EDB POSTGRES AI

Best Practices for High Availability and Disaster Recovery

Rahul Saha

enterprisedb.com

Table of contents

Introduction

In an era in which business continuity and data integrity are paramount, a robust database solution is not a luxury—it's a necessity. This white paper explores how EDB Postgres®, an enhanced version of the popular open source PostgreSQL database, provides a resilient foundation for mission-critical applications. It will delve into how EDB Postgres leverages a combination of enterprise-grade tools and architectural designs to ensure high availability (HA), disaster recovery (DR), and comprehensive backup and recovery. By examining key features such as bidirectional replication and multi-region portability, this paper will illustrate how EDB Postgres protects against a spectrum of failures, from localized hardware issues to catastrophic, site-wide disasters, while also enhancing performance for a globally distributed user base. Ultimately, this paper will demonstrate how EDB Postgres offers a complete solution for safeguarding data and maintaining uninterrupted service.

At the core of this resilience is a multilayered strategy. HA ensures that the database remains operational, with minimal interruption, by using tools such as EDB Failover Manager for automated failover within a single site. This addresses immediate threats such as hardware failures, ensuring that a standby server can instantly take over. For larger-scale events, DR and multi-region portability become paramount. EDB Postgres Distributed (PGD) enables a powerful active-active architecture, allowing the database to span multiple data centers or cloud regions. This not only provides a fallback in case an entire region is lost but also reduces latency for global users by allowing them to connect to the nearest database node.

Finally, the foundational safety net for any database system is a robust backup and recovery mechanism. EDB Postgres utilizes tools such as Barman, which provide a highly efficient and reliable way to create and manage backups. It supports crucial features such as point-in-time recovery (PITR), which allows the database to be restored to any specific moment in time, a critical capability for recovering from accidental data deletion or corruption. Together, these tools—Failover Manager, PGD, and Barman—form a holistic and cohesive strategy that ensures that EDB Postgres is not only a powerful database but a secure, available, and resilient platform for the most demanding enterprise workloads.

High availability and disaster recovery

1. High availability (HA)

HA is about keeping the database operational with minimal interruption. It's the "here and now" protection, focusing on rapid failover. For EDB PostgreSQL, HA is typically achieved through replication: A primary database server constantly replicates data to one or more standby servers. If the primary server fails, a standby can quickly take over, often automatically, to become the new primary. This prevents a complete service outage.

Key benefits include:

- **Minimizing downtime:** HA ensures that the database remains accessible even if a single server or component fails.
- **Load balancing:** Standby replicas can also be used for read-only queries, distributing the workload and improving performance.
- **Maintenance:** HA allows for planned maintenance on one server without bringing down the entire system.

2. Disaster recovery (DR)

DR is about preparing for catastrophic events, such as a fire, flood, or a large-scale power outage that affects an entire data center. It's the "just in case" plan. DR for EDB PostgreSQL involves setting up a secondary site, often in a different geographical location, that can take over if the primary site is destroyed or becomes inaccessible. This is typically accomplished through asynchronous replication or by shipping logs to a remote location. The goal is to restore services with a minimal loss of data. Key benefits include:

- **Protecting against catastrophic failures:** DR ensures business operations can resume even after a major disaster.
- **Geographical redundancy:** It provides a fallback in case of a regional disaster.
- **Compliance:** Many regulatory standards require a robust DR plan.

3. Recovery point objective (RPO)

RPO represents the maximum tolerable amount of data loss an organization can endure during an outage. It is expressed in terms of time, i.e., how far back in time data must be restored to resume normal operations.

RPO = How much data can I afford to lose?

Business perspective:

- **If RPO is hours**, it implies the business can afford to reenter several hours of lost data.
- **If RPO is seconds**, it suggests the business is highly sensitive to data loss (e.g., financial transactions, healthcare systems).

Example:

An RPO of 15 minutes means backups, replication, or journaling must be frequent enough that in case of a failure, the business will lose at most 15 minutes of data.

Key factors influencing RPO:

- **Criticality of data** (customer orders vs. internal reports)
- **Regulatory requirements** (compliance may dictate near-zero RPO)
- **Cost vs. tolerance trade-off** (lower RPOs usually require more advanced replication and higher investment)

When streaming replication is set up, there is continuous replication of WAL to standby servers or backup locations. This minimizes RPO, since data is replicated almost in real time.

Tools such as EDB Postgres Enterprise Manager (PEM) proactively monitor replication lag, backup health, and failover readiness. This ensures that SLAs for RPO/RTO are consistently met.

4. Recovery time objective (RTO)

RTO is the maximum acceptable duration of downtime after a failure before operations must be restored. It answers the question: “How quickly do we need to be back online?”

RTO = How long can I afford to be down?

Business perspective:

- **A short RTO** (seconds/minutes) is critical for systems that must be always available (e.g., online banking, e-commerce checkout).
- **A longer RTO** (hours/days) may be acceptable for less critical systems (e.g., reporting dashboards).

Example:

An RTO of one hour means that, after an outage, the database infrastructure must be operational and accessible within 60 minutes.

Key factors influencing RTO:

- **Customer expectations** (SaaS platforms promising 99.99% uptime must design for near-instant recovery)
- **Operational dependencies** (systems that other applications rely on may demand stricter RTOs)

- **Budget and technology** (shorter RTOs typically require clustering, automated failover, and replication infrastructure)

EDB PGD or Failover Manager ensure automatic failover to standby nodes if the primary goes down. This reduces RTO (downtime) by quickly promoting a standby.

EnterpriseDB (EDB) backup tools including Barman provide scheduled, incremental, and full backups with PITR, ensuring that databases can be restored to any given moment, aligning with tight RPOs.

EDB PGD enables multiple active nodes across regions. Practically, it achieves near-zero RPO and very low RTO, since another node can immediately take over.

5. Zero data loss (RPO = zero)

Achieving RPO = 0, which is no data loss even during catastrophic failures, requires careful synchronization between two or more data centers. PostgreSQL's native synchronous replication can provide zero-data-loss protection within a region; however, across geographically separated data centers, network latency and durability guarantees become challenging. EDB PGD uniquely addresses this challenge by combining bidirectional replication, distributed commit protocols, and transaction-level consistency guarantees.

To achieve zero RPO, the architecture typically includes:

- **Two active PGD nodes**, one in each data center, both capable of handling read and write operations
- **Synchronous replication groups** configured between data centers to ensure that each committed transaction is acknowledged by both before success is returned to the client
- **Automatic failover and promotion logic** that preserves consistency in the event of a site-wide failure

PGD's distributed transaction manager ensures **atomicity and durability** across nodes, preventing "split-brain" scenarios while guaranteeing zero data loss. This approach differentiates PGD from traditional asynchronous or semi-synchronous architectures, in which small replication delays can lead to data gaps during failover.

While many modern distributed databases achieve global consistency through consensus-based replication mechanisms, EDB PGD provides equivalent RPO = 0 guarantees while maintaining complete compatibility with the PostgreSQL ecosystem. EDB PGD extends the native PostgreSQL architecture with bidirectional, distributed transaction capabilities that ensure data durability and consistency across geographically separated nodes. Unlike systems that rely on forced global consensus for every transaction, EDB PGD allows organizations to maintain logical data boundaries, implement region-specific replication policies, and design geo-distributed architectures that align with regulatory and performance requirements. This approach provides the flexibility to operate in hybrid or multi-cloud environments, meet stringent data residency and compliance standards, and still benefit from PostgreSQL's mature tooling, extensions, and performance characteristics—all while ensuring zero data loss in the event of regional or data center–level failures.

Achieving true global resilience requires navigating the CAP trade-off—the balance between consistency, availability, and partition tolerance. In distributed architectures, perfect consistency across distant regions can increase transaction latency, while prioritizing local availability may introduce temporary divergence. EDB PGD provides the flexibility to fine-tune this balance based on business and compliance requirements. Through configurable synchronous and asynchronous replication modes, EDB PGD allows organizations to prioritize zero-data-loss (RPO = 0) consistency for critical transactions within or across data centers, while using asynchronous propagation for less sensitive or latency-tolerant workloads. This hybrid approach enables enterprises to achieve both regulatory-grade data protection and application-level performance optimization. By giving architects granular control over commit policies, replication sets, and regional failover boundaries, EDB PGD ensures that every deployment can be precisely aligned with its operational priorities—whether the goal is absolute consistency, minimal latency, or a compliant middle ground.

Zero-data-loss DR

- **Dedicated low-latency link:** A low-latency, high-bandwidth connection between data centers is the backbone of a zero-data-loss architecture. In synchronous or quorum-based replication, every transaction commit requires confirmation from remote nodes before it is acknowledged to the client. Network latency directly affects commit time; therefore, a private, high-speed, and encrypted connection with round-trip times below 5 milliseconds (RTT <5ms) is recommended. This link should be provisioned over redundant paths to minimize jitter and packet loss. Using private peering or dedicated interconnects between cloud regions can further improve consistency and isolation from public network instability. Continuous monitoring of bandwidth utilization and replication lag across this link ensures that performance and RPO = 0 objectives remain aligned.
- **Synchronous commit policy:** For critical workloads, it is important that each transaction achieves full durability across data centers before client acknowledgment. PGD supports this through the synchronous_ commit=remote_apply configuration, which ensures that a transaction is not considered complete until it has been written to and applied on at least one remote node. This approach provides true zero-data-loss protection (RPO = 0) while maintaining transactional integrity across regions. However, synchronous commits do introduce latency overhead proportional to the network distance between nodes. To balance this, many organizations configure synchronous replication for business-critical schemas or tables while using asynchronous modes for lower-risk workloads. This selective approach allows for predictable performance without compromising durability for the most important data.
- **Quorum-based commit rules:** PGD enhances traditional synchronous replication with quorum-based commit rules, allowing administrators to define how many acknowledgments are required before a transaction is committed. For example, a `1 local + 1 remote` quorum ensures that at least one remote data center confirms the transaction, achieving RPO = 0 while maintaining local responsiveness. This configuration provides flexibility to tune durability versus latency depending on workload and topology. In multi–data center deployments, quorum policies can be adjusted dynamically to handle maintenance or transient network degradation. By using quorum commit rules, organizations can maintain strong consistency without sacrificing availability or performance under varying operational conditions.
- **Geo-aware topology:** In global or regional deployments, physical distance inevitably affects transaction latency. It is thought to be best practice to design a geo-aware topology in which nodes are placed strategically close enough to support synchronous replication while still offering geographic redundancy. When ultra-low-latency connections are not feasible, PGD supports region-local writes with asynchronous global propagation, allowing each region to handle its own write operations independently while replicating changes asynchronously to others. This hybrid model provides a balance between local performance and global consistency. Each topology should be validated against the organization's RTO and RPO to confirm that performance and data durability goals are met.
- **Independent power and network domains:** Achieving true HA requires eliminating shared points of failure between data centers. Each participating site must have independent power supplies, networking infrastructure, and storage systems to ensure resilience during catastrophic events. This separation guarantees that a failure in one domain—whether caused by utility outages, routing issues, or hardware corruption—does not cascade across the entire system. PGD's distributed architecture leverages these isolated domains to maintain transactional integrity even when one data center becomes completely unavailable. Independent monitoring and alerting systems should also be deployed per site to maintain visibility during inter-site failures.
- **Failover automation with PGD CLI:** Automated failover is essential to minimize downtime during unplanned outages. Failover Manager and PGD's built-in coordination layer both provide controlled failover mechanisms that promote standby or secondary nodes automatically while preserving data consistency. Failover Manager monitors node health and network connectivity, triggering failover events when primary nodes become unresponsive. In multi–data center PGD deployments, PGD Command Line Interface (CLI) can also manage cross-region failovers using predefined rules to ensure deterministic, conflict-free promotion. Automation removes the risk of manual intervention delays and ensures that RTO objectives are consistently met. Regular simulation of failover events validates that these processes perform reliably under production conditions.

- **Regular DR testing:** Even the most advanced HA/DR design is only as effective as its tested execution. Regular DR drills help validate not just system resilience but also the operational readiness of the supporting teams. These tests should simulate complete data center loss, network partitioning, or corruption scenarios and measure the resulting RTO and RPO against defined SLAs. Testing should include full verification of data consistency across data centers after recovery and restore operations. Over time, these exercises build organizational confidence and reveal any procedural or configuration gaps. A documented test schedule—combined with automated failover verification—ensures continuous improvement of the DR posture.

EDB Postgres HA architecture

1. Physical replication

Physical streaming replication is the most widely used mechanism for achieving HA in PostgreSQL. It streams WAL records from the primary to standby nodes, creating byte-for-byte replicas. HA in EDB PostgreSQL is built on physical streaming replication. This mechanism ensures that a standby server maintains a nearly real-time copy of the primary server's data, allowing rapid recovery and minimal disruption in case of failure.

A. Streaming replication in EDB Postgres

- **WAL mechanism:** PostgreSQL uses WAL files to ensure durability. The WAL is at the heart of PostgreSQL's durability and crash recovery mechanism. Every change to the database—whether it's inserting a row, updating a value, or deleting data—is first recorded in the WAL on the primary before being applied to the data files. This ensures that PostgreSQL can recover from crashes by replaying the WAL and restoring the database to a consistent state. These WAL records can be continuously shipped and applied to a standby server.
- **WAL file storage:** By default, WAL files are stored in the directory $DATADIR/pg_wal, where $DATADIR is the main PostgreSQL data directory. This directory holds the active WAL files as well as recycled ones. Backup tools (e.g., `pg_basebackup`, Barman, pgBackRest) and replication mechanisms also read from this location to archive WAL segments or stream them to standby nodes.
- **WAL file size:** Each WAL file size is usually 16 MB. This size is not arbitrary—it is chosen as a balance between disk I/O efficiency and manageability. For example, smaller files would cause too many file rotations, increasing overhead, whereas larger files would delay archiving and replication. Please note that PostgreSQL 11+ allows customizing WAL segment size at cluster initialization (--wal-segsize), but 16 MB remains the standard default.
- **WAL file naming convention:** WAL filenames are 24-character hexadecimal strings that encode information about the timeline, log, and segment. Example: `00000001000000AC00000006`

  This can be broken down into three parts:

  1. **Timeline ID (first eight characters):** The first eight characters, '`00000001`', identify the timeline the WAL belongs to. In general, timelines are important in recovery and replication scenarios. For example, if a failover or recovery event occurs and the database diverges, a new timeline is created. This prevents conflicts between old WAL sequences and new WAL sequences.
  2. **Log ID (middle eight characters):** The middle 8 characters, '`000000AC`', represent the high-order part of the WAL position. They increase as WAL activity progresses and more logs are generated.
  3. **Segment ID (last eight characters):** These characters, '`00000006`' in the example above, identify the segment within the current log. Since each segment is 16 MB, this increments each time a new file is filled.
- **Concept of WAL logs and segments:** A log is composed of many segments. In PostgreSQL, the terms *WAL file* and *WAL segment* are often used interchangeably. Both refer to the same 16 MB chunk. For example, log `000000AC` could consist of segments `00000000`, `00000001`, `00000002`, etc., each 16 MB in size. Collectively, these form the WAL for that log.

- **No server identifier in filenames:** WAL filenames do not contain explicit identifiers for the PostgreSQL instance that created them. This is why archiving and organizing WALs properly is critical—tools such as Barman and `pgBackRest` add metadata to ensure that WALs are tracked by server and timeline.
- **Recycling of WAL segments:** PostgreSQL reuses old WAL files when possible to minimize file system overhead. This is why the number of files in `pg_wal/` may not grow indefinitely (unless WAL archiving or replication is misconfigured).
- **Critical role in backups and PITR:** WAL files, together with a base backup, are what enable PITR. By replaying WAL files up to a chosen moment, PostgreSQL can restore the database to any exact state in the past.
- **Primary-standby architecture:** In a typical EDB HA setup, one primary node and one or more standby nodes are present. Standbys are maintained through streaming replication; WAL changes are streamed in real time over a network connection. Standby servers accept read-only queries, which is helpful for offloading workload from the primary.
- **Streaming replication process:** The WAL sender (walsender) process on the primary continuously streams WAL changes to the WAL receiver (walreceiver) process on the standby. The standby replays the WAL records, keeping its dataset synchronized with the primary. Replication can be configured as synchronous or asynchronous replication.
    - **Synchronous:** In synchronous replication, commit on primary waits for confirmation from at least one standby. It ensures zero data loss (RPO = 0) but can impact performance. Synchronous replication should be used for mission-critical systems that require zero data loss (RPO = 0). In this mode, a transaction is not considered committed until at least one standby acknowledges receipt of WAL data.
    - **Asynchronous:** In asynchronous replication, primary does not wait for standby acknowledgment. It provides better performance but can result in minimal data loss (small RPO). For workloads in which performance is more critical than absolute zero data loss, use asynchronous replication. This delivers lower latency but accepts a small RPO (a few seconds of potential data loss).

It is important to maintain at least one synchronous standby (for data durability) and one or more asynchronous standbys (for geographic redundancy and read scaling). Distributing standbys across different availability zones or data centers helps to mitigate site-level failures. In an ideal production environment, the focus should be on low RTO and RPO, along with effective load balancing.

- **Replication slot:** EDB Postgres supports replication slots, which ensure that the primary retains WAL files until all subscribed standbys confirm receipt. This prevents a standby from falling behind or missing data during transient network issues. Configuring replication slots on the primary helps to guarantee WAL retention until all standbys have consumed the data. It is important to monitor replication lag and slot usage to prevent disk bloat if a standby falls behind.
- **Cascading replication:** In cascading replication, standby servers themselves can act as WAL senders, relaying WAL to other standby nodes, which improves scalability in geographically distributed HA architectures. In geographically distributed HA designs, use cascading replication so that a remote standby receives WAL from a closer standby, rather than directly from the primary. This reduces network overhead, improves efficiency, and adds resilience for large-scale DR topologies.

B. Benefits of EDB streaming replication

- **Minimized RPO through near-real-time replication:** By continuously streaming WAL records from the primary to standby servers, EDB ensures that data loss is reduced to seconds or eliminated entirely when synchronous replication is used. Replication mode can be aligned with business-criticality: synchronous for zero data loss, asynchronous for performance-sensitive systems.
- **Low RTO with automated or manual failover:** Standby servers can be promoted to primary in seconds or minutes, especially when combined with Failover Manager for automation. Test failover procedures regularly to validate that recovery time aligns with defined SLAs.

- **Improved scalability via read-intensive workload distribution:** Standby servers can be configured in hot standby mode to handle read-only queries. In the production environment, standby can be used to offload reporting, analytics, and read-heavy workloads to these replicas, reducing pressure on the primary while maintaining HA.
- **Deployment flexibility for varied business requirements:** Streaming replication supports both synchronous and asynchronous modes. It is always good to use a hybrid strategy—one synchronous standby for durability and one or more asynchronous standbys for geographic redundancy and performance.
- **Enhanced resilience with replication slots and cascading replication:** Replication slots ensure that WAL data is preserved until standbys have applied it, preventing data loss during temporary network issues. Cascading replication further strengthens DR designs by enabling remote standbys to replicate from a closer intermediary standby. It is always good to combine both features in multisite or geo-distributed architectures for maximum resilience.

C. Failover and switchover

Failover

Failover is a mechanism that is triggered when the primary database becomes unavailable (e.g., due to hardware, network, or site failure) and a standby is automatically or manually promoted to take over as the new primary. In a production environment, the goal is to minimize downtime (RTO) and prevent unacceptable data loss (RPO). If it is not managed carefully, failover can lead to split-brain scenarios (two primaries active simultaneously) or data inconsistency.

Failover Manager can automate failover to reduce human error and recovery time. To achieve zero data loss (RPO = 0), at least one synchronous replication is required to be configured. Failover Manager maintains quorum-based decision-making to avoid false promotions. It is important to test failover scenarios at regular intervals to validate RTO targets.

Switchover

Switchover is a planned role reversal between primary and standby, typically for maintenance, patching, or upgrades. Unlike failover, the original primary is still healthy. In general during the switchover process, standby is demoted to primary, then the original primary is demoted to standby.

Switchover is generally performed during maintenance windows after verifying replication health. Ensuring that there is no replication lag between primary and standby before initiating is a good practice. Failover Manager–controlled switchover procedures should be used to handle role changes cleanly and to prevent disruption. Multiple rehearsals should be practiced for the switchover process in non-production environments to avoid surprises during production changes.

EDB Failover Manager

Failover Manager provides a lightweight agent-based solution for managing failover and switchover in physical replication setups. While configuring Failover Manager, the deployment should consist of at least three Failover Manager agents (e.g., one per node plus a witness) to ensure reliable quorum-based decisions.

Failover Manager manages to avoid a split-brain scenario after the failover. In the case of a primary database failure, Failover Manager promotes a standby to be the new primary, and it also makes sure that the old primary cannot restart by creating a recovery.conf file in the data directory. This avoids any kind of split-brain scenario, especially if the primary failure was temporary or a DBA manually restarts the server. In case of a primary node failure, after failover, if the primary node comes back, then the Failover Manager agent checks the status of the cluster with other Failover Manager agents in the cluster. In case Failover Manager agents confirm the availability of another primary in the same cluster, then a Failover Manager agent on the old primary node creates a recovery.conf to avoid the possible split-brain situation.

Failover Manager can failover automatically or manage controlled switchover:

- **Automatic failover:** Failover Manager continuously monitors the health of the primary. If a failure is detected and confirmed by a quorum, it promotes the most up-to-date standby. Further, clients are redirected to the new primary automatically.
- **Controlled switchover:** Failover Manager supports manual switchover commands, allowing administrators to safely reassign roles without breaking replication. The original primary can be rejoined as a standby after role reversal.

D. Cascading replication

In high availability and disaster recovery (HA/DR) architectures, it is common to deploy multiple standby servers for redundancy, geographic resilience, and workload distribution. However, as the number of standbys increases, the primary server can become a bottleneck because it must stream WALs to every standby directly. This can lead to higher CPU utilization, increased network bandwidth consumption, and potential replication lag.

Cascading replication addresses this challenge by allowing a standby server to act as a replication source for other standbys. Instead of all standbys pulling WAL data from the primary, a subset of them can replicate from an intermediary standby.

Offload WAL streaming from the primary

Using cascading replication when deploying more than a few standbys to distribute replication traffic across multiple nodes reduces CPU and network load on the primary, ensuring that it remains focused on serving application traffic.

Optimize for geographic scalability

Positioning intermediary standbys close to downstream standbys in the same region or availability zone minimizes network latency and cross-region bandwidth usage, making replication more efficient in multi-region HA/DR designs.

Design with redundancy at every level

For critical workloads, configuring some standbys to replicate directly from the primary alongside cascading standbys avoids a single point of failure in cascading chains. It ensures that downstream nodes remain available even if an intermediate standby fails.

Monitor and control replication lag

Using monitoring tools such as EDB Postgres Enterprise Manager to track lag not only between the primary and its standbys but also between cascading nodes and their downstream replicas, which ensures cascading, does not introduce unacceptable delays that would compromise RPO objectives.

Combine carefully with replication slots

Use replication slots on intermediaries to prevent data gaps in downstream standbys, but regularly monitor WAL storage growth to avoid disk exhaustion, which guarantees downstream continuity while maintaining stability at the intermediary level.

Monitoring physical replication

Physical replication (streaming replication) relies on WAL being shipped from a primary to one or more standby servers. PostgreSQL exposes detailed state information on both sides.

On the primary

Postgres exposes all the details in the view ‘`pg_stat_replication`’. It lists one row per connected standby (streaming or cascading). Usually replication lag is measured by comparing `sent_lsn` with `replay_lsn`. Some key columns are listed below:

```
pg_stat_replication

        pid → process ID of WAL sender

        usename, application_name → identify replication client

        state → replication state (startup, catchup, streaming)

        sent_lsn → latest WAL LSN sent

        write_lsn, flush_lsn, replay_lsn → how far the standby has written, flushed, and
        replayed WAL

        sync_state → sync, async, or potential (important for synchronous replication)
```

pg_stat_wal_sender(v15+)

It exposes more detailed stats for WAL sender processes, including WAL throughput.

Pg_stat_archiver

This tracks WAL files successfully archived (or failed attempts). It is useful in WAL shipping + PITR setups (important when backups are part of DR).

pg_stat_wal (v13+)

It provides stats about WAL generation (volume, rate, FPI count). It helps detect workload patterns or WAL storming that may stress replication.

Functions for inspection

- `pg_current_wal_lsn() → shows current WAL position on primary`
- `pg_current_wal_insert_lsn() → shows WAL position of last inserted record`

On the standby

Postgres also exposes the views in standby servers for monitoring. The key views are as follows:

`pg_stat_wal_receiver:` This shows the state of the WAL receiver of one row. This view is used to validate whether the standby is receiving WAL without lag or disconnects.

The key columns to be followed are:

- status → whether receiving, waiting, or stopped
- `receive_start_lsn`, `received_lsn`, `latest_end_lsn` → current WAL positions
- `slot_name` → replication slot being used (if configured)

`pg_last_wal_receive_lsn() / pg_last_wal_replay_lsn()`

This function checks how far the standby has received and replayed WAL. The apply lag can also be calculated by `pg_last_wal_receive_lsn() vs pg_last_wal_replay_lsn()` difference.

`pg_is_in_recovery()`

This returns true on standby (in recovery). It is useful in scripts or monitoring probes.

Replica lag

In physical replication, lag refers to the difference between the primary server's current WAL position and the point up to which a standby has written, flushed, or replayed changes. Since physical replication streams raw WAL data, even small amounts of lag can affect failover readiness and RTOs. Lag is most often caused by network throughput limitations, I/O bottlenecks on standby storage, or excessive workload spikes on the primary that generate WAL faster than it can be consumed. Monitoring this lag closely is a critical best practice, as unmanaged replication delays can lead to data loss in failover scenarios, jeopardizing RPOs. By tracking write, flush, and replay positions, and by proactively tuning the infrastructure, organizations can maintain tighter consistency across nodes and ensure HA in production environments.

Replication lag = time or WAL distance between primary commit and standby apply.

The query below can be used to monitor the replication lag. It shows byte-lag per standby:

```
SELECT

  client_addr,

  state,

  sent_lsn,

  write_lsn,

  flush_lsn,

  replay_lsn,

  pg_wal_lsn_diff(sent_lsn, replay_lsn) AS byte_lag

FROM pg_stat_replication;
```

Standby perspective: This shows how far behind standby is in replaying WAL:

```
SELECT

  pg_last_wal_receive_lsn(),

  pg_last_wal_replay_lsn(),

  pg_wal_lsn_diff(pg_last_wal_receive_lsn(), pg_last_wal_replay_lsn()) AS apply_lag;
```

Best practices

- Deploy at least one synchronous standby to guarantee no data loss during a primary node failure.
- Add asynchronous standbys for geographic redundancy, accepting some replication lag in exchange for broader coverage.
- Use cascading replication to offload replication traffic from the primary and improve scalability.
- Monitor replication lag continuously, since lag directly impacts the RPO.

2. Logical replication

Logical replication in PostgreSQL provides fine-grained control over data replication by allowing specific tables, schemas, or subsets of data to be replicated from a publisher to one or more subscribers. Unlike physical replication, which maintains a full copy of the primary database, logical replication supports use cases such as selective data distribution, multi-version upgrades, and integration with external systems. As a best practice, users should use logical replication for scenarios that demand flexibility—such as online migrations, cross-version upgrades, and partial replication across environments—while continuing to rely on physical streaming replication for HA and DR. Combining both approaches in a hybrid architecture ensures operational efficiency, scalability, and resilience aligned with business continuity goals.

A. Use cases for logical replication

Selective replication

Logical replication allows users to replicate only specific tables or subsets of data. This is valuable when not all data needs to be shared with downstream systems. As an example, consider a scenario in which a user wants to replicate only customer-facing tables to a reporting or analytics cluster while excluding internal logs and archival data. In those scenarios, it is important to carefully scope the publication to avoid unnecessary replication overhead. Also, continuously monitoring replication lag ensures that selective replication does not compromise RPO objectives.

Bidirectional or active-active architectures

When combined with conflict management frameworks such as EDB PGD, logical replication supports bidirectional setups in which multiple nodes can accept writes. As an example, consider a scenario in which a global e-commerce platform, with write activity occurring in multiple regions, requires continuous replication to maintain consistency. In such a scenario, it is important to implement robust conflict detection and resolution policies and to use synchronous replication where possible to maintain data integrity. Also, regularly test conflict scenarios, a step critical for production readiness.

Cross-version upgrades and online migrations

Logical replication enables replication between different PostgreSQL versions, making it a powerful tool for upgrades and platform migrations. As an example, a scenario can be considered in which migration from PostgreSQL version 13 to 16 is performed without downtime by continuously replicating data until the switchover. As a good practice, perform replication dry-runs before the upgrade window, validate data consistency between publisher and subscriber, and plan a controlled cutover with minimal RTO.

Data integration across heterogeneous environments

Logical replication supports replication not just within PostgreSQL clusters but also into external systems via connectors, enabling hybrid architectures. As an example, consider streaming sales transactions from PostgreSQL into a downstream data warehouse for real-time analytics.

Workload isolation and scaling

Logical replication can replicate specific workloads to secondary clusters dedicated to analytics, reporting, or testing, keeping the primary optimized for online transaction processing (OLTP) traffic. As an example, users can offload heavy reporting queries from the production primary into a subscriber cluster dedicated to BI workloads. In such scenarios, users can replicate only required datasets and should validate query performance on the subscriber regularly. Also, ensure that replication lag does not affect reporting accuracy.

B. Logical replication for multi-region architecture

As enterprises expand globally and adopt hybrid or multi-region strategies, database replication requirements extend beyond traditional HA/DR. Logical replication in PostgreSQL enables granular and flexible replication across regions by publishing and subscribing to changes at the table or schema level. Unlike physical replication, which requires identical binary copies, logical replication is protocol-based and can operate across heterogeneous environments, different PostgreSQL versions, and varied infrastructure landscapes.

Selective data sharing

Logical replication can be used to allow only business-critical tables to be replicated across regions, reducing network overhead and enabling compliance with data residency laws.

Regional write capability with PGD

When combined with EDB PGD, logical replication supports bidirectional topologies, enabling local writes in different regions while synchronizing data globally.

Resilience against regional failures

Logical replication ensures that downstream subscribers remain up to date, providing continuity in case of a primary region outage.

As a good practice, deploying logical replication with regional publication/subscription pairs can minimize cross-region replication lag. Use synchronous replication for mission-critical datasets where RPO must be zero, and asynchronous replication for less-sensitive data to improve performance.

C. Logical replication for multi-cloud architecture

Logical replication empowers enterprises to extend PostgreSQL across multiple cloud providers, enabling data distribution, resilience, and workload optimization at a global scale.

Cross-cloud portability

Logical replication enables seamless replication between PostgreSQL instances running on different cloud providers (e.g., AWS, Azure, GCP, or private cloud).

Hybrid cloud scenarios

Logical replication allows enterprises to replicate workloads from on-premises data centers into public clouds for elasticity, analytics, or DR.

Cloud exit strategies

By maintaining replicated copies across providers, organizations avoid vendor lock-in and gain the flexibility to rebalance workloads.

Use logical replication to segment workloads across clouds (e.g., OLTP in one cloud, analytics in another). Also, always encrypt replication traffic between providers, and monitor replication lag closely so that SLAs are consistently met.

Operational best practices for multi-region and multi-cloud logical replication:

- **Latency management:** Place subscribers close to end users to reduce query latency, while also monitoring cross-region lag to ensure that RPO targets are consistently achieved.
- **Conflict resolution:** In bidirectional replication setups, establish clear conflict resolution rules so that data inconsistencies are avoided.

- **Monitoring and observability:** It is beneficial to use tools such as EDB Postgres Enterprise Manager or equivalent solutions to track replication health across distributed clusters.
- **Security hardening:** It is always recommended to secure replication channels with SSL/TLS and to manage access controls carefully across cloud boundaries.
- **Testing across providers:** It is good practice to regularly test failover and switchover in multi-region or multi-cloud topologies so that both RTO and RPO objectives are validated under real-world failure conditions.

D. EDB Postgres Distributed (PGD)

In enterprise-grade PostgreSQL deployments, use advanced clustering and replication tools such as EDB PGD to manage failover and switchover more effectively. Unlike traditional single-primary streaming replication, PGD provides a bidirectional architecture in which each node can accept writes and data is synchronized across the cluster with strong consistency guarantees. This makes it highly suitable for mission-critical workloads that demand both HA and geo-distributed resilience.

Failover with PGD

Failover occurs when the active node becomes unavailable due to a failure. With PGD, failover is not limited to promoting a single standby; instead, another active node in the cluster can continue to serve write traffic without disruption.

Configure PGD clusters with nodes distributed across different availability zones or regions, so that failover is not constrained to a single site. Use synchronous replication policies for critical data sets for which RPO must be zero, while adopting asynchronous replication for less-sensitive data to balance performance. Test automated failover regularly to validate that the cluster can meet defined RTO objectives under real-world failure conditions.

Switchover with PGD

Switchover refers to a planned role change, typically for maintenance or upgrades. With PGD, switchover is simplified because every node is capable of serving as a write node. Administrators can gracefully redirect applications to another node while performing maintenance on the original one.

It is best to perform switchover only after verifying replication health and ensuring that all nodes are synchronized. Coordinate switchovers during planned maintenance windows, minimizing impact on applications. Also, document and rehearse switchover procedures as part of standard operational playbooks.

Regional data localization

In modern, regulated environments, data localization is not only a design choice—it is a compliance requirement. Many jurisdictions mandate that specific categories of data (such as financial transactions or personal user information) remain within defined geographic boundaries. PostgreSQL, when combined with EDB PGD, provides a flexible foundation to enforce these localization rules without sacrificing HA or global performance.

A regional data-localization pattern ensures that data for a given geography is both written and replicated within that region, while global coordination and read access can be managed across multiple regions. PGD's bidirectional replication allows each region to operate autonomously, maintaining write availability even during cross-region network disruptions.

Regional data-localization architecture

- **Regional write primaries:** PGD's active-active architecture allows each region to have its own local writable node or shard, while maintaining logical consistency across the distributed cluster. Each node (or group of nodes) in PGD can be configured as a write leader for its region. Writes from local users are handled by the nearest regional node, reducing latency and meeting data-residency requirements. PGD supports replication sets and sequence synchronization, ensuring conflict-free multi-region write operations.

- **Controlled cross-region replication:** PGD provides fine-grained replication control through replication sets—logical groupings of tables or schemas that define what data is replicated to each node. Replication sets can be defined for sensitive vs. nonsensitive data. For instance, customer personally identifiable information (PII) stays within the EU node, while anonymized analytics data replicates globally. Logical replication sets can be used to separate sensitive data domains and control data movement between regions. This aligns with GDPR, RBI, and other data-localization frameworks.
- **Latency-aware routing:** Deployment of application-level routing is one option, as is use of a connection manager (such as pgBouncer or HAProxy), which directs user traffic to the nearest regional database node. It is best to use read-write routing policies that send write operations to the local regional node and also use read scaling (read-only replicas) for reporting workloads within each region.
- **Consistent policy enforcement:** PGD supports logical replication with row filtering, replication sets, and custom triggers that allow organizations to enforce region-based data rules. PGD's replication filtering can be used to prevent cross-region data flow of certain records.
- **Geo-fenced backups:** PGD nodes can be backed up independently, enabling region-specific geo-fenced backups. This maintains regionally isolated backup sets managed by tools such as Barman or pgBackRest, with retention and encryption policies tailored to local regulations.
- **Failover boundaries:** PGD provides autonomous failover within replication groups (regional clusters) while maintaining asynchronous or logical links between regions for DR. It restricts failover events to remain within a region. Also, it uses asynchronous or logical replication to maintain read-only replicas in other regions for DR purposes.

This pattern balances compliance with resilience—enabling organizations to meet localization mandates while ensuring that mission-critical services remain available, performant, and recoverable at all times.

Monitoring logical replication

Logical replication uses publications (on primary) and subscriptions (on standby/replica). Monitoring must be done at both ends.

On the publisher (primary)

- **`pg_stat_replication`:** Even logical replication connections appear here as WAL senders. These connections can be distinguished via the use of the application_name (subscription name).
- **`pg_publication / pg_publication_tables`:** This view has the metadata about publications and the tables included. It is useful to ensure correct coverage and identify missing tables.

On the subscriber

- **`pg_stat_subscription`:** The view gives one row per subscription, showing status and lag. This helps in detecting replication lag (`received_lsn vs replay_lsn`) and applying errors.
- **The key columns are:**
  - subid, subname → subscription identifiers
  - pid → apply worker PID
  - relid → table being synchronized (for initial sync workers)
  - `received_lsn, last_msg_send_time, last_msg_receipt_time` → WAL positions and timestamps
  - `apply_error_count` → failed apply attempts
- **`pg_subscription`:** This is the configuration metadata for subscriptions (connection string, publications). It is useful to audit where logical subscriptions exist.
- **`pg_stat_subscription_workers` (v15+):** This view provides per-worker statistics for initial sync and apply workers.

Replica lag

In logical replication setups, replication lag represents the delay between a transaction being committed on the publisher and its successful application on the subscriber. Even though logical replication offers flexibility for cross-version upgrades, selective replication, and multi-cloud distribution, the presence of lag can directly impact application consistency and user experience. It is important to recognize that logical replication involves decoding WAL changes into a logical format and transmitting them over a network before applying them on the subscriber, which increases sensitivity to factors such as network latency, system load, and apply worker throughput. Effective monitoring of logical replication lag, combined with tuning and resource planning, is therefore essential to ensure that RPO objectives are met and that distributed applications operate reliably in HA and DR environments.

Measuring replica lag

The query below can be used to measure replica lag. It is useful to measure both WAL position lag and network/apply delays.

```
SELECT

subname,

received_lsn,

last_msg_send_time,

last_msg_receipt_time,

now() - last_msg_receipt_time AS network_lag

FROM pg_stat_subscription;
```

Best practices

- Use logical replication for workload segmentation (e.g., OLTP in one region, analytics in another).
- Enabling distributed writes with conflict resolution is beneficial for bidirectional architectures.
- For multi-cloud deployments, heterogeneous environments require flexibility.
- Encrypt all logical replication traffic, particularly across regions or cloud boundaries.

Backup and recovery

Backup and recovery form the foundation of a resilient PostgreSQL deployment. No matter how robust an HA or DR architecture may be, it is always good to maintain a reliable backup strategy to protect against data corruption, user errors, hardware failures, or even large-scale outages. A well-designed backup and recovery plan ensures that organizations can meet their RPO and RTO, aligning database operations with business continuity goals.

It is beneficial to think of backup not only as an insurance policy but also as an enabler of operational flexibility. Backups make it possible to perform PITR, to provision new replicas efficiently, and to meet compliance requirements around data retention and auditability. At the same time, recovery is just as important as backup itself—validate that restoration procedures are tested, documented, and automated so that organizations can rely on them during real incidents.

PostgreSQL offers multiple backup strategies, including physical backups (such as basebackup) as well as advanced tools such as pgBackRest and Barman that support incremental backups, parallelization, compression, encryption, and centralized management. Select the right toolset based on scale, complexity, and regulatory needs, while always following best practices such as redundancy, encryption, automation, and regular testing.

Backup and recovery in PostgreSQL are not stand-alone tasks but a critical part of an organization's HA/DR best practices. By adopting a layered, tested, and secure backup strategy, enterprises can ensure resilience against data loss and operational disruption, no matter the scale or complexity of their deployments.

1. Backup

Ensuring reliable and consistent backups is one of the most critical aspects of any PostgreSQL HA and DR strategy. Backups not only protect against accidental data loss but also enable PITR and compliance with business continuity requirements. Adopt a layered approach to backups, combining different tools and strategies such as basebackup, pgBackRest, and Barman to balance performance, reliability, and operational efficiency.

Physical backup

It is good to understand that a physical backup in PostgreSQL is a byte-for-byte copy of the entire database cluster, including data files, configuration files, and transaction logs. This backup represents the full state of the system at a specific point in time. Unlike logical backups (which export only schema and data), physical backups preserve the exact internal structure, making them the foundation for HA and DR strategies.

A physical backup is typically created using tools such as pg_basebackup, Barman, or pgBackRest, all of which integrate WAL archiving. Always pair the physical backup with continuous archiving of WAL files so that the backup is not just a static snapshot but a dynamic recovery point that can be advanced to any moment in time.

Key best practices for physical backups include:

- Storing backups in multiple locations or regions to protect against site-wide disasters
- Automating backup scheduling and verification to eliminate human error
- Encrypting backup data both in transit and at rest
- Regularly testing recovery from backups to ensure that the backup process is reliable

Stand-alone backup

Define a stand-alone backup as a single, self-contained copy of the database state taken at a single moment. This can be:

- **Logical backup** (export of schema + data via pg_dump/pg_dumpall): It is portable across PostgreSQL versions, allows table-level restore, and is easy to inspect. But it can be slow for very large databases; restores are logical (replay of SQL) and may take a long time. It does not preserve physical layout or all internal objects.
- **Physical backup** (file-level copy of the PostgreSQL data directory): This may be a cold backup (database stopped) or a hot backup (live copy using pg_basebackup or file system snapshotting). It is simple and yields an immediately restorable physical image. But it requires downtime and may not be suitable for 24x7 systems.

The use of stand-alone backups is beneficial for migration, version upgrades, exporting subsets of schema/data, or short-term archival; or as base images for initializing standbys. It is good practice to automate backups and keep a catalog (timestamp, checksum, tool, retention policy), and to encrypt backups in transit and at rest. Storing copies in multiple locations can be beneficial (onsite + off-site or cloud) for geographic resiliency. Finally, it is always advisable to verify backups after they have been created (pg_verifybackup, tool-provided checks, or test restores).

Continuous archiving (WAL archiving + PITR)

Continuous archiving is best understood as the process of preserving every completed WAL segment after a base backup, so that the base image can be advanced by replaying WAL up to any desired point (PITR). This is the canonical mechanism for fine-grained recovery in PostgreSQL. It is beneficial to enable continuous archiving because it allows restoration to an exact instant (before logical errors, accidental deletes, or corruption), making it crucial for meeting strict RPO and compliance requirements.

Below are some of the recommended configuration changes that are required for continuous archiving in the postgresql.conf file:

```
wal_level = replica

archive_mode = on

archive_command = ‘test ! -f /mnt/pg_wal_archive/%f && cp %p /mnt/pg_wal_archive/%f’

archive_timeout = 60
```

Set `wal_level` at least to replica to ensure required WAL detail. archive_command should be robust and idempotent and should return success only after safe storage (use scripts that retry and verify uploads). For cloud/object storage, use secure upload tools (e.g., aws s3 cp with checksums; wrap in retry logic).

Replication slots can be used for standbys or basebackup consumers to avoid premature WAL removal, but it is also critical to monitor slots so WAL does not accumulate indefinitely.

WAL archives can be stored in durable, redundant storage as below:

- Local archival servers + asynchronous replication to offsite
- Direct upload to cloud object storage (S3/GCS/Azure Blob) with versioning and lifecycle policies

Encrypt archives and use checksums to protect against corruption, and monitor `pg_stat_archiver` to detect failed archive attempts and to alert on delays or errors. Finally, it is a good idea to run periodic test restores and PITR rehearsals to ensure that the WAL archive chain and base backup combination are usable.

Some best practices for continuous archiving:

- Have at least two independent WAL archive copies (local + cloud) for geographic redundancy.
- Make archive_command transactional and failure-resilient.
- Protect WAL archives via encryption, check-summing, and access controls.
- Remember that PITR recovery time depends on the amount of WAL to replay and hardware throughput—design retention and base backup cadence accordingly to balance RTO vs storage cost.

Incremental backup

This saves only changed data (blocks or files) since the last full (or incremental) backup, thereby reducing storage and transfer costs. PostgreSQL does not provide a simple, built-in, block-level incremental backup API in the way some file systems do. Historically, incremental-like behaviors are achieved by one of the following:

- **Backup orchestration tools** (e.g., pgBackRest, Barman) that implement incremental/differential logic on top of physical base backups and WAL
- **Storage-level snapshots** (LVM, ZFS, cloud block snapshots) that efficiently record changed blocks
- **Custom rsync/rsnapshot** workflows that synchronize changed files

Some best practices for continuous archiving:

- Use a well-supported backup manager (pgBackRest or Barman are commonly recommended) rather than ad hoc scripts for critical systems.
- Ensure that the backup manager also archives WAL and provides cataloging and verification features.
- Monitor and prune incremental chains if retention policies apply (avoid very long chains that slow restores).
- Regularly test restore of an incremental chain to validate the process under time pressure.

Backup strategy

Recommend a layered schedule, balancing durability and restore speed. For example:

- **Daily:** Full physical base backup (hot, via `pg_basebackup` or pgBackRest full) → kept for N days
- **Hourly:** Continuous WAL archiving to redundant targets (local + cloud) to enable PITR to any minute
- **Nightly or as needed:** Differential or incremental backups (via pgBackRest) to reduce storage and network usage
- **Weekly/monthly:** Offsite copy of full backup (cold archive) for regional disaster protection
- **Continuous:** Cataloging, checksum verification, and retention pruning automated via backup manager
- **Periodic:** Quarterly full restore tests across environments to validate RTO

A. Basebackup

The pg_basebackup utility is PostgreSQL's built-in tool for creating full physical backups of a cluster. It is simple to use and tightly integrated with the database. Know that `pg_basebackup` is PostgreSQL's native utility for taking a physical (file-level) base backup of a running cluster. A base backup produced by pg_basebackup is suitable as the starting point for PITR and for initializing a streaming/log-shipping standby. `pg_basebackup` uses the PostgreSQL replication protocol and takes the backup without blocking client activity, making it a foundational building block for HA and DR workflows.

Key conditions and prerequisites

Important conditions and prerequisites for basebackups include:

- **Replication permission and connection rules:** Ensure that the connection used by `pg_basebackup` is made with a user that has REPLICATION permission (or is a superuser), and that the configuration file `pg_hba.conf` permits replication connections from the backup host(s).
- **Server parameters:** Configure the server with sufficient `max_wal_senders` (so that `pg_basebackup` and any WAL streaming have available senders), and configure the parameter `wal_level = replica` (or higher) when WAL archiving or replication is required for PITR/standby creation.
- **Full-page writes for standby backups:** If a base backup is taken from a standby (to reduce load on the primary), enable the configuration parameter `'full_page_writes'` on the primary to ensure that the WAL contains the necessary full-page images for a consistent backup.
- **Disk and WAL retention planning:** It is important to ensure adequate storage for both the backup files and for WAL retention (either via `wal_keep_size`, replication slots, or WAL archiving), because missing the WAL needed to make the backup consistent will render the backup unusable.

Working of pg_basebackup

`pg_basebackup` is the native tool that needs to be used efficiently to achieve maximum performance. Important features, which help us understand the internals of the backup tool, include:

- **Replication protocol and checkpoint:** `pg_basebackup` connects over the replication protocol and instructs the server to perform a checkpoint at the start of the backup so that a consistent copy of the data directory can be produced. This checkpoint step can take time, depending on activity and `--checkpoint` setting.
- **WAL inclusion modes (-X / `--wal-method`):** Select a WAL method that matches the current environment and RPO goals. For example:
  - **`-X stream` (recommended default for many HA/DR cases):** WAL data is streamed in parallel while the backup runs. This avoids dependence on retained WAL files on the primary and typically produces a fully self-contained backup that can be started without consulting an external WAL archive. The stream method opens a second replication connection to stream WAL while the backup is running.

    - **`-X fetch`:** WAL files required for the backup are fetched at the end of the backup (relies on WAL files still present—i.e., `wal_keep_size` or archive must retain them). This method can be appropriate when network bandwidth is constrained but requires careful WAL retention planning.
    - **`-X none`:** WAL files are not included; this is rarely suitable for production unless an independent WAL archive strategy is needed and it is guaranteed to be available.
- **Replication slots and `-S` / `--slot` / `--create-slot`:** Use a replication slot when performing WAL streaming (`-X stream`) so that the primary does not remove WAL segments needed by the backup. `pg_basebackup` can create a temporary slot unless `--no-slot` is specified; alternatively a named slot can be created with `-C`/`-S` so the backup and later standby reuse the same slot settings. Monitoring and planning for WAL accumulation is important when replication slots are used.
- **Formats, compression, and manifests:** `pg_basebackup` can write a plain file hierarchy (default) or tar files (`-F t`). Use compression (`--compress` or `-z`/`--gzip`) for tar outputs to save storage and network bandwidth; note that some compression modes behave differently when WAL streaming is used. A backup manifest is produced by default; preserve that manifest and verify the backup with `pg_verifybackup`.
- **Incremental backups:** Know that `pg_basebackup` supports incremental mode via `--incremental` (server-side support), but incremental backups require post-processing (e.g., `pg_combinebackup`) before they can be used to restore. Use incremental mode only when the downstream tooling and processes are validated.
- **Performance and safety controls:** Use `--max-rate` to throttle basebackup traffic so that backups do not unduly impact primary workload. Avoid `--no-sync` in production because it skips file system syncs and can produce corrupt backups if a crash occurs. Choose `--checkpoint=fast` when short waiting time for the checkpoint is desired, but be mindful that fast can increase I/O.

Basebackup workflow

- **Prepare the server and accounts:** Create a dedicated replication user with only the REPLICATION privilege, and add an appropriate `pg_hba.conf` entry that only permits the backup host(s) to connect for replication.
- **Decide where to run the backup:** It is often beneficial to run `pg_basebackup` from a standby to reduce load on the primary, provided that one is able to validate the constraints of standby backups (e.g., some history files are not created on the standby, and a standby promotion during the backup will cause failure).
- **Choose WAL method and slot strategy:** Select `-X stream` and use a replication slot (`-S slotname` or `-C -S slotname`) when a backup needs to be self-contained and is suitable to become a standby immediately. If network or CPU constraints prevent streaming, `-X` fetch may be used, but ensure that WAL retention is sufficient on the server.
- **Run the basebackup with recommended flags (examples):** Run a command similar to:

  ```
  pg_basebackup -h primary.example.com -U repl_user -D /backups/pg_base_2025-09-29 \
  -F t --compress=gzip:6 -P -X stream -S basebackup_slot -R
  ```

  This creates a compressed tar backup, streams required WAL in parallel, uses a named replication slot (so WAL is preserved while the backup completes), and writes recovery connection settings (`-R`) so the backup is easier to promote to a standby.

  It is also a good idea to use a plain expansion (directory) backup if a file system layout is preferable:

  ```
  pg_basebackup -h primary.example.com -U repl_user -D /var/lib/pgbackup/base_2025-09-29 \

  -P -X stream -R --checkpoint=fast --max-rate=50M
  ```

  This example throttles to 50 MB/s and requests a fast checkpoint to reduce checkpoint latency.
- **Verify and archive the backup:** `Run pg_verifybackup` against the produced backup; copy backups to reliable, redundant storage (offsite/cloud); and register them in the backup catalog.
- **Provisioning a standby from the basebackup:** Restore the basebackup into the target data directory, ensure that standby.signal (or recovery.conf in older versions) and connection settings are present (if -R was used `pg_basebackup` will append them), and then start PostgreSQL so WAL replay can occur and the node becomes a standby.

Basebackup best practices

- **Prefer -X stream + replication slot for production backups:** Make the backup self-contained by streaming WAL, so that the backup can be brought up independently without waiting for external archive retrieval.
- **Use basebackup:** It is best primarily for small to medium environments or as a building block for initializing replicas.
- **Monitor replication slots and WAL storage carefully:** It is important to monitor disk usage when replication slots are used: An inactive slot can cause WAL to accumulate on the primary and exhaust storage. Implement alerts and retention policies.
- **Store basebackups:** Use reliable, redundant storage and combine them with WAL archiving so that PITR is possible.
- **Test restores regularly (PITR and full restore):** It is always best to rehearse restore scenarios on isolated environments so that RTO expectations are validated and staff are familiar with recovery playbooks.
- **Avoid using basebackup as the sole backup method:** Especially in large enterprise environments, basebackup can become resource intensive. In addition, it lacks advanced features such as compression or parallelism.

B. Pgbackrest

PgBackRest is a widely adopted backup and restore solution for PostgreSQL, designed to provide robust, high-performance, and feature-rich backup capabilities.

Pgbackrest best practices

- Use pgBackRest in enterprise environments, because it supports incremental, differential, and full backups, ensuring efficient use of resources.
- Leverage features such as parallel backup/restore, compression, and encryption to optimize storage and improve security.
- Integrate pgBackRest with PostgreSQL replication setups, as it can be used to quickly provision standby servers from backups, improving HA and DR readiness.
- Configure retention policies carefully so that old backups are pruned automatically while compliance requirements are still met.

C. Barman (Backup and Recovery Manager)

Barman is another enterprise-grade backup and recovery tool for PostgreSQL, designed to centralize backup management for multiple servers. Replication provides availability, whereas backups provide recoverability. Barman makes recoverability operationally manageable at scale by providing a centralized backup catalog, automated WAL handling for PITR, retention, and pruning policies, as well as support for geographically redundant backup topologies. Some key capabilities for Barman are backup orchestration, WAL archive ingestion, backup catalog, retention policies, verification, monitoring integration, and support for offsite/geographic redundancy.

Core concepts

Barman models backups and WAL with the following:

- **Server definition:** Barman manages one or more PostgreSQL servers configured in its barman.conf (INI) file. Each server block defines connection strings, backup_method, archive settings, and storage paths.
- **Base backups vs WAL archive:** Barman stores physical base backups (full) and continuously collects WAL segments (archive mode or streaming). The base backup + WAL chain is what enables PITR.

- **Backup catalog/metadata:** Barman records metadata for each backup (timestamp, size, WAL range, retention state). This catalog is the single source of truth for recovery operations.
- **Retention policy:** Barman enforces policies (redundancy or recovery-window based) that determine which backups are kept and which are pruned.
- **Streaming vs. archive ingestion:** Barman can receive WALs via PostgreSQL's archive_command, or it can receive WALs via a streaming interface (`barman receive-wal` / streaming archiver). Both methods are supported and can be combined for resilience.
- **Recovery operations:** Barman exposes a recover command that will restore a base backup and optionally replay WAL files to a chosen recovery target (time/LSN/XID/restore point).

Use cases

Below are some of the most common use cases used by enterprise grade applications:

- **Centralized backup management for many clusters:** Centralized backup operations and policies for multiple PostgreSQL clusters help teams manage DR consistently.
- **PITR:** Barman combines base backups and WAL archives to recover to any point in time.
- **DR across sites:** Barman nodes kept in different regions (cascading Barman or offsite replication) enable backups to survive a data-center loss.
- **Automated backup verification and monitoring:** Barman checks integrate into monitoring systems (alerts on failed backup, archive failures, disk pressure).
- **Fast standby provisioning:** Barman provisions new standbys quickly from catalogued base backups and WAL archives.

Barman CLI

- **CLI-driven:** Barman is primarily a command-line tool with a clear set of one-line actions: `barman backup <server>`, `barman recover <server> <backup-id>`, `barman list-backup <server>`, `barman check <server>`, `barman cron`, etc.
- **Configuration:** Barman uses an INI-style configuration (/etc/barman.conf plus per-server files under `/etc/barman.d/`). There are global settings and per-server sections. Typical keys include `conninfo`, `backup_method`, `basebackup_retry_times`, `retention_policy`, `streaming_conninfo`.
- **Actions:** Backup (`base/backups`), WAL archival ingestion (via `archive_command` or streaming), maintenance (`prune`, `compress`), and recovery are the core actions.

`Example config file:`

```
[mydb]

description = “Primary DB in us-east-1”

conninfo = host=primary.example.com user=barman_stream dbname=postgres

backup_method = postgres

streaming_conninfo = host=primary.example.com user=barman_stream

backup_directory = /var/lib/barman/mydb

retention_policy = RECOVERY WINDOW OF 7 DAYS
```

Important Barman workflows

- **Perform a backup:** To trigger a base backup (often via `pg_basebackup` or `rsync`) and register the resulting backup in Barman's catalog:
```
barman backup mydb
```
- **List available backups:** To list all available backup:
```
barman list-backup mydb
```
- **Check server health:** To validate connectivity, configuration, and WAL streaming/archiving:
```
barman check mydb
```
- **Recover a backup:** To restore the base backup to the target path and prepare restore_command/recovery configuration to replay WALs to the requested target:
```
barman recover --target-time "2025-10-01 09:59:59" mydb latest
/var/lib/postgresql/recovery
```
- **Stream WAL receive (if configured):** To start WAL streaming on the Barman side (depends on how streaming is set up):
```
barman receive-wal mydb
```
- **Verify backup integrity:** To detect corruption early:
```
barman verify mydb <backup-id>
```

Retention policies, pruning, and storage lifecycle

- **Retention types:** Barman supports retention by redundancy (number of backups to keep) and by recovery window (retain backups sufficient to restore to any point in the last N days).
- **Best practice:** Combine redundancy + recovery window: and keep recent daily backups for some period, weekly backups for longer, and monthly deep-archive copies offsite. Barman's retention policies can be used to automate pruning.
- **Pruning caution:** Prune only after verifying that older backups are restorable and WAL chain integrity is intact. Automated prune runs should be logged and auditable.

Barman best practices

Barman is good for managing backups for a fleet of PostgreSQL servers, as it provides centralized monitoring and administration.

- Enable WAL archiving and streaming to ensure that backups are always consistent and point-in-time recovery is possible.
- Adopt Barman's remote backup capabilities, so that backups are stored on separate infrastructure, reducing the risk of data loss due to local failures.
- Integrate Barman with alerting and monitoring systems to ensure that backup processes are validated and failures are detected early.

General best practices across backup tools

Redundancy

Maintain backups in multiple locations, including offsite or cloud storage, to safeguard against regional outages.

Testing

Test backup restoration regularly to validate that backups are usable and RTO/RPO objectives can be achieved.

Automation

Automate backup scheduling and verification so that human error is minimized.

Security

Encrypt backups and secure access to backup storage to protect sensitive data from unauthorized access.

Documentation

Maintain clear operational runbooks describing backup and recovery procedures, ensuring operational readiness during incidents.

2. Recovery and PITR

The true strength of PostgreSQL's physical backup mechanism comes from combining a base backup with the reapplication of WAL files.

- A base backup provides the static starting point—a consistent snapshot of the database cluster.
- WAL files record every change made after that snapshot.

When recovery is initiated, the database engine first restores the base backup. It then replays the WAL files sequentially, reapplying every transaction that occurred after the base backup was taken. This process ensures that the recovered database reflects the exact sequence of operations, maintaining consistency and durability.

Think of this as restoring a time machine. The base backup gives the user a fixed historical checkpoint, and WAL replay allows one to move forward in time from that checkpoint to any exact moment required.

Recovery example scenario

1. A base backup is taken at midnight.
2. WAL archiving is enabled, and WAL files are continuously shipped and stored.
3. At 10 a.m., a critical table is accidentally dropped.

With physical backup and WAL archiving in place:

- The database administrator restores the base backup taken at midnight.
- WAL files generated between midnight and 10 a.m. are replayed.
- Recovery is configured to stop just before the `DROP TABLE` command (using a recovery target time or transaction ID).

The result is a time-consistent copy of the database, as it existed just before the failure. No uncommitted transactions are applied, and the system maintains ACID integrity.

DR procedure

It is good to plan not only for routine node failures but also for the most severe scenario: the total destruction of the original PostgreSQL server (for example, due to hardware failure, natural disaster, or data center outage). In such cases, a well-tested recovery procedure ensures that service can be restored with minimal data loss and downtime.

PostgreSQL's DR relies on a combination of base backups and archived WAL files, which together allow the database to be reconstructed to a consistent state—or even to an exact point in time before failure.

Step 1: Restore the original base backup

The first action in a full DR is to restore the most recent base backup to a new PostgreSQL server. A base backup provides the foundational snapshot of the data directory. Without this, replaying WAL files alone would not be possible. Ensure that the restored base backup is from a verified backup source (e.g., pgBackRest, Barman, or `pg_basebackup` archives) and that it matches the WAL archive stream.

Best practices include:

- Restoring onto identical or compatible PostgreSQL versions and architectures
- Using checksums or built-in verification tools (`pg_verifybackup`) to ensure integrity
- Placing the restored data directory on production-grade storage with adequate IOPS and throughput to replay WAL efficiently

Step 2: Define the recovery target

Once the base backup is in place, PostgreSQL must be instructed how far to replay WAL files. This is achieved through the recovery target mechanism. From PostgreSQL 12 onward, the recovery configuration is simplified:

- Edit `postgresql.conf` to include recovery settings such as `restore_command`, `recovery_target_time`, or `recovery_target_name`.
- A simple marker filenamed `recovery.signal` must be created in the data directory.
- The presence of `recovery.signal` tells PostgreSQL to perform archive recovery when the server starts.
- **Important:** `recovery.signal` is only a trigger file; it contains no settings itself. All parameters are read from `postgresql.conf`.

Older approach (before Postgresql 12)

- In PostgreSQL versions prior to 12, recovery was driven by a dedicated configuration file called `recovery.conf`, placed in the data directory.
- This file contained all necessary recovery parameters (`restore_command`, `recovery_target_time`, etc.).
- On promotion, `recovery.conf` would be renamed to recovery.done to indicate that recovery was complete.

Recovery target options

Define the recovery target carefully:

- `recovery_target_time` → recover to a specific timestamp
- `recovery_target_lsn` → recover the database to a particular Log Sequence Number (LSN) within the WAL
- `recovery_target_xid` → recover to a specific transaction
- `recovery_target_name` → recover to a named restore point created earlier (pg_create_restore_point)
- `recovery_target='immediate'` → instructs PostgreSQL to stop replaying WAL as soon as the server reaches the earliest consistent point after the base backup—i.e., "as early as possible"
- `recovery_target_inclusive` → Specifies whether recovery stops immediately after the target (true) or just before the target (false)
- `recovery_target_action` → Defines the action when recovery reaches the target: pause, promote, or shutdown (with defaults based on target and hot standby)
- If no explicit recovery target is defined, PostgreSQL will replay all available WAL files and stop at the latest consistent point

Precise control of recovery

When hot standby is enabled, recovery can be controlled interactively after it has started. By default, recovery pauses when the configured target is reached, allowing administrators to inspect the database state before proceeding. The function:

- `pg_is_wal_replay_paused()` can be used to check whether replay is paused
- `pg_wal_replay_resume()` resumes the process

If a different recovery target is needed, the server must be stopped and the configuration must be updated and then restarted.

Step 3: Start the database server

Once the base backup is restored and the recovery configuration (`restore_command + recovery.signal` or `recovery.conf`) is set, the PostgreSQL server can be started.

During startup, PostgreSQL:

1. Detects `recovery.signal` (or `recovery.conf` in older versions)
2. Executes the configured `restore_command` to fetch archived WAL segments
3. Sequentially replays WAL entries on top of the base backup until it reaches the recovery target
4. Completes recovery, transitions to a consistent state, and allows promotion to a primary role

After successful recovery, the server is ready to accept client connections. If configured for PITR, the database reflects the exact state at the defined recovery target.

The role of restore_command

The `restore_command` is the single most important recovery parameter.

Purpose:

- `restore_command` tells PostgreSQL how to retrieve archived WAL segments from storage.
- Without a valid restore command, the recovery process cannot progress beyond the base backup.

How it works:

- On startup, PostgreSQL requests specific WAL segments by filename.
- The `restore_command` is executed for each segment, copying it from the archive to the pg_wal/ directory.
- If a requested WAL file is missing or corrupted, recovery fails.

Syntax: `restore_command = 'cp /archiveDir/%f %p'`

`%f` = requested WAL filename

`%p` = destination path in the server's pg_wal/ directory

Best practices for recovery

Recovering a PostgreSQL cluster reliably, whether after a catastrophic failure or for a targeted point-in-time correction, requires more than simply restoring a base backup. It depends on disciplined operational practices that ensure that WAL archiving, recovery procedures, and PITR workflows function predictably under pressure. By following a consistent set of recovery best practices, administrators can minimize data loss, reduce downtime, and guarantee the integrity of restored environments.

Below are key best practices that should be followed to obtain a robust, predictable, and operationally sound recovery process:

- Always validating WAL archiving, since missing or corrupted WAL files will break the recovery chain
- Using PITR not only for disaster events but also for operational fixes, such as recovering from user errors
- Testing recovery procedures in staging environments to ensure that they are well documented and executable under pressure

A physical backup provides a foundation for DR. When combined with WAL reapplication, it enables PostgreSQL to achieve time-consistent recovery, ensuring minimal data loss and fast restoration. This layered approach—base backup + WAL replay—is a best practice for any HA/DR architecture built on PostgreSQL.

PITR

PITR is one of the most powerful recovery features in PostgreSQL. Think of PITR as the ability to restore a database cluster not just to the state of the last backup but to any specific point in time between the backup and a desired recovery target. This makes PITR highly valuable for mitigating data corruption, user errors (such as accidental table drops), or logical mistakes in transactions.

PITR is enabled by combining a base backup (e.g., from `pg_basebackup`) with WAL that has been archived since that backup. PostgreSQL replays WAL files on top of the restored base backup until it reaches the requested recovery point, thus reconstructing the database state.

Key concepts behind PITR

Base backup

Always start PITR from a consistent physical base backup.

WAL archiving

WAL files record every change to the database. Configure `archive_mode` = on and an `archive_command` so that WAL files are preserved safely outside of the data directory.

Recovery target

PostgreSQL allows recovery to a particular timestamp, transaction ID, LSN (Log Sequence Number), or even a named restore point. This flexibility makes PITR adaptable to different business recovery goals.

Recovery process

During recovery, PostgreSQL replays WAL files until it reaches the recovery target, then stops or continues in read-only mode, depending on configuration.

Configuring PITR with WAL archiving

Prepare the primary for archiving: Configure the primary PostgreSQL instance to archive WAL files reliably. Some of the important parameters that need to be configured in primary postgrsql.conf are:

```
# postgresql.conf

wal_level = replica

archive_mode = on

archive_command = ‘cp %p /mnt/pgwal_archive/%f’
```

- `wal_level` = Replica ensures sufficient WAL information for PITR and replication
- `archive_command` = Copies completed WAL segments to a secure directory
- Use robust archiving methods such as rsync or cloud storage scripts with error checking.

Take a base backup: A base backup serves as the foundation for PITR.

```
pg_basebackup -h primary_host -U repl_user -D /backups/base_2025-09-29 \

        -F tar -z -X stream -P
```

Store the base backup and WAL archive in redundant and offsite storage for safety.

Simulate a failure or restore scenario: Suppose a user accidentally drops a table at 14:25. The last base backup was taken at 12:00, and WALs have been archived continuously since then. The objective is to recover the database to 14:24:59, just before the error.

Restore the base backup: On the target recovery server (or the same server after wiping the cluster directory):

```
tar -xvf /backups/base_2025-09-29.tar -C /var/lib/postgresql/data
```

It is important to ensure that ownership and permissions match PostgreSQL's requirements.

Configure recovery parameters: Create a postgresql.conf entry or use postgresql.auto.conf with archive settings, and add a recovery.signal file. Then specify recovery target options in postgresql.conf:

```
restore_command = ‘cp /mnt/pgwal_archive/%f %p’

recovery_target_time = ‘2025-09-29 14:24:59’
```

- `restore_command` retrieves archived WAL files as PostgreSQL replays them.
- `recovery_target_time` ensures that the database stops just before the unwanted change.

Other options include:

- `recovery_target_xid` for transaction ID
- `recovery_target_lsn` for exact log sequence number
- `recovery_target_name` for a named restore point created via `pg_create_restore_point()`

Start PostgreSQL in recovery mode: When PostgreSQL detects the recovery.signal file, it enters recovery mode and replays WAL until the recovery target is reached. Once complete, it will stop or continue in read-only mode.

```
pg_ctl -D /var/lib/postgresql/data start
```

Monitor `pg_wal/recovery.conf` logs for progress.

Promote the database: After verifying the database state, promote the server so that it resumes accepting writes:

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

This finalizes PITR, and the system is live at the desired recovery point.

Best practices for PITR with WAL archiving

Reliable archiving: Always use robust, error-checked archive commands and ensure that WAL archives are stored redundantly (local + cloud).

Frequent backups: Schedule base backups regularly so that recovery windows are minimized, reducing the volume of WAL that must be replayed.

Testing: Test PITR periodically on non-production systems to ensure that backup and restore procedures work as expected. It is best to involve not only DBAs but also application and infrastructure teams to ensure full end-to-end recovery validation.

Post-drill review: Review drill outcomes, capture lessons learned, and update documentation accordingly.

Retention policies: Define policies for WAL retention and cleanup, balancing storage usage with recovery needs.

Security: Encrypt both base backups and WAL archives, as they contain sensitive transactional data.

Operational runbooks: Document PITR procedures clearly so that DBAs can act quickly during incidents.

Continuous improvement: Refine recovery scripts, tooling, and documentation based on real test results and evolving infrastructure.

PITR with WAL archiving provides PostgreSQL deployments with fine-grained recovery capabilities, enabling restoration not just to the last backup but to any chosen point in time. By combining consistent base backups, reliable WAL archiving, and tested recovery procedures, enterprises can meet stringent RPO and RTO objectives while maintaining resilience against both technical failures and human errors.

DR scenario

Prepare PostgreSQL deployments not only for routine node failures but also for larger-scale disasters such as full data center outages. A single-node failure is the most common event, often caused by hardware breakdowns, operating system crashes, or network instability. At the same time, data center outages – whether from power loss, connectivity failures, or natural disasters – present a far greater risk to business continuity.

Walk through both scenarios in detail, since each requires a distinct recovery strategy. For node-level failures, automated failover within the same region provides fast recovery. For regional outages, configure cross-region replication as part of a geo-distributed DR plan. Such approaches ensure that organizations not only minimize downtime (low RTO) and data loss (low RPO) but also achieve resilience against both localized and large-scale failures.

1. Single-node failure

It is common for individual PostgreSQL nodes to fail due to hardware faults, OS crashes, or network issues. In such cases, the priority is to ensure minimal disruption and fast recovery.

- **Detection:** Use Failover Manager or PGD to continuously monitor node health. Automatic failure detection ensures faster reaction.
- **Failover:** When the primary node fails, promote a synchronous standby to the new primary. This minimizes data loss (low RPO) and ensures that applications reconnect with minimal downtime (low RTO).
- **Recovery of failed node:** After repair, reinitialize the failed node as a standby and allow it to catch up via streaming or cascading replication.
- **Best practice:** Test failover regularly and configure client connection managers (e.g., HAProxy, PgBouncer) so applications fail over transparently.

2. Data center outage

Because a complete site outage – caused by power failures, network isolation, or natural disasters – can occur, always have a geo-distributed DR strategy in place.

- **Replication across regions:** Maintain at least one standby replica in a separate region or cloud. Streaming replication is suitable for synchronous local standbys, while asynchronous replication is better across regions to balance latency.
- **Failover across sites:** When the primary data center becomes unavailable, promote the remote standby to primary.
- **Client redirection:** Use DNS failover, load balancers, or application connection managers to reroute traffic automatically to the DR site.
- **Re-synchronization:** Once the primary site is restored, rebuild it as a standby and reintegrate it into the replication topology.
- **Best practice:** Test region-wide failovers periodically to validate both RTO and RPO under real-world conditions.

3. Configuring cross-region replication for geo-distributed DR

Recognize that protecting PostgreSQL clusters against regional outages requires more than local HA; it requires extending replication across geographically distributed regions. By configuring cross-region replication, organizations gain the ability to survive full site failures while maintaining acceptable recovery point objectives

(RPO) and recovery time objectives (RTO). It is best to design such replication with considerations for latency, encryption, monitoring, and automation, since these factors directly impact both resilience and performance. A geo-distributed DR architecture not only safeguards against large-scale disasters but also provides flexibility for workload distribution, compliance with data locality regulations, and seamless business continuity across clouds or regions.

Base setup on primary region: Configure the primary cluster with `wal_level = replica`, `max_wal_senders`, and `archive_mode = on`. A dedicated replication user with least-privilege access should be created. WAL archiving to cloud/object storage is advisable as a secondary safety net.

Establishing cross-region standby: Take a base backup of the primary using `pg_basebackup` or pgBackRest and restore it in the remote region. Streaming replication can then be established with a `primary_conninfo` string in `recovery.conf` (or `standby.signal` for newer versions). For WAN connections, enable SSL/TLS encryption for replication traffic to protect data in transit.

Handling latency and mode selection: Replication can be set up in two modes: synchronous or asynchronous.

- **Synchronous replication:** This is better for zero data loss but only practical within low-latency regions.

  Asynchronous replication: This is usually more practical across continents, as it avoids latency bottlenecks while still ensuring a reasonably low RPO.

  It is a good practice to use cascading replication, so that only one standby pulls directly from the primary, reducing load and network overhead.

Automating failover and switchover: Tools such as Failover Manager are good for handling failover in a single region, but for multi-region HA/DR, EDB PGD is better suited. PGD provides bidirectional capabilities, allowing writes in multiple regions with conflict resolution and ensuring seamless failover across regions.

Validation and testing: Always simulate region-wide failures and test end-to-end recovery procedures, including application connection failover and data consistency checks. Continuous monitoring using EDB Postgres Enterprise Manager or equivalent tools helps track replication lag, WAL archive health, and cluster integrity.

Best practices for cross-region DR

- Always use encrypted connections for cross-region replication
- Maintain redundant standbys across at least two remote regions for true fault tolerance.
- Combine streaming replication with regular physical backups so that recovery options remain flexible.
- Monitor replication lag closely to ensure that RPO commitments are realistic.
- Keep SOPs and runbooks updated so that teams can act quickly during a regional outage.

Both single-node failures and regional outages can be mitigated effectively when PostgreSQL is deployed with proper HA and DR strategies. By implementing cross-region replication, encrypting traffic, monitoring performance, and regularly testing failovers, organizations can ensure resilience, meet SLA targets, and safeguard critical business operations against disasters.

Cyber repave

As organizations mature their HA/DR posture, traditional backup and failover strategies must evolve to address cyber resilience—recovering not just from infrastructure failure but from malicious or accidental data compromise. A cyber repave is the controlled process of restoring database operations from verified-clean sources while ensuring that compromised artifacts are never reintroduced into production.

In an EDB Postgres environment, this process leverages existing HA/DR building blocks—backups, WAL archives, and standby systems—but applies them with stricter verification, isolation, and credential hygiene. The following steps outline a best-practice runbook for executing a cyber repave that preserves data integrity, supports forensic analysis, and restores full operational trust in the environment.

Determining the recovery strategy: PITR or clean standby promotion

The first and most critical decision following a cyber event is to determine how to recover—whether to perform a PITR to a known pre-incident state or to promote a verified-clean standby.

- **PITR path:** This approach is chosen when both the primary and standby systems are suspected of contamination, tampering, or logical corruption. PITR enables restoration to a precise moment before the compromise by replaying WAL files up to a specific timestamp or LSN. This effectively rolls back malicious changes and recovers a clean state.
- **Clean standby promotion path:** When at least one standby replica is verified to have halted replication before the intrusion occurred, promoting that standby provides the fastest route to recovery. This method avoids restoring from scratch while ensuring data integrity.

The choice between these paths should be informed by forensic analysis, audit logs, and replication timelines. The fundamental rule remains constant: Always restore from the last known-good, uncompromised state—never from a potentially tainted environment.

Incident timeline mapping: Correlate log timestamps, monitoring alerts, and user activity to accurately identify the incident onset time. This helps determine the safe PITR point or replication cutoff.

Standby validation framework: Maintain a periodic checksum verification or hash comparison mechanism between primary and standby data blocks to simplify validation during incident triage.

Replication slot isolation: Disable or drop logical replication slots temporarily to prevent compromised data from propagating to subscribers during decision-making.

Security confirmation gate: Introduce a formal "clean node verification checklist" before promoting a standby—covering OS patches, binary signatures, and database-level checksums.

Rebuilding the environment: Golden image and infrastructure-as-code (IaC) deployment

After selecting the recovery source, the next step is to rebuild all infrastructure components from trusted, immutable assets. This guarantees that the recovered system is free from latent malware, configuration drift, or backdoor entries.

- **Golden image rebuild:** Reinstall operating systems, PostgreSQL or EDB PGD binaries, and supporting packages using pre-validated base images maintained in a secure repository.
- **Infrastructure-as-code (IaC) redeployment:** Recreate the database cluster, replication topology, network configurations, and access policies entirely from code using tools such as Ansible, Terraform, or CloudFormation.
- **Validation:** Compare package versions, configuration hashes, and dependency manifests against the baseline to confirm system integrity. This step ensures that every element of the environment—OS, middleware, and database—is reconstituted from clean, reproducible definitions. The result is a trusted operational foundation before any data restoration begins.
- **Immutable infrastructure control:** Maintain the golden images in a version-controlled registry (e.g., AWS AMIs, container images, or VM templates) with cryptographic signatures to confirm authenticity before use.
- **Dependency allow-listing:** Verify all installed libraries, plugins, and extensions against an approved manifest to prevent reintroducing tampered components.
- **Patch and version alignment:** Ensure that the rebuilt systems use the same PostgreSQL and EDB PGD versions as before the incident—or a patched minor release that fixes the exploited vulnerability.
- **Configuration drift validation:** Compare new configurations against stored baselines (via tools such as pg_ config dumps or GitOps diffs) to ensure complete consistency.
- **Automated compliance checks:** Always run compliance scanning tools post-rebuild (e.g., CIS, STIG, or internal policy scripts) to validate that the repaved environment aligns with enterprise hardening standards.

Isolated restoration and data validation: The clean-room approach (for PITR path)

If the PITR path is selected, the restoration must occur in a clean-room environment—an isolated, quarantined network disconnected from production and external access.

- **Restore operations:** Deploy the latest verified base backup and reapply WAL files to reach the desired recovery point (pre-incident timestamp or restore point).

  Validation checks:

  - Run `pg_verify_checksums` to ensure data file integrity.
  - Validate expected record counts and referential consistency.
  - Compare schema definitions and table hashes against golden references.
  - Execute critical application queries or test transactions.
- **Controlled cutover:** Once validation is complete, transition the clean environment into production by updating DNS, load balancer targets, or connection pools.

The clean-room approach eliminates the risk of reintroducing malicious or corrupted data, while providing assurance that the recovered system is operationally and cryptographically trustworthy.

Network isolation enforcement: Ensure that the clean-room environment uses isolated VLANs or VPCs with no outbound internet access and no connectivity to production.

Controlled access: Limit access to restoration nodes strictly to authorized DBAs and security analysts; enforce session logging and MFA authentication.

Parallel verification environment: Run a read-only standby in parallel during validation to compare query outputs between the clean-room restore and the last verified backup.

Extended consistency testing: Always include automated application-level regression tests and integrity scripts that verify business logic correctness (not just database structure).

Data sanitization option: For sensitive incidents, scrub or mask confidential data before using the restored environment for validation testing.

Credential and secret rotation: Restoring trust in access controls

Following recovery, credential hygiene becomes paramount. Any compromise could have exposed authentication tokens, certificates, or keys—making full rotation mandatory.

- **Database and replication users:** Regenerate all database, replication, and maintenance user credentials.
- **Encryption keys:** Rotate KMS (Key Management Service) keys and TDE (Transparent Data Encryption) master keys where applicable.
- **Certificates and tokens:** Issue new TLS certificates and SSH keys, and revoke legacy automation tokens.
- **Configuration updates:** Replace credentials in connection pools, backup scripts, and monitoring agents.

This credential and key rotation process ensures that no compromised credentials persist in the environment, fully restoring the trust boundary across all data paths and automation channels.

Centralized secret management integration: Integrate all credentials with centralized secret managers (e.g., HashiCorp Vault, AWS Secrets Manager) to enforce lifecycle and rotation policies.

Ephemeral credentials: Implement short-lived credentials and session tokens for administrative access, reducing persistent exposure.

Audit trail of rotation events: Maintain an immutable audit log of all credential rotations and access revocations for compliance verification.

Cross-system synchronization: Verify that rotated credentials are updated consistently across all integrated systems—backups, monitoring tools, replication agents, and DR clusters.

Rotation scheduling: Define a post-recovery cadence for recurring rotations (e.g., every 90 days) to prevent credential staleness.

Evidence preservation and forensic continuity

It is important to conduct forensic preservation alongside recovery to maintain compliance, legal defensibility, and support root-cause investigation.

- **Immutable snapshots:** Capture disk-level snapshots of affected data volumes, WAL directories, and system logs before any cleanup operations begin.
- **Recovery metadata:** Record precise details such as timestamps, LSNs, backup identifiers, and recovery configurations used during restoration to ensure traceability.
- **Incident archive:** Retain all backups and WAL archives covering the incident window until forensic analysis and audit reviews are completed.

This documentation enables auditors and security teams to reconstruct the timeline of the incident, verify recovery actions, and confirm that no tampering occurred during the remediation process.

Chain-of-custody logging: Maintain signed logs for each snapshot, backup, and WAL file preserved to support forensic admissibility.

Data tagging: Tag preserved artifacts (snapshots, archives) with metadata—incident ID, timestamp, and hash digest—to prevent confusion in multi-incident scenarios.

Write-once storage: Store preserved evidence in immutable object storage (e.g., WORM/S3 Object Lock) to guarantee non-alteration.

Cross-team coordination: Involve InfoSec and Legal teams early to determine retention timelines and reporting obligations (e.g., regulatory breach notification).

Memory and process capture: If possible, capture running process lists, memory dumps, and connection traces before shutdown to support root-cause analysis.

Post-restore assurance: Validation, monitoring, and observation window

Once the database environment has been restored or the standby promoted, it must operate under enhanced observation for an extended period to ensure stability and data integrity.

- **Enhanced monitoring:** Enable deeper telemetry across query patterns, connection origins, replication lag, WAL activity, and user sessions.
- **Integrity verification:** Perform daily `pg_verify_checksums` runs and consistency validation jobs throughout the observation window to confirm structural soundness.
- **Operational resilience:** Validate that replication, backup, and WAL archiving processes resume seamlessly and generate clean, new WAL streams.
- **Behavioral analytics:** Integrate anomaly detection tools to identify irregular access or unexpected query behavior.

Once the system demonstrates sustained integrity, availability, and clean replication flow, it can return to standard monitoring thresholds and operational baselines.

Baseline comparison: Compare post-recovery performance metrics and query plans with historical baselines to confirm that performance anomalies are not lingering.

Replication health automation: Schedule periodic replication consistency checks (e.g., checksum validation or row count parity checks across nodes).

Behavioral alerts: Configure anomaly-detection thresholds on connection attempts, schema changes, or DDL executions to detect post-recovery exploitation attempts.

User and role audit: Audit all database roles, privileges, and ownerships to ensure that no unauthorized accounts were created during or after the incident.

Controlled return to production: Establish a formal "stability checkpoint" (e.g., after 7 or 14 days) before declaring the environment fully normalized and resuming standard operational cadence.

The cyber repave runbook complements traditional HA/DR strategies by extending recovery capabilities into the realm of cyber resilience. Through structured decision-making, isolated restoration, infrastructure immutability, credential rotation, and forensic rigor, organizations can restore database operations confidently—even in the aftermath of a security breach.

By embedding these practices within EDB's HA/DR framework—leveraging tools such as Barman, PGD, and Failover Manager—enterprises can achieve not only operational continuity but also trusted recovery assurance, ensuring that every post-incident environment is verifiably clean, consistent, and compliant.

Conclusion

Achieving high availability (HA) and disaster recovery (DR) in PostgreSQL requires a holistic approach that combines technology, process, and people. Throughout this white paper, it has been shown that core concepts such as recovery point objective (RPO) and recovery time objective (RTO) provide the foundation for designing resilient systems, and that EnterpriseDB (EDB) tools and features play a key role in meeting these objectives.

It is best practice to use streaming replication for near real-time data protection, ensuring low RPO and fast failover capabilities. Configure failover and switchover using solutions such as EDB Failover Manager or EDB Postgres Distributed, as this makes recovery faster, automated, and predictable. Cascading replication adds another layer of scalability and efficiency, distributing read workloads while reducing pressure on the primary node. In multi-region and multi-cloud environments, adopt logical replication, which supports selective replication, workload segmentation, and resilience across heterogeneous infrastructures.

At the same time, HA/DR cannot depend on replication alone. Always maintain reliable backups using methods such as basebackup, pgBackRest, or Barman, since these tools ensure flexibility in both backup strategies and restoration options. Advanced recovery mechanisms such as point-in-time recovery (PITR) provide fine-grained protection against human errors or logical data corruption, and it is good practice to validate these recovery paths through regular demonstrations and documented recovery procedures. Such practices not only increase operational confidence but also provide valuable evidence for compliance and audit requirements.

Recognize that best practices—such as securing replication and backup traffic with encryption, monitoring replication lag and backup integrity, running recovery drills, and keeping SOPs up to date—are what transform theoretical architectures into reliable, production-ready HA/DR strategies.

In conclusion, a layered approach that combines replication, backup, recovery testing, and operational discipline is the best path to resilience in PostgreSQL environments. By adopting these best practices, organizations can ensure that critical applications remain available, recoverable, and performant even in the face of failures, thereby aligning database operations tightly with business continuity goals.

About the author

Rahul Saha
Principal Solution Architect, EDB
Connect on LinkedIn

Rahul Saha collaborates with customers to design scalable architectures, drive seamless cloud migrations, and optimize database performance across diverse and complex environments. His wide-ranging technical expertise includes deep experience across major cloud platforms such as AWS, Azure, OCI, and GCP. Over the course of his career, he has supported numerous large-scale enterprise transformations and database modernization initiatives.