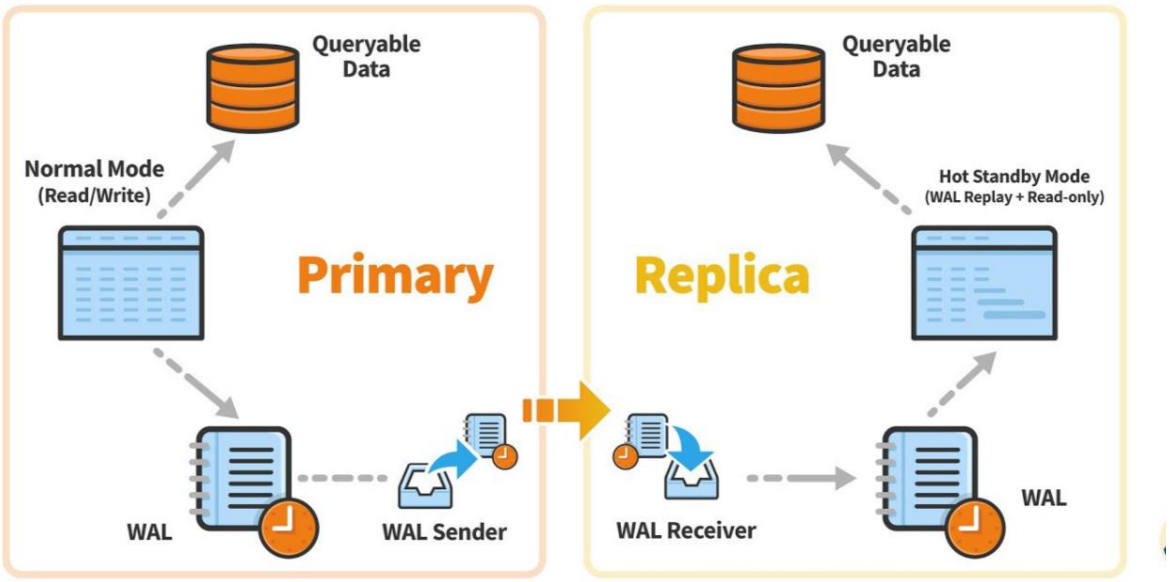
MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

DAY – 5

TOPIC: High Availability, Load Balancing, and Replication in Postgresql



High Availability, Load Balancing, and Replication in Postgre

Database servers are often deployed in configurations that allow them to work together to achieve several critical goals:

- **High Availability (HA):** Ensuring that database services remain accessible even if a primary server fails, with minimal downtime.
- **Load Balancing:** Distributing read (and sometimes write) workloads across multiple servers to improve performance and scalability.
- **Replication:** Maintaining identical copies of data on multiple servers to support HA, load balancing, or disaster recovery.

Ideally, database servers could work together seamlessly. While web servers serving static content are easily load-balanced, databases, especially those with a mix of read/write requests, present a significant challenge. The fundamental difficulty is **synchronization**: any write operation performed on one server must be reliably propagated to all other servers to ensure data consistency across the cluster.

Since no single solution perfectly eliminates the impact of this synchronization problem for all use cases, Postgre offers multiple approaches. Each solution addresses synchronization differently, optimizing for specific workloads and requirements.

Key Concepts and Terminology:

- **Read/Write (Master/Primary) Server:** The server that is allowed to modify the data.
- **Standby (Secondary) Server:** A server that tracks changes made on the primary.
    - **Warm Standby:** A standby server that cannot accept connections until it is promoted to a primary. It's essentially a ready-to-go recovery point.
    - **Hot Standby:** A standby server that can accept read-only queries while continuously recovering from the primary. This enables read load balancing.
- **Synchronous Solutions:** A data-modifying transaction is not considered committed until it has been safely written and acknowledged by *all* participating servers. This guarantees zero data loss upon failover and ensures all load-balanced servers return consistent results. However, it can incur significant performance overhead, especially over slow networks.
- **Asynchronous Solutions:** A data-modifying transaction is considered committed on the primary before its propagation to other servers is complete. This introduces a potential for minor data loss in a failover scenario (transactions committed on the primary but not yet replicated) and might lead to slightly stale results on load-balanced standbys. Asynchronous communication is used when synchronous performance impact is unacceptable.
- **Granularity:** Solutions can operate at different levels:
    - **Entire Database Server:** Most Postgre replication solutions operate at the cluster level.
    - **Per-Table or Per-Database:** Logical replication (Postgre 10+) offers this flexibility.
- **Performance Trade-off:** There is always a trade-off between functionality (e.g., synchronous guarantee) and performance. A fully synchronous solution over a high-latency network can significantly reduce transaction throughput.

This section will outline various failover, replication, and load balancing solutions available in Postgre.

.1. Comparison of Different Solutions

MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

Before diving into specifics, it's useful to understand the landscape of Postgre HA/Replication.

Feature / Solution	**Dump (pg_dump)**	**File System Backup (Offline)**	**Continuous Archiving / Physical Replication (Streaming Replication)**	**Logical Replication (CREATE PUBLICATION/SUBSCRIPTION)**	**Third-Party Tools (e.g., Patroni, Repmgr)**
Type	Logical Backup	Physical Backup	Physical Replication (WAL-based)	Logical Replication (Row-based)	Cluster Management (often built on Streaming Replication)
Purpose	Point-in-time snapshot, migration, dev/test setups	Disaster Recovery, Base for PITR	HA, DR, Read Scalability (Hot Standby), PITR	Selective Replication, Data Distribution, Upgrade Path	Automated Failover, Clustering, Monitoring
Data Loss on Failover	N/A (backup, not HA)	N/A (backup, not HA)	Zero (Synchronous) / Minimal (Asynchronous)	Minimal (Asynchronous)	Zero (Synchronous) / Minimal (Asynchronous)
Consistency	Point-in-time snapshot	Point-in-time snapshot (must be offline or snapshot	Extremely High (via WAL replay)	Transactional (per table/db), but can be eventual	High (depends on underlying replication)

Feature / Solution	Dump (pg_dump)	File System Backup (Offline)	Continuous Archiving / Physical Replication (Streaming Replication)	Logical Replication (CREATE PUBLICATION/SUBSCRIPTION)	Third-Party Tools (e.g., Patroni, Repmgr)
		consistent)			
Read Scalability	No	No	Yes (Hot Standby)	Yes	Yes
Write Scalability	No	No	No (Single Primary)	Limited (Multi-master possible but complex)	No (Single Primary)
Granularity	Per database, schema, table	Entire cluster	Entire cluster	Per database, table, or set of tables	Entire cluster
Version Comp.	High (can restore to newer PG versions)	Low (tied to exact PG version & OS)	Low (tied to exact PG version & often OS)	High (can replicate between different major PG versions)	Depends on underlying technology
Complexity	Low	Low (offline) / Medium (snapshot)	Medium to High (setup, monitoring)	Medium (setup, conflict resolution)	High (setup, external dependencies)

.2. Log-Shipping Standby Servers (Physical Replication)

This is the most common and robust form of Postgre replication, enabling both high availability and read scalability. It relies on the Postgre **Write-Ahead Log (WAL)**.

.2.1. Planning

- **Hardware:** Primary and standby servers should have similar hardware for consistent performance.
- **Network:** A reliable, low-latency, high-bandwidth network connection between primary and standbys is critical, especially for synchronous replication.
- **Storage:** Sufficient storage on both primary and standbys for data and WAL files.
- **WAL Archiving Strategy:** A robust WAL archiving strategy is the foundation, even if streaming replication is used, as it provides a safety net for PITR and can cover network outages.
- **Failover Strategy:** Plan how failover will be detected and managed (manual, semi-automated, fully automated).
- **Monitoring:** Implement comprehensive monitoring for replication lag, disk space, and server health.

.2.2. Standby Server Operation

A standby server operates by continuously receiving and applying WAL records from the primary. It is essentially in a continuous recovery mode.

- **Recovery:** When a standby starts, it reads its PGDATA directory, looks for the standby.signal file (or recovery.signal for one-off recovery), and starts applying WAL records.
- **Recovery Target:** It applies WAL records until it reaches a specific recovery_target (e.g., end of available WAL, a specific time, LSN, or transaction ID) or, in the case of a hot/warm standby, continuously applies them as they arrive.
- **Read-Only:** While in recovery, the standby is read-only. For Hot Standby, it can serve read queries, but any write attempt will fail.

2.3. Preparing the Primary for Standby Servers

The primary server needs to be configured to generate and provide the necessary WAL information for standbys.

postgre.conf settings on Primary:

Ini, TOML

wal_level: Determines the amount of information written to the WAL.

'replica' is sufficient for physical replication (streaming and archiving).

'minimal' is not enough.

wal_level = replica

archive_mode: Enables or disables WAL archiving. Recommended even with streaming.

'on' enables archiving.

archive_mode = on

archive_command: Shell command to archive WAL segments.

Essential for continuous archiving and PITR.

Example: copy to a shared NFS directory

archive_command = 'test ! -f /mnt/nfs_archive/%f && cp %p /mnt/nfs_archive/%f'

max_wal_senders: Maximum number of concurrent connections from standbys (WAL senders).

Each standby, plus pg_basebackup, consumes a slot.

max_wal_senders = 10 # Adjust based on number of standbys + backup needs

wal_keep_size (or wal_keep_segments in older versions):

Amount of WAL to retain in pg_wal for standbys that might fall behind.

Ensures that a standby doesn't need to go to the archive too quickly.

Measured in MB (or 16MB segments in older versions).

wal_keep_size = 1024 # Keep 1GB of WAL on primary

If synchronous replication is desired (covered later)

synchronous_commit = on # 'on' or 'remote_write', 'local' etc.

synchronous_standby_names = 'ANY 1 (standby_1, standby_2)' # Names of standbys for synchronous commit

Restart the Primary: After changing wal_level and archive_mode, you must restart the primary server for these changes to take effect.

2.4. Setting Up a Standby Server

Setting up a standby involves creating a base backup of the primary and then configuring the standby to connect and recover from the primary's WAL stream.

Steps:

1. **Create Base Backup:** Use pg_basebackup from the primary to create an initial copy of the data directory for the standby. This is performed from the standby server or a separate client that can access the primary.

On the standby server or a client machine:

Ensure the Postgre data directory is empty or doesn't exist.

Make sure the 'postgres' user (or whatever user Postgre runs as) owns the target directory.

Example:

mkdir -p /var/lib/postgre/16/main_standby

chown -R postgres:postgres /var/lib/postgre/16/main_standby

chmod 0700 /var/lib/postgre/16/main_standby

pg_basebackup -h primary_ip -U replication_user -D /var/lib/postgre/16/main_standby -Fp -P -v -R -c fast -w

- -h primary_ip: IP address or hostname of the primary server.
- -U replication_user: A dedicated replication user (must have REPLICATION privilege on the primary, often postgres user initially).
- -D /var/lib/postgre/16/main_standby: Destination directory for the standby's data.
- -Fp: Plain format (creates a regular directory structure).
- -P: Show progress.
- -v: Verbose output.

- -R: **Crucial.** This automatically creates a standby.signal file and appends a primary_conninfo entry to the postgre.conf file in the base backup. This prepares the standby for streaming replication.
- -c fast: Forces a checkpoint on the primary for faster base backup start.
- -w: Prompt for password for the replication user.

2. **Configure postgre.conf on Standby:**
   - The -R option usually handles primary_conninfo. You might want to adjust other settings.
   - If using restore_command for archive recovery (as a fallback or for PITR):

Ini, TOML

In /var/lib/postgre/16/main_standby/postgre.conf

This is typically added by -R, but verify/adjust:

primary_conninfo = 'host=primary_ip port=5432 user=replication_user password=your_password application_name=standby_1'

If you want Hot Standby (read-only queries on standby)

hot_standby = on

For archive recovery fallback

restore_command = 'cp /mnt/nfs_archive/%f %p' # Or your actual archive command

3. **Create standby.signal (if not using -R or custom setup):**
   - If -R was not used, manually create an empty file named standby.signal in the root of the standby's data directory. This signals Postgre to start in standby mode.
4. **Start Standby Server:**

sudo systemctl start postgre@16-main_standby # Or appropriate command

The standby server will start, connect to the primary, and begin streaming WAL records. You can check pg_stat_replication on the primary to confirm the connection.

2.5. Streaming Replication

Streaming replication is the mechanism by which the standby continuously receives WAL records directly from the primary, rather than waiting for full WAL segments to be archived. This significantly reduces replication lag.

- **WAL Sender (Primary):** The primary server runs a "WAL sender" process for each connected standby.
- **WAL Receiver (Standby):** The standby server runs a "WAL receiver" process that connects to the WAL sender and continuously receives WAL data.
- **primary_conninfo:** This parameter in the standby's postgre.conf defines how the standby connects to the primary.

Monitoring Streaming Replication:

On the primary, query pg_stat_replication:

```
SELECT application_name, client_addr, state, sync_state,
pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn) AS lag_bytes
FROM pg_stat_replication;
```

This will show connected standbys, their state (e.g., streaming), and lag_bytes (how many bytes behind the primary they are).

2.6. Replication Slots

Replication slots (introduced in Postgre 9.4) are a critical feature for robust physical replication. They automatically prevent the primary from removing WAL segments that are still needed by standbys.

- **Problem without Slots:** Without slots, if a standby falls too far behind or disconnects for a long time, the primary might recycle or remove WAL segments that the standby still needs. This leads to the standby being "out of sync" and requiring a new base backup.
- **Solution with Slots:** A replication slot ensures the primary retains necessary WAL segments until the standby associated with that slot has confirmed receipt.
- **Types:**
    - **Physical Replication Slots:** Used for streaming replication (covered here).
    - **Logical Replication Slots:** Used for logical replication.

Creating a Replication Slot (on Primary):

```
SELECT pg_create_physical_replication_slot('my_standby_slot');
```

Using a Replication Slot (in Standby's primary_conninfo):

Ini, TOML

In standby's postgre.conf

primary_conninfo = 'host=primary_ip port=5432 user=replication_user password=your_password application_name=standby_1 **replication_slot_name=my_standby_slot**'

Monitoring Replication Slots:

On the primary, query pg_replication_slots:

SELECT slot_name, slot_type, active, restart_lsn, confirmed_flush_lsn, wal_status

FROM pg_replication_slots;

- active: Indicates if a standby is currently connected to the slot.
- wal_status: Shows if WAL files are being retained for this slot.
- **Caution:** If a standby using a slot becomes permanently disconnected or crashes, the slot will continue to accumulate WAL files on the primary, potentially filling its disk. Monitor slots and remove unused ones: SELECT pg_drop_replication_slot('my_standby_slot');

2.7. Cascading Replication

Cascading replication allows standbys to receive WAL streams from other standbys, rather than directly from the primary.

- **Architecture:** Primary -> Standby 1 -> Standby 2 -> Standby 3...
- **Benefits:**
  - Reduces load on the primary, as it only needs to serve WAL to the first-tier standbys.
  - Can create more complex replication topologies for geographic distribution or further read scaling.
- **Setup:** A cascading standby's primary_conninfo points to its upstream standby, not the original primary. The upstream standby also needs max_wal_senders configured appropriately.

2.8. Synchronous Replication

Synchronous replication guarantees that a transaction is not considered committed on the primary until its WAL record has been safely received and written to disk by at least

one (or more) designated synchronous standbys. This provides **zero data loss** on failover.

- **Trade-off:** This comes at the cost of increased transaction latency, as the primary must wait for acknowledgment from the standby(s).
- **Configuration (on Primary):**

Ini, TOML

synchronous_commit: Controls when a commit is reported back to the client.

'on' (default): Waits for local flush and remote write + flush.

'remote_write': Waits for local flush and remote write.

'local': Only waits for local flush.

'off': No wait.

For true synchronous replication, use 'on' or 'remote_write'.

synchronous_commit = on

synchronous_standby_names: Defines which standbys are synchronous.

'ANY N (name1, name2, ...)' waits for N standbys from the list.

'FIRST N (name1, name2, ...)' waits for N specific standbys in order.

'name1,name2': Waits for all listed standbys in order.

The 'application_name' in the standby's primary_conninfo must match these names.

synchronous_standby_names = 'ANY 1 (standby_1, standby_2)'

In this example, a transaction will not commit until *any one* of standby_1 or standby_2 has written the WAL to disk (if remote_write) or flushed it (if on).

2.9. Continuous Archiving in Standby

Even with streaming replication, continuous WAL archiving (e.g., to S3 or a separate NFS share) is highly recommended.

- **Disaster Recovery (DR):** It provides a backup for scenarios where both primary and standbys are lost, or for historical PITR beyond the WAL retention of the standby.
- **Network Outage Resiliency:** If streaming replication breaks (e.g., network partition), the standby can fall back to recovering from the WAL archive once the

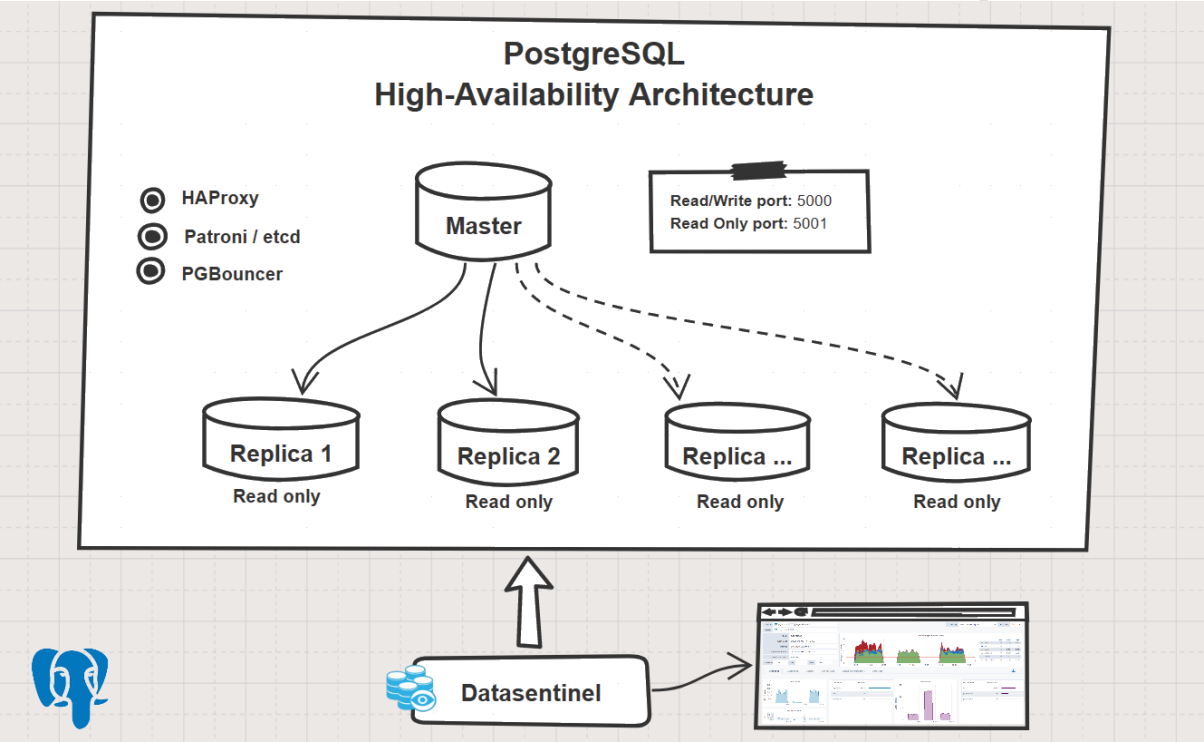
connection is restored, preventing the need for a new base backup if wal_keep_size isn't enough.

- **Standalone PITR:** Allows point-in-time recovery to any point since the last base backup, even without a running standby.

The restore_command configured on the standby allows it to fetch missing WAL files from the archive if they are no longer available directly from the primary's pg_wal (e.g., if wal_keep_size was too small or the primary recycled them).

---

3. Failover



Failover is the process of switching from a failed primary server to a standby server.

- **Detection:** How a primary failure is detected (e.g., monitoring tools, heartbeat).
- **Decision:** Manual, semi-automated (requires administrator confirmation), or fully automated (using clustering tools like Patroni, Repmgr).
- **Promotion:** The chosen standby is promoted to become the new primary.

Manual Promotion (on the chosen standby):

Create a trigger file in the standby's data directory

touch /var/lib/postgre/16/main_standby/trigger_file_promote

OR (preferred in modern Postgre)

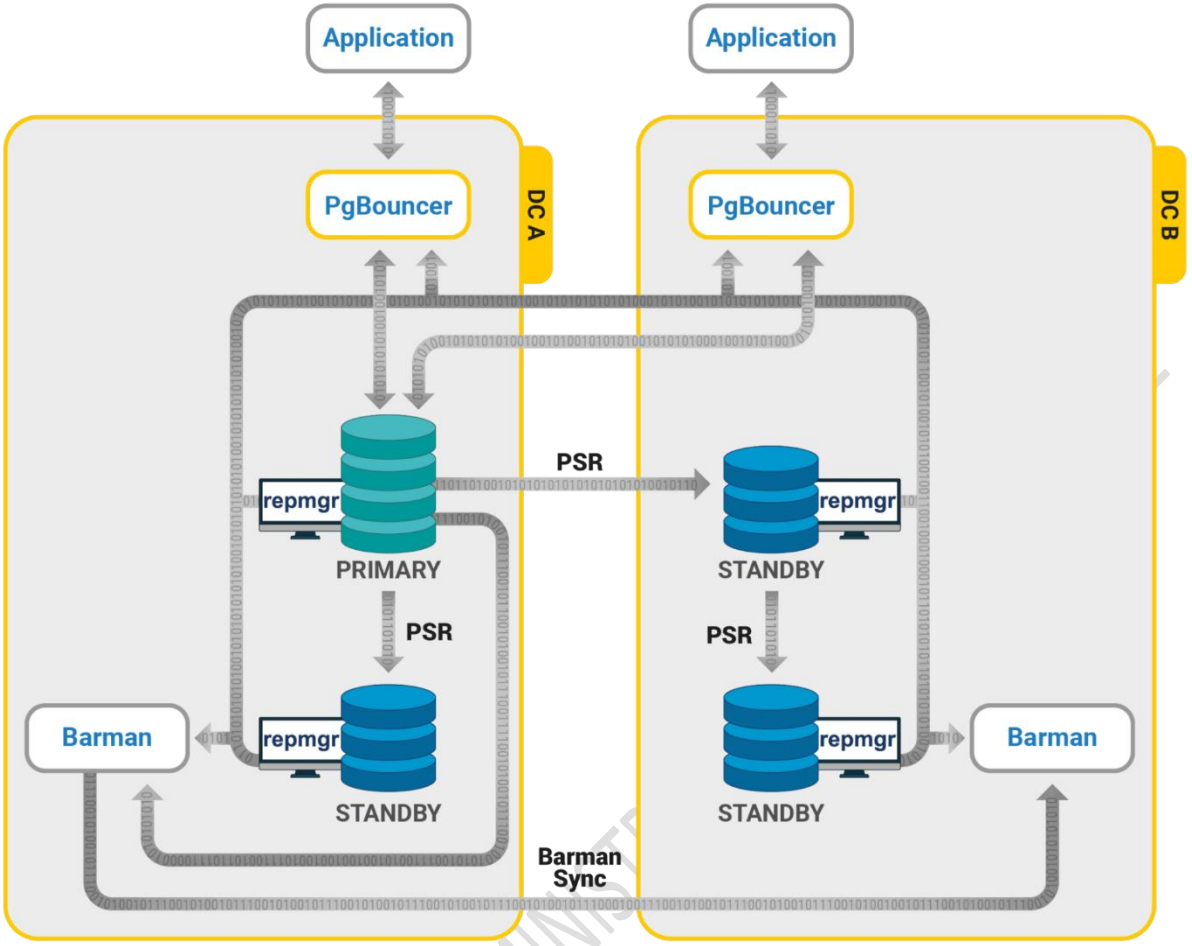
SELECT pg_promote();

    - Upon promotion, the standby completes recovery, removes its standby.signal file, and transitions to normal read/write operation.

- **Redirect Clients:** Applications must be reconfigured or automatically redirected to connect to the new primary. DNS updates, virtual IP addresses, or load balancer reconfigurations are common methods.
- **Old Primary:** The old primary, if it comes back online, must *never* be allowed to start as a primary while the new primary is running. It must either be brought up as a new standby or reinitialized.
- **Split-Brain:** A critical danger in HA is "split-brain," where two servers erroneously believe they are the primary, leading to data divergence. Failover tools employ fencing or quorum mechanisms to prevent this.

---

4. Hot Standby



Hot Standby (introduced in Postgre 9.0) is a specific type of physical streaming replication where the standby server is configured to allow **read-only queries** while it is in continuous recovery mode. This is a powerful feature for:

- **Read Load Balancing:** Distributing read queries across multiple servers to improve performance and scalability.
- **Disaster Recovery:** Providing an instantly available, up-to-date copy of the database for read-only access during a primary outage.

4.1. User's Overview

From a user's perspective, a hot standby behaves like a read-only database. Queries will execute normally. Any attempt to perform a DDL (e.g., CREATE TABLE) or DML (e.g., INSERT, UPDATE, DELETE) operation will result in an error indicating that the database is in recovery mode.

4.2. Handling Query Conflicts

Since the standby is continuously applying changes from the primary, long-running queries on the hot standby can conflict with incoming WAL records.

- **Conflict Scenarios:** If an incoming WAL record attempts to modify or delete data that a running query on the standby is currently using (e.g., reading a tuple that is about to be updated or deleted), a conflict occurs.
- **wal_receiver_timeout**: How long the WAL receiver waits for data from the primary. If no data is received within this time, it assumes the connection is dead.
- **hot_standby_feedback = on:** (Recommended) When enabled on the standby, it sends information back to the primary about currently running queries. The primary will then temporarily retain old row versions that these queries might need, preventing conflicts and query cancellations on the standby. This avoids query cancellations but can cause bloat on the primary if standbys have very long-running queries.
- **max_standby_streaming_delay / max_standby_archive_delay:** These parameters control how long the standby will wait for conflicting queries to finish before canceling them to apply WAL.
    - If a conflicting query exceeds this delay, Postgre will cancel the query on the standby.
    - Set these carefully; too short, and queries are frequently canceled; too long, and replication lag increases significantly.
- **Error Handling:** Applications connecting to a hot standby must be prepared to handle query cancellation errors.

4.3. Administrator's Overview

- **Configuration:** The primary setting is hot_standby = on in the standby's postgre.conf.
- **Resource Usage:** Hot standbys consume resources (CPU, memory, I/O) for both recovery and serving queries. Monitor these resources closely.
- **Read-Only Failover:** In a failover scenario, the hot standby becomes the new primary. Its role shifts from read-only to read/write.

4.4. Hot Standby Parameter Reference

- hot_standby = on (in postgre.conf on standby)

- max_standby_streaming_delay = 30s (on standby, how long to wait for conflicting queries with streaming WAL)
- max_standby_archive_delay = 30s (on standby, how long to wait for conflicting queries with archived WAL)
- hot_standby_feedback = on (on standby, recommended to reduce conflicts)

4.5. Caveats

- **Consistency vs. Freshness:** While hot standbys provide a highly consistent view, there will always be a slight lag behind the primary (replication lag). Applications requiring absolute real-time data should always query the primary.
- **Query Planning:** Query plans might differ between primary and standby due to subtle timing differences in ANALYZE or VACUUM processes.
- **No Writes:** Attempts to write to a hot standby will fail. This is a common pitfall for applications not designed to distinguish between primary and standby.
- **Resource Contention:** Heavy read workloads on the standby can contend with the WAL recovery process for I/O and CPU resources, potentially increasing replication lag.

This detailed explanation covers the core concepts and practical setup for Postgre High Availability, Load Balancing, and Replication using the log-shipping (physical replication) method. Implementing these solutions robustly often involves additional tools and careful operational procedures to manage failover, monitoring, and automation.

🧠 Overview of Postgre High Availability & Replication Solutions

Postgre offers multiple strategies for replication and failover, each suited to different infrastructure setups and business continuity goals.

📦 1. Shared Disk Failover

🔍 Concept

- A single shared disk is accessed by multiple servers.
- Only one server (primary) is active at a time.

- If the primary fails, the standby mounts the disk and starts Postgre.

✅ **Pros**

- No data synchronization overhead.
- Fast failover with no data loss.

❌ **Cons**

- Shared disk is a single point of failure.
- Standby must not access disk while primary is active.

🛠️ **Example Setup**

Assuming a shared NFS mount:

```
# On standby server after primary failure
mount /mnt/shared_pgdata
pg_ctl -D /mnt/shared_pgdata start
```

💾 **2. File System (Block Device) Replication**

🔍 **Concept**

- Uses tools like **DRBD** to mirror disk blocks from primary to standby.
- Ensures consistent file system state.

✅ **Pros**

- No need for Postgre-level replication.
- Transparent to Postgre.

❌ **Cons**

- Requires careful ordering of writes.
- Failover must ensure consistency.

🛠️ **Example (DRBD)**

```
# DRBD configuration snippet
resource pgdata {
 device /dev/drbd0;
 disk /dev/sda1;
```

```
 meta-disk internal;
 on primary {
  address 192.168.1.1:7788;
 }
 on standby {
  address 192.168.1.2:7788;
 }
}
```

📜 3. Write-Ahead Log (WAL) Shipping

🔍 Concept

- WAL records are shipped to standby servers.
- Can be **file-based** or **streaming replication**.

✅ Pros

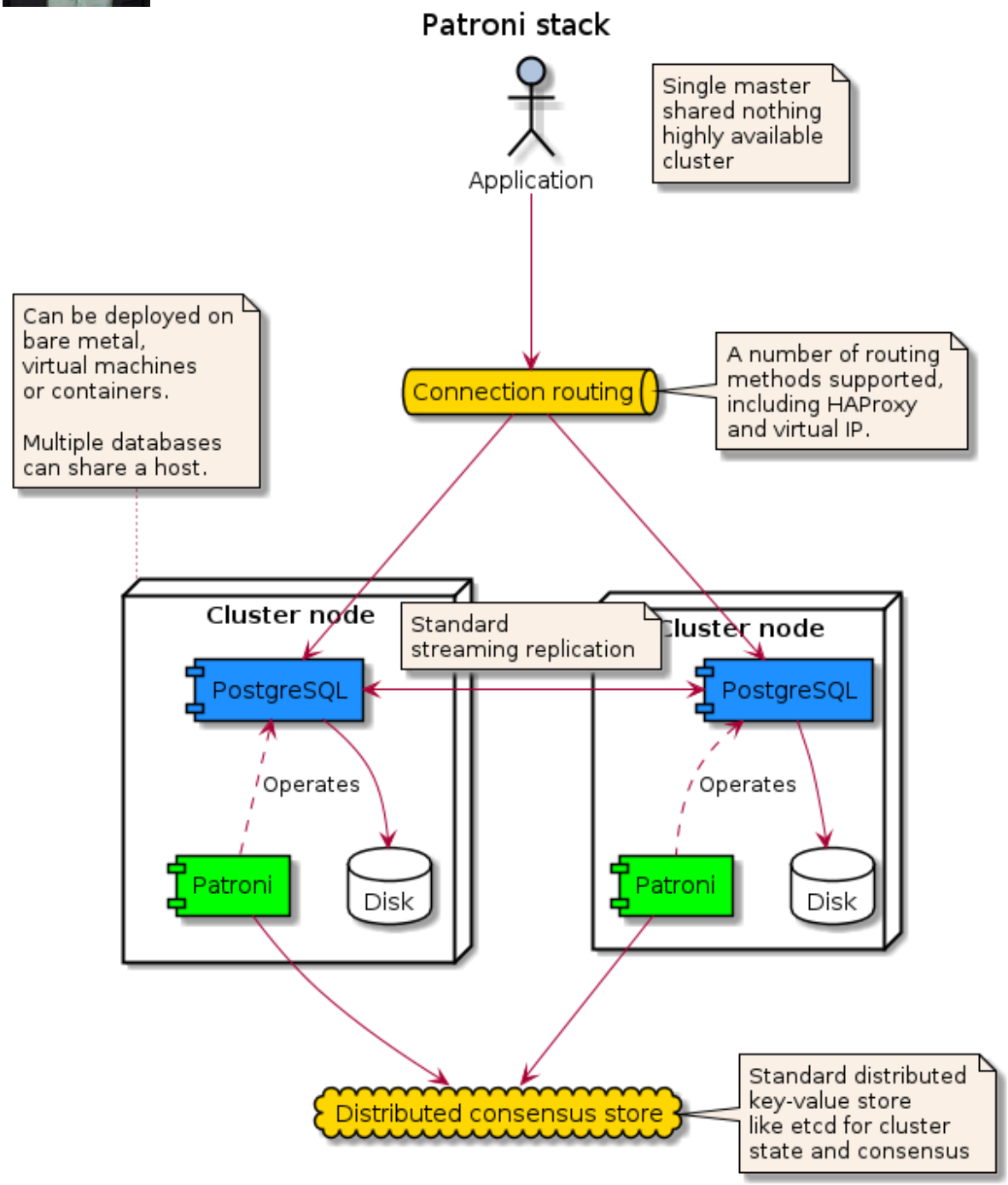
- Built-in Postgre feature.
- Supports hot standby (read-only queries on standby).

❌ Cons

- Entire cluster must be replicated.
- Asynchronous mode risks data loss.

MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia



🛠️ **Setup Example**

File-Based WAL Archiving

postgre.conf

archive_mode = on

archive_command = 'cp %p /var/lib/postgre/wal_archive/%f'

Streaming Replication

On standby

```
pg_basebackup -h primary_host -D /var/lib/postgre/data -U replicator -P --wal-method=stream
```

🔄 4. Logical Replication

🔍 Concept

- Replicates changes at the table level using logical decoding.
- Allows selective replication and bidirectional sync.

✅ Pros

- Per-table granularity.
- Supports multi-directional replication.

❌ Cons

- Requires Postgre 10+.
- Doesn't replicate DDL changes.

🛠️ Setup Example

-- On publisher

```
CREATE PUBLICATION my_pub FOR TABLE users;
```

-- On subscriber

```
CREATE SUBSCRIPTION my_sub
 CONNECTION 'host=publisher dbname=mydb user=replicator password=secret'
 PUBLICATION my_pub;
```

🧲 5. Trigger-Based Replication

🔍 Concept

- Uses triggers to capture changes and replicate them.
- Often asynchronous and per-table.

✅ Pros

- Fine-grained control.
- Good for analytical workloads.

❌ **Cons**

- Complex setup.
- Possible data loss during failover.

🛠️ **Example Tool: Slony-I**

```
# Slony-I configuration involves defining sets and subscriptions
slonik <<EOF
 cluster name = my_cluster;
 node 1 admin conninfo = 'host=primary dbname=mydb user=slony';
 node 2 admin conninfo = 'host=standby dbname=mydb user=slony';
 create set (id = 1, origin = 1, comment = 'User table');
 add table (set id = 1, id = 1, fully qualified name = 'public.users');
 subscribe set (id = 1, provider = 1, receiver = 2);
EOF
```

🧱 **6. -Based Replication Middleware**

🔍 **Concept**

- Middleware intercepts queries and distributes them.
- Each server executes queries independently.

✅ **Pros**

- Load balancing for read queries.
- No Postgre changes needed.

❌ **Cons**

- Risk of inconsistent data.
- Requires two-phase commit for consistency.

🛠️ **Example Tool: Pgpool-II**

```
# pgpool.conf
load_balance_mode = on
replication_mode = on
```

```
master_slave_mode = off
```

🌐 7. Asynchronous Multimaster Replication

🔍 Concept

- Each server operates independently.
- Periodic sync resolves conflicts.

✅ Pros

- Works over unreliable networks.
- Good for remote or mobile nodes.

❌ Cons

- Conflict resolution needed.
- Data may be stale.

🛠️ Example Tool: Bucardo

```
bucardo add db postgres1 dbname=mydb host=host1

bucardo add db postgres2 dbname=mydb host=host2

bucardo add sync my_sync tables=users dbs=postgres1:source,postgres2:target
```

🔁 8. Synchronous Multimaster Replication

🔍 Concept

- All servers accept writes.
- Data is synced before commit.

✅ Pros

- True high availability.
- No primary/standby distinction.

❌ Cons

- High latency.
- Complex coordination.

🛠️ Postgre Two-Phase Commit

```
-- On all servers
```


```
BEGIN;

INSERT INTO users VALUES (1, 'Mukesh');

PREPARE TRANSACTION 'tx1';

-- After coordination

COMMIT PREPARED 'tx1';
```

📊 Feature Comparison Matrix

Feature	Shared Disk	FS Repl.	WAL Shipping	Logical Repl.	Trigger Repl.	Middleware	Async MM	Sync MM
Popular Examples	NAS	DRBD	Built-in	pglogical	Slony	Pgpool-II	Bucardo	—
Communication Method	Disk	Blocks	WAL	Logical Decoding	Table Rows		Rows	Rows
No Special Hardware		✅	✅	✅	✅	✅	✅	✅
Multiple Primary Servers				✅		✅	✅	✅
No Overhead on Primary	✅		✅	✅		✅		
Fast Failover	✅	✅	✅ (sync)	✅ (sync)		✅		
Read Queries on Replicas			✅ (hot standby)	✅	✅	✅	✅	✅
Per-Table Granularity				✅	✅		✅	✅
No Conflict Resolution Needed	✅	✅	✅		✅	✅		

🧩 Other Solutions

🔷 **Data Partitioning**

- Split tables across servers by region or department.
- Each server owns a subset of data.

🔷 **Parallel Query Execution**

- Distribute parts of a query across servers.
- Combine results centrally.

🛠️ **Tool: PL/Proxy**

```
-- Example function using PL/Proxy
CREATE FUNCTION get_user(integer) RETURNS user AS $$
 SELECT * FROM users WHERE id = $1;
$$ LANGUAGE plproxy;
```

🧭 **Overview: Log-Shipping Standby Servers**

Log-shipping is a method of replication where **Write-Ahead Logs (WAL)** from the primary server are continuously shipped to one or more standby servers. These standby servers replay the WALs to stay in sync and can take over in case of failure.

📌 **2.1 Planning**

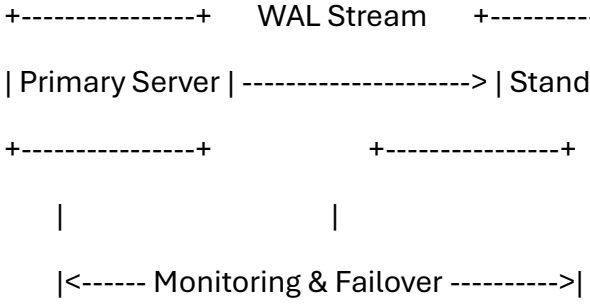
🎯 **Purpose**

- Design a robust replication and failover strategy.
- Define Recovery Point Objective (RPO) and Recovery Time Objective (RTO).

🧠 **Key Considerations**

- Number of standby servers.
- Network bandwidth and latency.
- Storage for WAL archives.
- Failover automation (e.g., using tools like **repmgr**, **Patroni**, or **Pacemaker**).

📊 **Diagram**



🖥️ 2.2 Standby Server Operation

🔄 Purpose

- Keep standby servers in sync by replaying WAL files.
- Standby can be in **warm** (not accepting queries) or **hot** (read-only queries allowed) mode.

🛠️ Example

```
# Standby automatically replays WALs
pg_ctl start -D /var/lib/postgre/data
```

✅ Advantages

- High availability.
- Read scalability with hot standby.

❌ Disadvantages

- Lag in asynchronous mode.
- Requires careful WAL management.

🛠️ 2.3 Preparing the Primary for Standby Servers

🔧 Steps

1. Enable WAL archiving:

```
archive_mode = on
archive_command = 'cp %p /var/lib/postgre/wal_archive/%f'
```

2. Set wal_level to replica or logical:

```
wal_level = replica
```

3. Configure max WAL senders:

max_wal_senders = 5

🏗️ 2.4 Setting Up a Standby Server

🔧 Steps

1. Take a base backup:

```
pg_basebackup -h primary_host -D /var/lib/postgre/data -U replicator -P --wal-method=stream
```

2. Create standby.signal file:

```
touch /var/lib/postgre/data/standby.signal
```

3. Configure recovery:

```
primary_conninfo = 'host=primary_host user=replicator password=secret'

restore_command = 'cp /var/lib/postgre/wal_archive/%f %p'
```

🌊 2.5 Streaming Replication

🎯 Purpose

- Real-time WAL streaming from primary to standby.

🛠️ Setup

On primary:

```
wal_level = replica

max_wal_senders = 5
```

On standby:

```
primary_conninfo = 'host=primary_host user=replicator password=secret'
```

✅ Advantages

- Low latency.
- Efficient bandwidth usage.

❌ Disadvantages

- Requires stable network.
- Standby may lag if network is slow.

🎛️ 2.6 Replication Slots

🎯 Purpose

- Prevent WAL files from being deleted before standby consumes them.

🛠️ Create Slot

```
SELECT * FROM pg_create_physical_replication_slot('standby_slot');
```

Configure on standby:

```
primary_conninfo = 'host=primary_host user=replicator password=secret'
primary_slot_name = 'standby_slot'
```

✅ Advantages

- Ensures WAL retention.
- Prevents data loss.

❌ Disadvantages

- Can cause disk bloat if standby is down.

🌐 2.7 Cascading Replication

🎯 Purpose

- A standby server acts as a WAL source for other standbys.

🛠️ Setup

On intermediate standby:

```
primary_conninfo = 'host=primary_host user=replicator'
hot_standby_feedback = on
```

On downstream standby:

```
primary_conninfo = 'host=intermediate_standby user=replicator'
```

✅ Advantages

- Reduces load on primary.
- Scales replication tree.

❌ Disadvantages

- Adds complexity.
- Failure in intermediate affects downstream.

🔁 2.8 Synchronous Replication

🎯 Purpose

- Ensures no transaction is committed until at least one standby confirms receipt.

🛠️ Setup

On primary:

```
synchronous_standby_names = 'standby1'
```

On standby:

```
primary_conninfo = 'host=primary_host user=replicator'
```

✅ Advantages

- Zero data loss.
- Strong consistency.

❌ Disadvantages

- Slower commits.
- Risk of blocking if standby is slow.

🔄 2.9 Continuous Archiving in Standby

🎯 Purpose

- Archive WALs on standby for PITR or cascading replication.

🛠️ Setup

```
archive_mode = on
archive_command = 'cp %p /var/lib/postgre/standby_archive/%f'
```

✅ Advantages

- Redundant WAL storage.
- Enables PITR from standby.

❌ Disadvantages

- Extra storage needed.

- Complexity in archive management.

🧪 Real-Life Project Scenario

🏢 Scenario: E-commerce Platform

Problem

- Primary database crashed during peak sale.
- Standby was lagging due to network issues.
- WAL files were deleted before standby caught up.

Solution

- Implement **replication slots** to retain WALs.
- Switch to **synchronous replication** for critical tables.
- Use **cascading replication** to distribute load.

Outcome

- Zero data loss.
- Faster failover.
- Improved scalability.

🛠️ Step-by-Step Failover Automation Script (Using repmgr)

repmgr is a popular tool for managing Postgre replication and automating failover.

🔧 Prerequisites

- Postgre installed on primary and standby nodes.
- repmgr installed on all nodes.
- Passwordless SSH between nodes.
- Replication user created in Postgre.

🧩 Setup Overview

1. Install repmgr

sudo apt install repmgr

2. Create Replication User

CREATE USER replicator WITH REPLICATION ENCRYPTED PASSWORD 'yourpassword';

3. Configure postgre.conf

wal_level = replica

max_wal_senders = 10

hot_standby = on

4. Configure pg_hba.conf

host replication replicator 0.0.0.0/0 md5

5. Initialize repmgr on Primary

repmgr -f /etc/repmgr.conf primary register

6. Initialize repmgr on Standby

repmgr -h primary_host -U replicator -d replication_db -f /etc/repmgr.conf standby clone

repmgr -f /etc/repmgr.conf standby register

7. Enable repmgrd Daemon

sudo systemctl enable repmgrd

sudo systemctl start repmgrd

8. Automated Failover

When the primary fails, repmgrd promotes the standby:

repmgr standby promote

🧪 **Test Failover**

sudo systemctl stop postgre # Simulate primary failure

📊 **Visual Dashboard Design for Monitoring Replication Health**

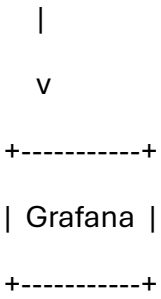
You can use **Grafana + Prometheus + Postgre Exporter** to build a real-time dashboard.

📦 **Architecture**

+----------------+   +------------------+   +-------------+

| Postgre DBs | --> | Postgre Exporter | --> | Prometheus |

+----------------+   +------------------+   +-------------+



📦 Metrics to Monitor

Metric	Description
pg_replication_lag	WAL lag between primary and standby
pg_stat_replication	Connection status of standbys
pg_last_xact_replay_timestamp	Time of last replayed transaction
pg_is_in_recovery()	Indicates if node is standby

🎨 Grafana Panels

🔷 Panel 1: Replication Lag

- **Type**: Line graph
- **Query**: pg_replication_lag{instance="standby1"}

🔷 Panel 2: Standby Status

- **Type**: Table
- **Query**: pg_stat_replication

🔷 Panel 3: WAL Replay Time

- **Type**: Gauge
- **Query**: pg_last_xact_replay_timestamp

🔷 Panel 4: Node Role

- **Type**: Single Stat
- **Query**: pg_is_in_recovery()

🧪 Real-World Scenario: Retail Analytics Platform

MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

🧨 Problem

- Primary node failed during Black Friday.
- No alert triggered.
- Manual failover took 30 minutes.

✅ Solution

- Implemented repmgr for automatic failover.
- Deployed Grafana dashboard with alerting rules:
  - Alert if replication lag > 10 seconds.
  - Alert if standby disconnects.

🚀 Result

- Failover in under 10 seconds.
- Zero downtime.
- Real-time visibility into replication health.

Adding **alerting features** to your Postgre replication dashboard (especially if you're using **Grafana**) transforms it from a passive monitor into an active guardian. Here's how you can set it up step-by-step:

🚨 Step-by-Step: Adding Alerts in Grafana

🧰 Prerequisites

- Grafana connected to Prometheus (or another data source).
- Postgre Exporter running and collecting metrics.
- Dashboard panels already created.

🔔 1. Choose the Panel for Alerting

Alerts are tied to **time series panels** (like graphs or single stats). For example:

- Replication lag
- WAL replay timestamp

- Node role (primary/standby)

⚙️ 2. Open Panel → Alert Tab

1. Click the panel title → **Edit**.
2. Go to the **Alert** tab.
3. Click **Create Alert Rule**.

📐 3. Define Alert Conditions

Set up a condition like:

WHEN avg() OF query(A, 5m, now) IS ABOVE 10

This means: if the average replication lag over 5 minutes exceeds 10 seconds, trigger an alert.

You can also use:

- pg_last_xact_replay_timestamp to detect stale standby
- pg_stat_replication to check if standby is disconnected

📤 4. Set Notification Channels

Grafana supports:

- ✉️ Email
- 📱 Slack
- 🐦 Telegram
- 🧭 Microsoft Teams
- 🧠 Webhooks (for custom integrations)

To configure:

1. Go to **Alerting → Notification Channels**.
2. Add your preferred channel.
3. Link it to your alert rule.

🧪 5. Test Your Alert

Simulate a failure:

sudo systemctl stop postgre # on primary

Watch the replication lag spike and confirm the alert fires.

🎯 Example: Alert for Replication Lag

Condition: avg() OF query(A, 5m, now) > 10

Evaluate every: 1m

For: 2m

Notification: Slack Channel "DB Alerts"

Message: "Replication lag exceeded 10s on standby1"

🧠 Pro Tips

- Use **"For"** duration to avoid false positives from short spikes.
- Combine alerts with **annotations** to mark events on graphs.
- Use **dashboard variables** to make alerts dynamic across environments.

🛠️ Real-Life Use Case: Financial Services App

Problem

- Replication lag silently grew during peak hours.
- Standby became stale, risking data loss on failover.

Solution

- Alert triggered when lag > 5s for 2 minutes.
- Slack notification sent to DB team.
- Auto-failover initiated via repmgr.

Outcome

- Zero downtime.
- Proactive response before users noticed

This will allow **Grafana alerts** to trigger **automated scripts** on your server when certain conditions are met (like replication lag or standby disconnect). Here's a full setup using **Python + Flask**, with a shell script trigger.

🧰 What You'll Need

- Python 3 installed
- Flask (pip install flask)

- A shell script you want to run (e.g., failover, notification, cleanup)
- Grafana alert configured to send a webhook

🧠 **Architecture**

Grafana Alert → Webhook → Flask Server → Trigger Script

📝 **Step 1: Create Your Shell Script**

Let's say you want to promote a standby server:

```
# promote_standby.sh
#!/bin/bash
echo "Promoting standby server..."
/usr/bin/repmgr standby promote
```

Make it executable:

```
chmod +x promote_standby.sh
```

🐍 **Step 2: Build the Webhook Handler in Python**

```
# webhook_handler.py
from flask import Flask, request
import subprocess

app = Flask(__name__)

@app.route('/grafana-alert', methods=['POST'])
def handle_alert():
    data = request.json
    alert_name = data.get('title', 'Unknown Alert')
    state = data.get('state', 'unknown')

    print(f"Received alert: {alert_name} - State: {state}")
```

```
  if state == "alerting":
    try:
      subprocess.run(["/path/to/promote_standby.sh"], check=True)
      return "Script executed successfully", 200
    except subprocess.CalledProcessError as e:
      return f"Script failed: {e}", 500
  return "No action taken", 200


if __name__ == '__main__':
  app.run(host='0.0.0.0', port=5000)
```

🚀 **Step 3: Run the Webhook Server**

python3 webhook_handler.py

Make sure port 5000 is open and reachable from Grafana.

🔔 **Step 4: Configure Grafana Webhook**

In Grafana:

1. Go to **Alerting → Notification Channels**
2. Add a **Webhook**
3. Set the URL:
4. http://your-server-ip:5000/grafana-alert
5. Test it with a sample alert.

🧪 **Example Payload from Grafana**

Grafana sends JSON like this:

```
json
{
 "title": "Replication Lag Alert",
 "state": "alerting",
 "message": "Lag exceeded threshold",
```

```
 "evalMatches": [...]
}
```

You can customize the handler to parse evalMatches and trigger different scripts based on alert type.

🧠 Bonus: Add Logging

For production, add logging to a file:

python

```
import logging
logging.basicConfig(filename='webhook.log', level=logging.INFO)
```

🧨 Real-World Use Case

Scenario

- Alert: Replication lag > 10s
- Webhook triggers failover script
- Standby promoted automatically
- Slack notification sent via another script

Outcome

- <10s recovery time
- No manual intervention
- Peace of mind during peak hours

🧭 What Is Postgre Failover?

Failover is the process of automatically or manually switching from a **primary (master)** Postgre server to a **standby (replica)** when the primary becomes unavailable. This ensures **minimal downtime**, **data integrity**, and **business continuity**.

⚙️ Failover Scenarios

Scenario	Action Required
✅ Primary fails	Promote standby to primary

Scenario	Action Required
✅ Standby fails	No failover; restart or rebuild standby
⚠️ Old primary restarts after failover	Must prevent it from acting as primary (STONITH)
✅ Planned switchover	Promote standby for maintenance/testing

🔁 Failover Workflow

🔷 1. Detection

- Use heartbeat tools (e.g., Pacemaker, Keepalived, repmgrd) to detect primary failure.

🔷 2. Promotion

- Promote standby using:

```
pg_ctl promote -D /var/lib/postgre/data
```

or

```
SELECT pg_promote();
```

🔷 3. STONITH (Shoot The Other Node In The Head)

- Prevent old primary from rejoining as primary.
- Use fencing tools or disable Postgre on old primary.

🔷 4. Rebuild Standby

- Use pg_rewind to sync old primary as new standby:

```
pg_rewind --target-pgdata=/var/lib/postgre/data --source-server="host=new_primary user=replicator"
```

🧰 Tools for Failover Automation

Tool	Purpose	Notes
repmgr	Failover, monitoring, promotion	Lightweight, widely used
Patroni	HA with etcd/consul coordination	Advanced, cloud-native
Pacemaker	Cluster management & fencing	Enterprise-grade

Tool	Purpose	Notes
pg_auto_failover	Built-in failover logic	Easy setup, good for small clusters

🧪 **Example: repmgr Failover Setup**

Primary:

repmgr -f /etc/repmgr.conf primary register

Standby:

repmgr -f /etc/repmgr.conf standby clone

repmgr -f /etc/repmgr.conf standby register

Enable Daemon:

sudo systemctl enable repmgrd

sudo systemctl start repmgrd

Simulate Failover:

sudo systemctl stop postgre # on primary

Standby will auto-promote.

🧠 **Logical Replication Slot Failover**

If using **logical replication**, ensure slots are synced before failover:

- Check slot status:

SELECT * FROM pg_replication_slots;

- Use tools like pglogical or bucardo for slot management.

📉 **Degenerate State & Recovery**

After failover:

- Only one server is active (new primary).
- Rebuild standby using:
    - pg_basebackup
    - pg_rewind
    - Or clone from new primary

🧪 Real-World Scenario: Banking App

Problem

- Primary crashed during transaction peak.
- Old primary restarted and began accepting writes.
- Data divergence occurred.

Solution

- Implemented STONITH using fencing agent.
- Switched to Patroni with etcd for coordination.
- Added witness node to prevent split-brain.

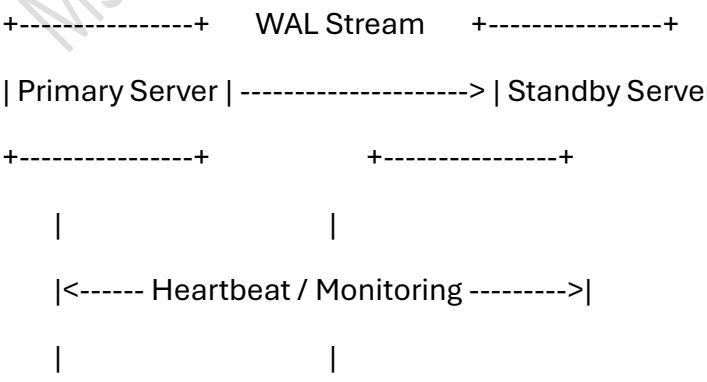
Result

- Zero data loss.
- <10s failover time.
- Fully automated recovery.

✅ Best Practices

- 🔄 **Test failover regularly** (scheduled switchovers).
- 📜 **Document procedures** for manual intervention.
- 🔐 **Secure fencing mechanisms** to avoid split-brain.
- 📊 **Monitor replication lag** and health.
- 🧪 **Use pg_rewind** for fast standby rebuilds.
- 🧠 **Consider witness nodes** for quorum-based decisions.

📊 Visual Diagram



v v

Failover Triggered Standby Promoted

| |

v v

STONITH Old Primary Rebuild New Standby

🛠️ 1. Custom Failover Script (Shell-Based)

This script promotes the standby server and updates DNS or virtual IP to redirect traffic.

📄 **failover.sh**

```
#!/bin/bash

# Define variables
PGDATA="/var/lib/postgre/data"
VIP="192.168.1.100"
INTERFACE="eth0"
LOGFILE="/var/log/pg_failover.log"

echo "$(date): Failover initiated" >> $LOGFILE

# Promote standby
pg_ctl promote -D $PGDATA
echo "$(date): Standby promoted to primary" >> $LOGFILE

# Assign virtual IP
ip addr add $VIP/24 dev $INTERFACE
echo "$(date): Virtual IP assigned to new primary" >> $LOGFILE
```

```
# Optional: Notify via Slack or email
curl -X POST -H 'Content-type: application/json' \
--data '{"text":"Postgre failover: standby promoted"}' \
https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK

exit 0
```

Make it executable:

```
chmod +x failover.sh
```

You can trigger this via a webhook, cron job, or monitoring tool like repmgrd.

🧨 2. STONITH Fencing Strategy

STONITH (Shoot The Other Node In The Head) prevents the old primary from rejoining after failover.

🔐 Strategy Options

Method	Description	Tool
Power fencing	Use IPMI or SSH to power off old primary	fence_ipmilan, fence_ssh
Service fencing	Disable Postgre service on old primary	systemctl stop postgre
Network fencing	Remove old primary from load balancer	HAProxy, Keepalived

🧪 Example: SSH Fencing Script

```
#!/bin/bash
ssh postgres@old-primary 'sudo systemctl stop postgre'
echo "STONITH: Postgre stopped on old primary"
```

You can integrate this into your failover script or use cluster tools like **Pacemaker** to automate fencing.

📊 3. Grafana Alert Integration

Let's create an alert that triggers when replication lag exceeds a threshold and sends a webhook to your failover handler.

🔔 Alert Rule in Grafana

Condition:

WHEN avg() OF query(A, 5m, now) IS ABOVE 10

Notification Channel:

- Type: Webhook
- URL: http://your-server:5000/grafana-alert

🐍 **Webhook Handler (Python + Flask)**

python

```
from flask import Flask, request
import subprocess

app = Flask(__name__)

@app.route('/grafana-alert', methods=['POST'])
def handle_alert():
  data = request.json
  if data.get('state') == 'alerting':
    subprocess.run(["/path/to/failover.sh"])
    return "Failover triggered", 200
  return "No action taken", 200

app.run(host='0.0.0.0', port=5000)
```

🧠 **Real-World Scenario: Sydney-Based SaaS Platform**

Problem

- Primary DB failed during client onboarding.
- Standby promoted, but old primary restarted and caused split-brain.
- No alerting system in place.

Solution

- Deployed failover script with fencing.
- Integrated Grafana alerts with webhook triggers.
- Added Slack notifications for visibility.

Outcome

- <15s recovery time.
- No data loss.
- Fully automated failover with human oversight.

🔥 Postgre Hot Standby – Technical Deep Dive

📜 1. What is Hot Standby?

Hot Standby is a Postgre feature allowing **read-only query access** on a **standby server** while it is applying **WAL (Write-Ahead Logging)** records from the primary server. This enables:

- **High availability** during primary downtime
- **Query offloading** to reduce primary load
- **Recovery inspection** with up-to-date but read-only access

It works by **replaying WAL** logs on a standby while still **accepting SELECT and other non-DML statements**.

🧑‍💻 2. User's Overview

✅ Allowed Commands in Hot Standby:

Read-only transactions support:

-- Data access

SELECT * FROM my_table;

COPY my_table TO '/tmp/data.csv';

-- Cursor control

```
DECLARE mycursor CURSOR FOR SELECT * FROM my_table;

FETCH NEXT FROM mycursor;

CLOSE mycursor;


-- Transaction control

BEGIN;

SAVEPOINT sp1;

ROLLBACK TO SAVEPOINT sp1;

END;


-- Locking (non-blocking)

LOCK my_table IN ACCESS SHARE MODE;


-- Planning

PREPARE stmt AS SELECT * FROM my_table WHERE id = $1;

EXECUTE stmt(101);

DEALLOCATE stmt;


-- System settings

SHOW transaction_read_only;

SET statement_timeout = '1min';

RESET statement_timeout;
```

❌ **Disallowed Operations:**

These will raise errors in standby mode:

```
-- DML

INSERT INTO my_table VALUES (1);      -- ❌

UPDATE my_table SET col = 'X';        -- ❌
```

DELETE FROM my_table WHERE id = 1; -- ❌

-- DDL

CREATE TABLE test (id INT); -- ❌

ALTER TABLE test ADD COLUMN name TEXT; -- ❌

DROP TABLE test; -- ❌

-- Lock escalation

LOCK my_table IN ACCESS EXCLUSIVE MODE; -- ❌

-- Sequences

SELECT nextval('seq_name'); -- ❌

-- LISTEN/NOTIFY

LISTEN channel; -- ❌

⚔️ 3. Handling Query Conflicts

⚠️ Conflict Scenarios:

WAL Action on Primary Conflict on Standby

DROP TABLESPACE	Temp file use
VACUUM (cleanup)	Active query holds snapshot of affected rows
DROP DATABASE	Active sessions
DROP TABLE	Querying same table

Postgre **must cancel** standby queries that block WAL replay.

🔧 Tuning Parameters:

postgre.conf (standby)

```
hot_standby = on
max_standby_archive_delay = 300s       # wait 5 mins before canceling
max_standby_streaming_delay = 300s
hot_standby_feedback = on              # send info back to primary to prevent VACUUM cleanup
```

🧠 **Monitoring Query Conflicts:**

```
-- Standby system view
SELECT * FROM pg_stat_database_conflicts;
```

🛠️ **4. Administrator's Overview**

🏳️ **Enabling Hot Standby:**

1. On **Primary Server**:

```
wal_level = replica
archive_mode = on
archive_command = 'cp %p /var/lib/postgre/wal_archive/%f'
max_wal_senders = 10
hot_standby = on
```

2. On **Standby Server**:
    - Use base backup and restore:

```
pg_basebackup -h primary_host -D /var/lib/postgre/15/main -U repl_user -Fp -Xs -P
```

    - Create standby.signal in the data directory.
    - Configure recovery options:

```
# postgre.conf
hot_standby = on

# standby.signal present in PGDATA
primary_conninfo = 'host=primary_host port=5432 user=repl_user password=secret'
```

3. Start standby:

```
pg_ctl start -D /var/lib/postgre/15/main
```

4. Confirm log messages:

LOG: consistent recovery state reached

LOG: database system is ready to accept read-only connections

---

🧰 5. Key Parameters

Parameter	Description	Applies To
hot_standby	Enables read-only queries during recovery	Standby
wal_level	Must be replica or logical for standby to work	Primary
max_standby_archive_delay	Delay before canceling query during WAL archive replay	Standby
max_standby_streaming_delay	Delay before canceling query during streaming	Standby
hot_standby_feedback	Prevents VACUUM cleanup that can break standby queries	Standby
primary_conninfo	Connection to primary server for WAL streaming	Standby
standby.signal	File that activates standby mode	Standby

📋 6. Caveats & Considerations

- Cannot create **temporary tables** in standby.
- DDLs (CREATE INDEX, VACUUM, ANALYZE) are **not permitted**.
- WAL changes like DROP DATABASE will forcibly **disconnect** sessions on standby.
- LISTEN/NOTIFY, nextval(), etc. are **restricted**.
- **Autovacuum** is disabled on standby.

- Statistics like pg_stat_all_tables.n_tup_ins are not updated by WAL replay.
- **No Serializable isolation** in standby.

🔎 7. Helpful Diagnostic Queries

```
-- Check if session is in hot standby mode
SHOW in_hot_standby;

-- Or older versions
SHOW transaction_read_only;

-- Check lock status
SELECT * FROM pg_locks;

-- Check WAL replay delay
SELECT now() - pg_last_xact_replay_timestamp() AS standby_lag;

-- Cancel query manually
SELECT pg_cancel_backend(pid)
FROM pg_stat_activity
WHERE state = 'active'
 AND backend_type = 'client backend';
```

📄 8. Example Scripts

🔄 Setup replication user (on primary):

```
CREATE ROLE repl_user WITH REPLICATION LOGIN PASSWORD 'secret';
```

📤 Archive script example:

archive_command = 'rsync -a %p standby:/var/lib/postgre/wal_archive/%f'

📂 **Base backup:**

pg_basebackup -h primary -D /data/pg_standby -U repl_user -Fp -Xs -P -R

Automatically sets up standby.signal and primary_conninfo.

✅ **Best Practices**

- Keep config parameters (max_connections, max_prepared_transactions, etc.) **equal or higher** on standby.
- Monitor and tune max_standby_*_delay based on workload.
- Set hot_standby_feedback = on **cautiously** to avoid **bloat** on primary.
- Use pg_stat_database_conflicts to monitor cancellations.
- Don't rely on triggers or sequences during hot standby.

---

1 Standard Unix Tools

Overview: Tools like ps, top, iostat, vmstat offer OS-level visibility into CPU, memory, and I/O usage.
Use Case: Diagnose system-level bottlenecks during heavy DB activity.
CLI integration:

Monitor Postgre processes

ps aux | grep postgres

I/O and CPU usage overview

iostat -xd 2

vmstat 5

---

2 The Cumulative Statistics System

Purpose: Central to monitoring: Postgre maintains shared-memory statistics automatically and persists them across clean restarts. Useful for long-term performance analysis. DataSunrise+6mstips.com+6Postgre+6Postgre

2.1 Statistics Collection Configuration

Files: postgre.conf controls collection—e.g., track_counts, track_io_timing, stats_temp_directory.

track_counts = on

track_io_timing = on

2.2 Viewing Statistics

Views: Many pg_stat_* views reflecting cumulative stats or session-level activity (see next subsections). Postgre

---

2.3 pg_stat_activity

Purpose: One row per connection showing user, state, current query, wait event, etc. Excellent for identifying long queries and blocked sessions. PostgreStack Overflow
Example:

```
SELECT pid, usename, datname, client_addr, state,
    now() - query_start AS duration,
    wait_event_type, wait_event,
    substring(query,1,100) AS snippet
FROM pg_stat_activity
WHERE state <> 'idle'
 AND now() - query_start > interval '5 minutes';
```

Terminate idle sessions older than 30 min:

```
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE state in ('idle in transaction')
 AND state_change < current_timestamp - INTERVAL '30 min'
 AND pid <> pg_backend_pid();
```

``` :contentReference[oaicite:4]{index=4}

---
```

2.4 pg_stat_replication

Purpose: Shows one row per WAL sender on primary server, including `write_lsn`, `flush_lsn`, `replay_lsn`, and lag times. Useful to monitor replication state. :contentReference[oaicite:5]{index=5}

```
SELECT pid, application_name, client_addr,
    state, write_lsn, flush_lsn, replay_lsn,
    write_lag, flush_lag, replay_lag
FROM pg_stat_replication;
```

2.5 pg_stat_replication_slots

Purpose: One row per replication slot; includes active, restart_lsn, confirmed_flush_lsn. You can track WAL retention backlog. designandexecute.com
Script:

```
SELECT slot_name, plugin, slot_type, active,
    restart_lsn, confirmed_flush_lsn
FROM pg_stat_replication_slots;
```

2.6 pg_stat_wal_receiver

Purpose: On standby server: shows WAL receiver status and LSNs. designandexecute.com
Script:

```
SELECT pid, status, received_lsn, last_msg_send_time,
    last_msg_receipt_time, latest_end_lsn
FROM pg_stat_wal_receiver;
```

2.7 pg_stat_recovery_prefetch

Purpose: Shows block prefetch statistics during recovery on standby. Helps tune recovery performance.

Script:

SELECT * FROM pg_stat_recovery_prefetch;

2.8-2.2.12 Other Views

- **pg_stat_subscription**, **pg_stat_subscription_stats**: logical replication subscribers details.
- **pg_stat_ssl**, **pg_stat_gssapi**: security connection metadata.
- **pg_stat_archiver**, **pg_stat_bgwriter**, **pg_stat_checkpointer**, **pg_stat_io**, **pg_stat_slru**: internal process stats (archiver, background writer, I/O, SLRU usage). wiki.postgre.org+15Postgre+15tutorialdba.com+15

Script examples:

SELECT * FROM pg_stat_bgwriter;

SELECT * FROM pg_stat_archiver;

SELECT * FROM pg_stat_database_conflicts;

2.17-2.2.22 Table/Index Statistics

- **pg_stat_database**: summary stats per DB—transactions, tuples, temp files, etc.
- **pg_stat_database_conflicts**: cancellations due to standby conflicts.
- **pg_stat_all_tables**, **pg_stat_all_indexes**, **pg_statio_all_tables**, **pg_statio_all_indexes**, **pg_statio_all_sequences**: detailed table/index usage, I/O stats.
- **pg_stat_user_functions**: execution counts and timing for user-defined functions.
  Example:

SELECT datname, numbackends, xact_commit, xact_rollback, blks_read, blks_hit

FROM pg_stat_database;

SELECT schemaname, relname,

```
    seq_scan, seq_tup_read, idx_scan, idx_tup_fetch
FROM pg_stat_all_tables
ORDER BY seq_scan DESC LIMIT 10;
```

---

3 Viewing Locks

Purpose: Use pg_locks to understand lock contention. Combine with pg_stat_activity to locate blocked sessions.
Script:

```
SELECT a.pid AS blocked_pid,
    a.query AS blocked_query,
    l.mode, l.granted,
    p.pid AS locker_pid,
    p.query AS locker_query
FROM pg_stat_activity a
JOIN pg_locks l ON a.pid = l.pid
JOIN pg_stat_activity p ON l.locktype = 'relation'
 AND l.granted = false
 AND p.pid = l.pid;
```

---

4 Progress Reporting

Supported for **ANALYZE, CLUSTER, COPY, CREATE INDEX, VACUUM, BASE_BACKUP**.
Postgre+1mstips.com+1wiki.postgre.org+9Postgre+9postgrespro.com+9

4.1 ANALYZE

```
SELECT pid, datname, relid::regclass AS table,
    phase, sample_blks_scanned, sample_blks_total,
    ext_stats_computed, ext_stats_total
FROM pg_stat_progress_analyze;
```

4.2 CLUSTER / VACUUM FULL

```
SELECT pid, relid::regclass AS table, phase,
    heap_blks_total, heap_blks_scanned,
    heap_blks_vacuumed
FROM pg_stat_progress_cluster;
```

4.3 COPY

```
SELECT pid, datname, command, bytes_processed,
    rows_removed, total_rows
FROM pg_stat_progress_copy;
```

4.4 CREATE INDEX / REINDEX

Use join with pg_stat_activity for context Data Egret+3mstips.com+3w3resource.com+3:

```
SELECT p.pid, p.relid::regclass AS table, p.index_relid::regclass AS index,
    p.phase, p.blocks_done, p.blocks_total,
    p.lockers_done, p.lockers_total
FROM pg_stat_progress_create_index p
JOIN pg_stat_activity a ON a.pid = p.pid;
```

4.5 VACUUM

```
SELECT pid, relid::regclass AS table, phase,
    heap_blks_total, heap_blks_scanned,
    heap_blks_vacuumed, index_vacuum_count
FROM pg_stat_progress_vacuum;
```

4.6 Base Backup

```
SELECT pid, datname, bytes_total, bytes_streamed, files_done
FROM pg_stat_progress_basebackup;
```

---

5 Dynamic Tracing

Overview: Use DTrace / LTTng / USDT probes built into Postgre (on supported OSes). Compiling with --enable-dtrace or --with-systemtap sets up built-in probes. Useful for advanced tracing at function-call level.
Script: Not standard ; requires building Postgre with tracing enabled and using OS tools like dtrace -n 'postgre$target::exec_simple_query:entry { printf("%s", copyinstr(arg0)); }'.

---

6 Monitoring Disk Usage

6.1 Determining Disk Usage

Use OS tools or functions:

```
SELECT pg_database.datname,
    pg_size_pretty(pg_database_size(datname)) AS size
FROM pg_database;
```

```
SELECT tablename,
    pg_size_pretty(pg_total_relation_size(tablename::regclass)) AS total_size
FROM pg_tables
WHERE schemaname='public';
```

6.2 Disk Full Failure

If server hits full disk, errors occur during commits or checkpointing; monitor with OS alerts (df -h) and Postgre logs. pg_stat_bgwriter provides checkpoint stats to detect abnormal behavior.

🛠️ **Option 1: Bash/p Monitoring Toolkit**

A lightweight, script-based toolkit using plain p, Bash, and watch to monitor key system views in real-time or on a schedule.

✅ **Installation**

- Install p and Bash on your monitoring server.
- Export PGHOST, PGUSER, PGPASSWORD, and PGDATABASE.

Example monitor_pg.sh Script

```
#!/usr/bin/env bash

# Query slow queries >60s

p -At -F',' -c "

SELECT pid, usename, datname,

    now() - query_start AS runtime,

    wait_event, query

FROM pg_stat_activity

WHERE state = 'active'

 AND now() - query_start > interval '60 seconds'

ORDER BY runtime DESC

;" > slow_queries.csv


# Replication lag (on primary)

/p -At -F',' -c "

SELECT application_name, write_lag, flush_lag, replay_lag

FROM pg_stat_replication;

" >> slow_queries.csv


# Database size summary

p -At -F',' -c "

SELECT datname, pg_size_pretty(pg_database_size(datname))

FROM pg_database;

" > db_sizes.csv


# pg_stat_bgwriter

p -At -F',' -c "
```

```
SELECT checkpoints_timed, checkpoints_req,
    buf_written_checkpoints, buf_clean
FROM pg_stat_bgwriter;
" >> bgwriter_summary.csv
```

Run periodically (e.g. cron or watch):

```
watch -n 60 bash monitor_pg.sh
```

Use \watch interactively in p:

```
SELECT * FROM pg_stat_activity; \watch 5
```

Stack Overflow+11Fastware Postgre+11DEV Community+11CommandPrompt Inc.+1PostgreScripts+1MS Tips+3PostgreScripts+3Postgre Wiki+3

Option 2: Prometheus Exporter + Grafana Dashboard

Ideal for long-term metrics collection, visualization, and alerting.

Setup Overview

1. Run Postgre Exporter via Docker or binary:

```
docker run -d \
 --name pg_exporter \
 -e DATA_SOURCE_NAME="postgre://user:pass@host:5432/db?sslmode=disable" \
 -p 9187:9187 \
 prometheuscommunity/postgres-exporter
```

Redrock Postgres+7w3resource+7Grafana Labs+7

2. Add to Prometheus scrape_configs:

```
scrape_configs:
 - job_name: pg_exporter
   static_configs:
    - targets: ['exporter-host:9187']
```

Postgre Europe+7w3resource+7HowtoForge+7

3. Import prebuilt Grafana dashboard (e.g. Grafana Labs Dashboard ID “Postgres Overview”):
    - Displays QPS, cache hit rate, connections, database sizes, locks & conflicts

Use the Grafana community dashboard as visual baseline:

- Panels include: Active sessions, replication lag, cache hit ratio, disk usage, WAL throughput

You can also configure alerts based on Prometheus rules (e.g., high latency, low cache hit, high replication lag).

📜 Option 3: Cheat-Sheet & Dashboard Quickref

A set of p views you can embed into Grafana dashboards via Postgre datasource.

Example Queries:

```
-- Active long-running sessions (>30s)
SELECT pid, usename, datname, now()-query_start AS runtime, wait_event, query
FROM pg_stat_activity
WHERE state = 'active'
  AND now() - query_start > interval '30 seconds';


-- Lock contention
SELECT a.pid AS blocked, a.query AS blocked_query,
    l.mode, p.pid AS locker, p.query AS locker_query
FROM pg_stat_activity a
JOIN pg_locks l ON a.pid = l.pid AND NOT l.granted
JOIN pg_stat_activity p ON p.pid = l.pid;
```

📊 Comparison Summary

Tool/Approach	Visualization	Alerting	Ease of Use	Ideal For
Bash / p scripts	CLI (watch/csv/log)	Manual	Very simple	Quick state checks, on-the-fly troubleshooting
Prometheus + Grafana	Rich Grafana panels	Fully automated	Moderate setup	Long-run monitoring, aggregation, alerting, dashboards
view cheat-sheet	p or Grafana UI	Conditional	Quick integration	Custom dashboards, ad-hoc insights

---

✅ **Recommendations**

- Start with the **Bash/p toolkit** to monitor key views (pg_stat_activity, pg_stat_bgwriter, pg_stat_database_conflicts, etc.).
- For scalable, long-term monitoring with visualization and alerting, implement **Postgres Exporter + Prometheus + Grafana**:
    - Quickstart guides and community dashboards are readily available tutorialdba.com+2Postgre+2PostgreScripts+2Redrock Postgres+5Grafana Labs+5Grafana Labs+5
    - Use PromQL (e.g. cache hit ratio, QPS, replication lag) for dashboards and alerts w3resourceDEV Community

---

When working with Postgre, especially if transitioning from a Microsoft Server background, several tips can help optimize scripts and database management:

1. Query Optimization:

- EXPLAIN and ANALYZE:

Regularly use EXPLAIN to see the query plan and EXPLAIN ANALYZE to execute the query and see the actual execution statistics. This helps identify bottlenecks and areas for improvement, similar to Server's execution plans.

- **Indexing:**

Create appropriate indexes, including covering indexes, to speed up query performance. Monitor for index bloat and use tools like pg_repack or pgstattuple to manage it.

- LIMIT Clause:

Use LIMIT to restrict the number of rows returned, similar to Server's TOP clause.

- PREPARE Statements:

Utilize PREPARE for frequently executed queries to reduce parsing overhead, analogous to prepared statements in Server.

- Avoid Wildcards at the Beginning of LIKE:

Similar to Server, avoid using wildcards at the beginning of LIKE clauses (%keyword) as it prevents index usage.

2. Database Maintenance:

- **Autovacuum Tuning:**

Adjust autovacuum settings (autovacuum_vacuum_threshold, autovacuum_analyze_threshold, etc.) based on your workload to ensure efficient cleanup of dead tuples and prevent table bloat.

- **WAL Archiving:**

Implement Write-Ahead Logging (WAL) archiving for point-in-time recovery and disaster recovery, a crucial aspect of database resilience.

- **Monitoring Slow Queries:**

Use the pg_stat_statements extension to identify and analyze slow or frequently executed queries, similar to Server's query store or profiler.

- **Monitor Current Activity:**

Use pg_stat_activity to monitor real-time database activity, including active connections, queries, and wait events, which can help diagnose performance issues.

3. Data Handling:

- **Data Types:**

Use efficient data types for columns, considering storage and performance implications.

- COPY Command:

Leverage the COPY command for fast bulk data loading, which is significantly more efficient than individual INSERT statements.

- **JSONB Type:**

For unstructured or semi-structured data, consider using the JSONB data type, but be mindful of potential bloat and manage it accordingly.

- **Foreign Keys:**

Establish foreign key constraints before loading data to ensure referential integrity, especially when dealing with dimensional modeling.

4. Migration and Compatibility (if applicable):

- **Server to Postgre Migration:**

If migrating, understand the differences in syntax and features. Tools and guides exist to assist with this process.

- **JDBC Driver (for Java applications):**

Ensure you have the correct Postgre JDBC driver for connecting Java applications to your Postgre database.

- **Scripting Differences:**

Be aware of syntax differences between T- (Server) and PL/pg (Postgre) when converting stored procedures, functions, and triggers.

🛠️ 1. Standard Unix Tools

🔍 Overview

Postgre doesn't reinvent the wheel — it leverages classic Unix tools like top, ps, vmstat, iostat, and strace for system-level monitoring.

🧪 Example

ps -ef | grep postgres

💡 Use Case

Track active Postgre processes, memory usage, and CPU load. Combine with pg_stat_activity for deeper insight.

📊 2. The Cumulative Statistics System

🔍 Overview

Postgre maintains internal statistics views that track activity, performance, and resource usage across sessions and operations.

⚙️ 2.1. Statistics Collection Configuration

🔍 Overview

Controls whether statistics are collected. Key GUCs: track_activities, track_counts, track_io_timing.

🧪 Example

SHOW track_activities;

ALTER SYSTEM SET track_io_timing = on;

SELECT pg_reload_conf();

💡 Use Case

Enable detailed performance metrics for query planning and IO analysis.

📈 2.2. Viewing Statistics

🔍 Overview

Use pg_stat_* views to inspect collected statistics.

🧪 Example

SELECT * FROM pg_stat_database;

💡 Use Case

Monitor query counts, tuples read/returned, and database-level activity.

👀 2.3. pg_stat_activity

🔍 Overview

Shows current sessions, queries, states, and wait events.

🧪 Example

SELECT pid, usename, state, query, wait_event FROM pg_stat_activity;

💡 Use Case

Identify long-running queries, blocked sessions, or idle connections.

🔁 2.4. pg_stat_replication

🔍 Overview

Tracks streaming replication status from the primary.

🧪 Example

SELECT pid, state, sent_lsn, write_lsn, flush_lsn, replay_lsn FROM pg_stat_replication;

💡 Use Case

Monitor replication lag and health of standby nodes.

📦 2.5. pg_stat_replication_slots

🔍 Overview

Shows logical/physical replication slot usage.

🧪 Example

SELECT * FROM pg_stat_replication_slots;

💡 Use Case

Detect slot bloat or unused slots that prevent WAL cleanup.

📡 2.6. pg_stat_wal_receiver

🔍 Overview

Provides WAL receiver status on standby nodes.

🧪 Example

SELECT * FROM pg_stat_wal_receiver;

💡 Use Case

Validate WAL streaming, connection status, and sync lag.
