archive/%f'

# Create WAL archive directory
sudo mkdir -p /var/lib/postgresql/14/main/wal_archive
sudo chown postgres:postgres
/var/lib/postgresql/14/main/wal_archive
```

Step 2: Configure Authentication

Edit /etc/postgresql/14/main/pg_hba.conf to allow replication connections:

```
# Add these lines for replication
# TYPE  DATABASE        USER            ADDRESS
METHOD
host    replication     replicator      192.168.1.0/24
md5
host    all             replicator      192.168.1.0/24
md5

# For local testing, you can use:
host    replication     replicator      127.0.0.1/32
md5
```

Create a replication user:

```
-- Create replication user
CREATE USER replicator WITH REPLICATION LOGIN PASSWORD
'secure_password';

-- Grant connection privileges to databases
GRANT CONNECT ON DATABASE company TO replicator;
```

Restart PostgreSQL to apply configuration changes:

```
sudo systemctl restart postgresql
```

Step 3: Create the First Standby Server

Method 1: Using pg_basebackup (Recommended)

On the standby server, stop PostgreSQL and create a base backup:

```
# Stop PostgreSQL on standby
sudo systemctl stop postgresql

# Remove existing data directory
sudo rm -rf /var/lib/postgresql/14/main

# Create base backup from primary
sudo -u postgres pg_basebackup \
    -h 192.168.1.100 \
    -U replicator \
    -D /var/lib/postgresql/14/main \
    -P \
    -W \
    -R

# The -R flag automatically creates standby.signal and
sets up recovery configuration
```

Method 2: Manual Setup

If not using the -R flag, manually configure the standby:

```
# Create standby.signal file
sudo -u postgres touch
/var/lib/postgresql/14/main/standby.signal

# Configure recovery in postgresql.conf
sudo -u postgres tee -a
/var/lib/postgresql/14/main/postgresql.conf << EOF

# Standby server configuration
primary_conninfo = 'host=192.168.1.100 port=5432
user=replicator password=secure_password
application_name=standby1'
hot_standby = on
EOF
```

Start the standby server:

```
sudo systemctl start postgresql
```

Step 4: Verify Replication Setup

On the primary server, check replication status:

```
-- View active replication connections
SELECT
    pid,
    client_addr,
    client_hostname,
    client_port,
    application_name,
    state,
    sent_lsn,
```

```
    write_lsn,
    flush_lsn,
    replay_lsn,
    write_lag,
    flush_lag,
    replay_lag,
    sync_state,
    sync_priority
FROM pg_stat_replication;

-- Check replication slots
SELECT
    slot_name,
    slot_type,
    database,
    active,
    restart_lsn,
    confirmed_flush_lsn
FROM pg_replication_slots;
```

On the standby server, verify it's receiving data:

```
-- Check if server is in recovery mode
SELECT pg_is_in_recovery();  -- Should return 't' (true)

-- Check last received WAL
SELECT
    pg_last_wal_receive_lsn(),
    pg_last_wal_replay_lsn(),
    pg_last_xact_replay_timestamp();

-- Calculate replication lag
SELECT
    EXTRACT(EPOCH FROM (now() -
pg_last_xact_replay_timestamp())) AS lag_seconds;
```

Step 5: Test Replication

On the primary server, insert new data:

```
-- Insert new employees
INSERT INTO employees (name, department, salary) VALUES
    ('Eve Brown', 'Engineering', 78000.00),
    ('Frank Miller', 'Sales', 72000.00);

-- Update existing records
UPDATE employees SET salary = salary * 1.05 WHERE
department = 'Engineering';

-- Check current data on primary
SELECT * FROM employees ORDER BY id;
```

On the standby server, verify the data was replicated:

```
-- Connect to standby (read-only)
SELECT * FROM employees ORDER BY id;

-- Verify we cannot write to standby
INSERT INTO employees (name, department, salary) VALUES
    ('Should Fail', 'Test', 50000.00);
-- This should fail with: "cannot execute INSERT in a
read-only transaction"
```

Advanced Configuration Examples

Synchronous Replication

For applications requiring guaranteed durability, configure synchronous replication:

On Primary Server

```
-- Configure synchronous standby names in postgresql.conf
synchronous_standby_names = 'standby1,standby2'

-- Or use ANY syntax for flexibility
synchronous_standby_names = 'ANY 1
(standby1,standby2,standby3)'

-- Or use FIRST syntax for priority
synchronous_standby_names = 'FIRST 2
(standby1,standby2,standby3)'
```

Testing Synchronous Replication

```
-- On primary, insert data and measure commit time
\timing on

BEGIN;
INSERT INTO employees (name, department, salary) VALUES
('Sync Test', 'Engineering', 80000.00);
COMMIT;  -- This will wait for standby confirmation

\timing off
```

Cascading Replication

Set up a multi-tier replication hierarchy:

Configure Intermediate Standby as Primary

On the intermediate standby server, enable it to accept connections from downstream standbys:

```
# In postgresql.conf on intermediate standby
max_wal_senders = 5
hot_standby = on
```

Configure Downstream Standby

```
# On downstream standby, point to intermediate standby
primary_conninfo = 'host=intermediate_standby_ip port=5432
user=replicator password=secure_password
application_name=downstream1'
```

Replication Slots

Use replication slots for better WAL retention management:

Create Replication Slots on Primary

```
-- Create physical replication slot
SELECT
pg_create_physical_replication_slot('standby1_slot');

-- View replication slots
SELECT slot_name, slot_type, database, active, restart_lsn
FROM pg_replication_slots;
```

Configure Standby to Use Slot

```
# In postgresql.conf on standby
primary_slot_name = 'standby1_slot'
```

High Availability with pg_auto_failover

For automated failover, consider using pg_auto_failover:

```
# Install pg_auto_failover
sudo apt-get install postgresql-14-auto-failover

# Initialize monitor node
pg_autoctl create monitor --hostname monitor-hostname --
auth trust

# Create primary node
pg_autoctl create postgres --hostname primary-hostname --
auth trust --monitor postgres://monitor-
hostname:5432/pg_auto_failover

# Create standby node
pg_autoctl create postgres --hostname standby-hostname --
auth trust --monitor postgres://monitor-
hostname:5432/pg_auto_failover
```

Monitoring Physical Replication

Essential Monitoring Queries

Primary Server Monitoring

```
-- Current replication status overview
SELECT
    application_name,
    client_addr,
    state,
    write_lag,
    flush_lag,
    replay_lag,
    sync_state
FROM pg_stat_replication
ORDER BY application_name;

-- WAL generation rate
SELECT
    (pg_current_wal_lsn() - '0/0') / 1024 / 1024 AS
current_wal_mb,
    pg_walfile_name(pg_current_wal_lsn()) AS
current_wal_file;

-- Replication lag in bytes
SELECT
    application_name,
    client_addr,
    pg_wal_lsn_diff(pg_current_wal_lsn(), flush_lsn) AS
lag_bytes,
    pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn) AS
replay_lag_bytes
FROM pg_stat_replication;
```

Standby Server Monitoring

```
-- Recovery status
SELECT
    pg_is_in_recovery() AS in_recovery,
```

```
    pg_is_wal_replay_paused() AS replay_paused,
    pg_last_wal_receive_lsn() AS last_received,
    pg_last_wal_replay_lsn() AS last_replayed;

-- Recovery progress
SELECT
    pg_last_xact_replay_timestamp() AS last_replay_time,
    EXTRACT(EPOCH FROM (now() -
pg_last_xact_replay_timestamp())) AS lag_seconds;

-- Applied vs received WAL
SELECT
    pg_wal_lsn_diff(pg_last_wal_receive_lsn(),
pg_last_wal_replay_lsn()) AS receive_replay_lag_bytes;
```

Automated Monitoring Script

```
#!/bin/bash
# PostgreSQL replication monitoring script

PGHOST="localhost"
PGPORT="5432"
PGUSER="postgres"
PGDATABASE="postgres"

# Check if server is primary or standby
IS_PRIMARY=$(psql -h $PGHOST -p $PGPORT -U $PGUSER -d
$PGDATABASE -t -c "SELECT NOT pg_is_in_recovery();" |
xargs)

echo "=== PostgreSQL Replication Status ==="
echo "Server: $PGHOST:$PGPORT"
echo "Role: $([ "$IS_PRIMARY" = "t" ] && echo "Primary" ||
echo "Standby")"
echo "Time: $(date)"
echo

if [ "$IS_PRIMARY" = "t" ]; then
    # Primary server checks
    echo "--- Replication Connections ---"
    psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE -
c "
        SELECT
            application_name,
            client_addr,
            state,
            write_lag,
            flush_lag,
            replay_lag
        FROM pg_stat_replication
        ORDER BY application_name;
    "

    echo "--- WAL Status ---"
    psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE -
c "
        SELECT
            pg_current_wal_lsn() as current_wal_lsn,
            pg_walfile_name(pg_current_wal_lsn()) as
current_wal_file;
    "
else
    # Standby server checks
    echo "--- Recovery Status ---"
    psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE -
c "
        SELECT
            pg_last_wal_receive_lsn() as last_received,
            pg_last_wal_replay_lsn() as last_replayed,
            pg_last_xact_replay_timestamp() as
last_replay_time,
            EXTRACT(EPOCH FROM (now() -
pg_last_xact_replay_timestamp())) as lag_seconds;
    "
fi
```

```
echo "--- Connection Count ---"
psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE -c "
    SELECT
        count(*) as total_connections,
        count(*) FILTER (WHERE state = 'active') as
active_connections
    FROM pg_stat_activity;
"
```

Failover Procedures

Manual Failover

When the primary server fails, promote a standby to become the new primary:

Step 1: Promote Standby

```
# On the standby server, promote it to primary
sudo -u postgres pg_ctl promote -D
/var/lib/postgresql/14/main

# Or using SQL (PostgreSQL 12+)
psql -c "SELECT pg_promote();"
```

Step 2: Verify Promotion

```
-- Check that server is no longer in recovery
SELECT pg_is_in_recovery();  -- Should return 'f' (false)

-- Verify write capability
INSERT INTO employees (name, department, salary) VALUES
    ('Failover Test', 'Engineering', 85000.00);
```

Step 3: Update Application Connection Strings

Update your application configuration to point to the new primary server.

Step 4: Reattach Remaining Standbys

Other standby servers need to be reconfigured to follow the new primary:

```
# Stop standby
sudo systemctl stop postgresql

# Update primary connection info
sudo -u postgres tee
/var/lib/postgresql/14/main/postgresql.conf << EOF
primary_conninfo = 'host=new_primary_ip port=5432
user=replicator password=secure_password
application_name=standby2'
EOF

# Start standby
sudo systemctl start postgresql
```

Automated Failover with Patroni

For production environments, consider using Patroni for automated failover:

```
# patroni.yml configuration example
scope: postgres-cluster
name: node1
```

```
restapi:
  listen: 0.0.0.0:8008
  connect_address: node1:8008

etcd:
  hosts: etcd1:2379,etcd2:2379,etcd3:2379

bootstrap:
  dcs:
    ttl: 30
    loop_wait: 10
    retry_timeout: 30
    maximum_lag_on_failover: 1048576
    postgresql:
      use_pg_rewind: true
      parameters:
        max_connections: 200
        max_worker_processes: 8

postgresql:
  listen: 0.0.0.0:5432
  connect_address: node1:5432
  data_dir: /var/lib/postgresql/14/main
  authentication:
    replication:
      username: replicator
      password: secure_password
    superuser:
      username: postgres
      password: postgres_password
```

Performance Optimization

Network Optimization

```
# Optimize network buffer sizes for replication
# In postgresql.conf on both primary and standby
tcp_keepalives_idle = 600
tcp_keepalives_interval = 30
tcp_keepalives_count = 3

# Increase shared buffers for better performance
shared_buffers = 256MB  # Adjust based on available RAM

# Optimize WAL settings
wal_buffers = 16MB
wal_writer_delay = 200ms
```

Storage Optimization

```
# Use separate disks for WAL and data if possible
# In postgresql.conf
wal_sync_method = fsync  # or fdatasync on Linux
fsync = on
synchronous_commit = on  # or off for async replication

# Optimize checkpoint behavior
checkpoint_completion_target = 0.9
checkpoint_timeout = '15min'
max_wal_size = '2GB'
```

Monitoring Performance

```
-- Monitor WAL generation rate
SELECT
    name,
    setting,
```

```
    unit
FROM pg_settings
WHERE name IN ('wal_level', 'max_wal_senders',
'wal_keep_size');

-- Check WAL file size and count
SELECT
    count(*) as wal_file_count,
    sum(size) as total_wal_size
FROM pg_ls_waldir();

-- Monitor replication throughput
SELECT
    application_name,
    pg_wal_lsn_diff(sent_lsn, '0/0') / 1024 / 1024 AS
sent_mb,
    pg_wal_lsn_diff(flush_lsn, '0/0') / 1024 / 1024 AS
flushed_mb
FROM pg_stat_replication;
```

Troubleshooting Common Issues

Replication Lag

```
-- Identify slow queries on primary
SELECT
    pid,
    now() - pg_stat_activity.query_start AS duration,
    query
FROM pg_stat_activity
WHERE (now() - pg_stat_activity.query_start) > interval '5
minutes'
AND state = 'active';

-- Check for long-running transactions
SELECT
    pid,
    state,
    now() - xact_start as xact_duration,
    now() - query_start as query_duration,
    query
FROM pg_stat_activity
WHERE xact_start IS NOT NULL
ORDER BY xact_start;
```

Connection Issues

```
-- Check connection limits
SELECT
    setting::int as max_connections,
    count(*) as current_connections,
    setting::int - count(*) as available_connections
FROM pg_settings, pg_stat_activity
WHERE name = 'max_connections'
GROUP BY setting;

-- Verify replication user permissions
SELECT
    rolname,
    rolsuper,
    rolreplication,
    rolcanlogin
FROM pg_roles
WHERE rolname = 'replicator';
```

WAL File Management

```
-- Check WAL retention settings
SELECT name, setting, unit FROM pg_settings
WHERE name IN ('wal_keep_size', 'max_wal_size',
'min_wal_size');

-- Monitor WAL file accumulation
SELECT count(*) as wal_files FROM pg_ls_waldir();

-- Check for inactive replication slots
SELECT
    slot_name,
    slot_type,
    database,
    active,
    restart_lsn,
    pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn) AS
retained_bytes
FROM pg_replication_slots
WHERE NOT active;
```

Best Practices

Security

1. **Use strong passwords** for replication users
2. **Restrict network access** using pg_hba.conf and firewalls
3. **Enable SSL/TLS** for replication connections
4. **Use certificates** for mutual authentication
5. **Monitor replication user access** regularly

Performance

1. **Monitor replication lag** continuously
2. **Use replication slots** to prevent WAL file removal
3. **Optimize checkpoint settings** for your workload
4. **Consider using synchronous replication** for critical data
5. **Monitor disk space** on all servers

Operational

1. **Test failover procedures** regularly
2. **Document your replication topology**
3. **Monitor standby server health**
4. **Plan for network partitions**
5. **Keep standby servers in sync with primary configuration**

Disaster Recovery

1. **Implement multiple standby servers** in different locations
2. **Regular backup verification** on standby servers
3. **Document recovery procedures**
4. **Test restore procedures** from backups
5. **Monitor backup retention policies**

Conclusion

PostgreSQL physical replication provides a great foundation for high availability, disaster recovery, and read scaling. Its binary-level replication ensures strong consistency and automatic synchronization, making it ideal for mission-critical applications.

Key advantages of physical replication:

- **Strong consistency** - Standbys are exact copies of the primary
- **Automatic synchronization** - No manual intervention required
- **High performance** - Minimal overhead on primary server
- **Simple setup** - Well-integrated with PostgreSQL core
- **Flexible deployment** - Supports various architectures and use cases

By following the examples and best practices in this guide, you can implement a reliable PostgreSQL replication solution that meets your organization's availability and scalability requirements.

Remember to regularly test your replication setup, monitor performance metrics, and maintain proper documentation of your configuration for successful long-term operations.

Constantin Alexander

Cloud Data Architect | DeltaLake + Apache Iceberg | Apache Flink + Apache Fluss | Data Governance | Ai Ops | 🐘 PostgreSQL support 🐘 | AWS & GCP Specialist | Data Modeling | AI | DevOps

Azadkumar followed

✓ Following